TURUN YLIOPISTON JULKAISUJA
ANNALES UNIVERSITATIS TURKUENSIS

SARJA - SER. A I OSA - TOM. 446

ASTRONOMICA - CHEMICA - PHYSICA - MATHEMATICA

Memristive Computing

by

Eero Lehtonen

TURUN YLIOPISTO
UNIVERSITY OF TURKU
Turku 2012

From the Business and Innovation Development (BID) unit, and the Department of Information Technology
University of Turku, Finland

Supervisors

Adjunct Professor Mika Laiho
Business and Innovation Development (BID)
University of Turku
FIN-20014 University of Turku
Finland

Dr. Tech. Jussi Poikonen
Department of Communications and Networking
Aalto University
FIN-00076 Aalto University
Finland

Dr. Tech. Jarkko Paavola
Business and Innovation Development (BID)
University of Turku
FIN-20014 University of Turku
Finland

Reviewers

Assistant Professor Dmitri Strukov
Electrical and Computer Engineering Department
University of California, Santa Barbara
Harold Frank Hall, Rm 5153
Santa Barbara, CA 93106-9560, USA

Dr. Tech. Lauri Koskinen
Department of Micro- and Nanosciences
Aalto University
Rm ECDL I307B
PL 13000, 00076 Aalto, Finland

Opponent

Dr. Ricardo Carmona Galán
Instituto de Microelectrónica de Sevilla
CSIC-Universidad de Sevilla
Avda. Américo Vespucio s/n
41092 Sevilla, Spain

ISBN 978-951-29-5148-2 (PRINT)
ISBN 978-951-29-5147-5 (PDF)
ISSN 0082-7002
Painosalama Oy - Turku, Finland 2012

Abstract

Memristive computing refers to the utilization of the *memristor*, the fourth fundamental passive circuit element, in computational tasks.

The existence of the memristor was theoretically predicted in 1971 by Leon O. Chua, but experimentally validated only in 2008 by HP Labs. A memristor is essentially a nonvolatile nanoscale programmable resistor — indeed, memory resistor — whose resistance, or *memristance* to be precise, is changed by applying a voltage across, or current through, the device.

Memristive computing is a new area of research, and many of its fundamental questions still remain open. For example, it is yet unclear which applications would benefit the most from the inherent nonlinear dynamics of memristors. In any case, these dynamics should be exploited to allow memristors to perform computation in a natural way instead of attempting to emulate existing technologies such as CMOS logic. Examples of such methods of computation presented in this thesis are memristive stateful logic operations, memristive multiplication based on the translinear principle, and the exploitation of nonlinear dynamics to construct chaotic memristive circuits.

This thesis considers memristive computing at various levels of abstraction. The first part of the thesis analyses the physical properties and the current-voltage behaviour of a single device. The middle part presents memristor programming methods, and describes microcircuits for logic and analog operations. The final chapters discuss memristive computing in large-scale applications. In particular, cellular neural networks, and associative memory architectures are proposed as applications that significantly benefit from memristive implementation. The work presents several new results on memristor modeling and programming, memristive logic, analog arithmetic operations on memristors, and applications of memristors.

The main conclusion of this thesis is that memristive computing will be advantageous in large-scale, highly parallel mixed-mode processing architectures. This can be justified by the following two arguments. First, since processing can be performed directly within memristive memory architectures, the required circuitry, processing time, and possibly also power consumption can be reduced compared to a conventional CMOS implementation. Second, intrachip communication can be naturally implemented by a memristive crossbar structure.

Acknowledgements

The research leading to this thesis was funded by University of Turku and the Graduate school in Electronics, Telecommunications and Automation (GETA). Further financial support was provided by the Nokia foundation, the Finnish foundation for technology promotion (Tekniikan Edistämissäätiö, TES), and the Fulbright foundation.

Several people have had a major impact on the completion of this work. First of all I want to thank doctor Mika Laiho for supervising this work. His guidance and active participation in the research leading to this thesis have been invaluable. I wish to thank doctor Jussi Poikonen also for supervision, co-authoring of several papers, and of all the help I received when refining the draft of this thesis. I am grateful to doctor Jarkko Paavola for supervising my research work, and especially for the guidance he gave me in the beginning of my graduate studies.

I wish to thank professor Ari Paasio, the director of BID Technology, for providing the excellent working conditions for my research work. My gratitude goes also to my friends and colleagues Tero Hurnanen, Jari Tissari, and doctor Tero Jokela for the great atmosphere we have in our work place. I wish to express my gratitude for many people who have had a positive impact to my work at University of Turku, especially Mikko Jalonen, Peter Virta, professor Valery Ipatov, professor Juhani Karhumäki, doctor Alexey Dudkov, doctor Ilpo Lahti, and professor Jouni Isoaho. I wish to thank professor Wei Lu for co-authoring two joint publications, and professor Jennifer Hasler for many useful advice I have received during our collaboration. Also, I wish to thank the pre-examiners of this thesis, professor Dmitri Strukov and doctor Lauri Koskinen, for their insightful comments and prompt reviews.

I had a privilege to visit the Redwood Center for Theoretical Neuroscience at University of California, Berkeley, during the academic year 2011/2012. I wish to thank professor Bruno Olshausen for the invitation and the excellent working conditions, and doctor Pentti Kanerva for his guidance and friendship, and for introducing me to the fascinating field of cognitive computing.

Finally I want to thank my family, my parents Jouko and Pirkko, and my sister Anna for the example, love, and guidance they have given to me throughout all my life. I thank my beloved Iina for all her love and support, and Kaapo, just for being who he is.

Contents

List of Symbols

The following lists the symbols used in this thesis. As can be seen, some symbols, such as e, have multiple meanings. The meaning of such a symbol should be clear from its context.

$\rightarrow$	Material implication operation
$\not\leftarrow$	Converse nonimplication operation
a	Crystal periodicity
A	Set of auxiliary memristors (stateful logic)
$\mathbf{A}$	SDM address matrix
a_j	Auxiliary memristor (stateful logic)
$A(i, j; k, l)$	A-template of a CNN
α	Constant of the generic analog memristor model
α	Tilting angle of a nanowire crossbar w.r.t. CMOS layer
α_S	Constant for the Schottky current equation
α_T	Constant for the tunnelling current equation
B	Subset $\{0, 1\}$ natural numbers
$B(i, j; k, l)$	B-template of a CNN
β	Constant of the generic analog memristor model
β_S	Constant for the Schottky current
β_T	Constant for the tunnelling current
C	Network capacity of an associative memory
$C(\cdot)$	Capacitance
$\mathbf{C}$	SDM content matrix
$\mathbf{d}$	SDM distance vector
e	Euler's number
e	Unit charge
E	Energy
E	Local electric field
E_0	Characteristic electric field of a crystal
$\mathcal{E}$	Average electric field
η	Constant of the generic analog memristor model
F	Fabrication feature size

GND	Ground voltage
$G(\cdot)$	Conductance
γ	Number of memristors on a segmented nanowire crossbar
$H(\cdot)$	Heaviside function
$i(\cdot)$ or I	Electric current
$i(V)$	Current at voltage V
$I(m)$	Current through memristor m
k_B	Boltzmann constant
L	Length
$L(\cdot)$	Inductance
$L(\cdot)$	Logistic map
λ	Constant of the generic analog memristor model
M	Vector capacity of an associative memory
m	Memristor
$M(\cdot)$	Memristance
μ	Ionic mobility
N	Number of CMOS cells in a CMOL circuit
N	Number of neurons in an associative memory
$P(\cdot)$	Power
P	Set of input memristors (stateful logic)
p_i	Input memristor (stateful logic)
φ	Magnetic flux
π_k	An AND-clause of non-inverted input variables
q	Electric charge
$Q = \{q_1, q_2\}$	Set of work memristors (stateful logic)
R	Set of result memristors (stateful logic)
$R(\cdot)$	Resistance
R_{OFF}	Maximum resistance of a bistable memristor
R_{ON}	Minimum resistance of a bistable memristor
R_0	Reference resistance in a stateful logic circuit
r_k	Result memristor (stateful logic)
s	Switch
$S_3(\cdot)$	Three-input parity function
σ_k	An OR-clause of non-inverted input variables
t	Time
T	Time scale
$\mathcal{T}$	Temperature
Θ	Threshold value
$\mathbf{u}$	Input vector (associative memory)
$\mathbf{v}$	Stored vector (associative memory)
$v(\cdot)$ or V	Electric voltage
v_{cond}	Conditional voltage (stateful logic)

V_{DD}	High rail voltage
$V(m)$	Voltage across a memristor
v_{prog} or V_P	Programming voltage
v_{read} or V_R	Read voltage
v_{set}	Set voltage (stateful logic)
V_{SS}	Low rail voltage
$v_\epsilon(T)$	ϵ-threshold voltage at time scale T
V^{T-}	Negative threshold voltage of a memristor
V^{T+}	Positive threshold voltage of a memristor
V^T	Threshold voltage of a memristor, assuming $V^{T-} = -V^{T+}$.
w or $\mathbf{w}$	Memristor's state variable
W	Willshaw memory matrix
$w_{\max}$	Maximum value of a state variable
$w_{\min}$	Minimum value of a state variable
$\mathbf{y}$	SDM activation pattern
$\mathbf{z}$	Input vector (associative memory)

Chapter 1

Introduction

1.1 Motivation

A *memristor* is a passive two-terminal resistive component, whose resistance changes as a function of the voltage across or the current through it. Its existence was first postulated by Leon Chua in 1971 [17], but at that time all the known physical emulations of this device required an internal power supply. It took nearly 40 years until in 2008 the first passive realization of a memristor was reported by HP Labs [104]. This publication ignited massive interest in the field of memristor research, even though physical memristors had been studied at least for a couple of decades by then. The connection between these strangely behaving components and the original theoretical definition by Chua just had not been previously discovered.

The apparent strangeness in the electric behaviour of memristors is that their I-V curves form pinched hysteresis loops, and the forms of these loops depend on the amplitudes and frequencies of the input voltage signals. This phenomenon can be formally treated by defining a state variable, which determines the memristor's instantaneous resistance also known as the *memristance*. For thin-film memristors such as the device reported by HP Labs, the state variable corresponds to a statistical measure of the configuration of dopant ions inside the memristor, which can be relocated by applying an external electric field. Since relatively large electric fields are required to move ions within the memristive material, the dimensions of a thin-film memristor must be in the order of nanometers.

Memristors are typically formed within a nanowire crossbar in order to facilitate the fabrication process. Nanowires can be patterned for example by e-beam lithography process, and when memristive material is stacked between the nanowire layers, it follows that a memristor is in self-aligned manner formed at each crosspoint of two mutually perpendicular nanowires [43]. A memristive memory architecture can then be implemented by comple-

menting a memristive crossbar with active CMOS circuitry for selecting and driving the nanowires of the crossbar [43]. The fabrication of digital memories is the driving force of memristor technology, since very dense memory architectures can potentially be manufactured. Moreover, from the perspective of memory technology, it is very advantageous that memristors are practically non-volatile, which means that they retain their states even when unpowered. In recent press releases it has been stated that memristive memory should become commercially available in 2014 [78], while a similar resistive memory technology called the *phase change memory* is already currently available for mobile device manufacturers [79].

This thesis focuses on memristive computing. The main idea here is to take advantage of the physical characteristics of memristors in order to enable processing that is difficult or area consuming to realize with pure CMOS circuitry. To date, various physical implementations of memristors have been reported. They can be binary, as the devices presented in [57,104], meaning that they have two distinct states of resistivity. Some of the reported memristors have small numbers of discrete states [36,44], while some are analog meaning that their memristances can be changed in a continuous fashion [13,32]. Each of these different classes of memristors can be used in different computational applications. For example, analog memristors can be used for continuous arithmetic operations as is explained in Chapter 5. On the other hand, binary memristors can be used to perform so-called *stateful logic* as discussed in Chapter 6, which allows for direct implementation of logic computing within memristive crossbars. These elementary operations can be used to implement more involved applications, such as the ones presented in Chapters 8 and 9. Since the field of memristive computing is rather new, most of the contents of this thesis can be regarded as basic research forming a basis for future work.

The CMOS/memristor hybrid architecture discussed in Chapter 7 allows the memristors to be used both as memory units and as programmable connections between different parts of a CMOS chip. This makes memristors well-suited for parallel processing applications, in which CMOS processing units communicate with each other via programmable nanowire crossbars. In this thesis, such parallel processing systems are described in Chapters 8 and 9, which among other memristor applications discuss implementations of various artificial neural network architectures.

The claim I make in this thesis is that memristive computing will be advantageous in large-scale, highly parallel mixed-mode processing architectures. A justification for this claim is that since part of the processing can be performed within memory, the processing time, required circuitry, and also possibly power consumption of the system can be reduced.

Memristive architectures are ideally suited for computation within a memory, and thus memristors should not be regarded only as memory, but

also as nanoscale computing units. An analogy of memristive computation can be found in biology, where local memory units called synapses perform a major part of the computation realized in the nervous system. A synapse acts as a memory unit and a communication link between neurons, but it also performs significant computational tasks — a role similar to that of a memristor in memristive computing. As the reduction in the size of CMOS transistors will eventually cease, it is crucial to investigate new computing architectures beyond the conventional von Neumann paradigm. I believe that memristors will allow for processing with scale and speed currently unavailable in CMOS computing architectures.

1.2 Organization of the thesis

In the following, I review the rest of the chapters in this thesis, and refer to the relevant original publications.

- **Chapter 2. Memristor Fundamentals**

  Theoretical definitions and general properties of different classes of memristive systems are presented. A definition of the memristor for the purposes of this thesis is given.

- **Chapter 3. Thin-Film Memristor**

  The fabrication and characteristics of different thin-film memristors are considered. A SPICE model of a generic analog memristor model is presented.

- **Chapter 4. Memristor Programming**

  The I-V behaviours of memristive devices are investigated. Closed-form expressions for the switching time, power, and energy are derived for the generic memristor model. The so-called threshold voltage phenomenon for programming the state of a memristor is described. Different methods for progamming the state of a memristor are presented. Related publications:

  [56] M. Laiho, E. Lehtonen, A. Russell, P. Dudek: *Memristive synapses are becoming reality*, an article in the newsletter of the Institute of Neuromorphic Engineering, November 2010

  My contributions: Modelling and simulating the analog memristor.

  [66] E. Lehtonen, J. H. Poikonen, M. Laiho, W. Lu: *Time-Dependency of the Threshold Voltage in Memristive Devices*, Proc. IEEE International Symposium on Circuits and Systems 2011, Rio de Janeiro, May 2011.

My contributions: All of the theoretical results and simulations presented in the first four sections of the paper. The results presented in the fifth section were derived in collaboration with Poikonen.

- **Chapter 5. Analog Arithmetic Operations**

  The implementation of elementary arithmetic operations — addition, subtraction, multiplication, and division — with analog memristors is described. A memristor circuit implementing these operations is presented. Related publications:

  [53] M. Laiho, E. Lehtonen: *Arithmetic Operations within Memristor-Based Analog Memory*, Proc. 12th IEEE CNNA – International Workshop on Cellular Nanoscale Networks and Applications, Berkeley, February 2010.

  My contributions: Formulation of the memristor model used in the paper, and its mathematical analysis.

  [55] M. Laiho, E. Lehtonen, W. Lu: *Memristive Analog Arithmetic Within Cellular Arrays*, Proc. IEEE International Symposium on Circuits and Systems 2012, Seoul, May 2012

  My contributions: Formulation of the translinear principle used for memristance multiplication, and the signed addition on memristances. SPICE simulations of the memristive arithmetic unit.

- **Chapter 6. Memristive Implication Logic**

  Logic computing with memristors is considered. Memristors are naturally suited for performing stateful logic operations, of which the so-called *material implication* operation can be most straightforwardly implemented. Various methods for synthesizing an arbitrary Boolean function with implication logic are presented. Related publications:

  [58] E. Lehtonen, M. Laiho: *Stateful implication logic with memristors*, Proc. IEEE/ACM International Symposium on Nanoscale Architectures 2009, pp. 33 – 36, July 2009

  My contributions: Major contributions to the text and theoretical analysis, including the derivation of the synthesis method, and the simulations of the computational sequences.

  [62] E. Lehtonen, J. H. Poikonen, M. Laiho: *Two Memristors Suffice to Compute All Boolean Functions*, IET Electronic Letters, Vol. 46, Iss. 3, pp. 239 – 240, February 2010.

  My contributions: Formulation of the recursive conjunctive form. Proof of this form was discovered by Poikonen, and we derived the presented formulation together.

[89] J. H. Poikonen, E. Lehtonen, M. Laiho: *On Synthesis of Boolean Expressions for Memristive Devices Using Sequential Implication Logic*, accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits

My contributions: Assisted in formulating the reduction method described in Section III. Discovery of the multi-input implication form.

[64] E. Lehtonen, J. H. Poikonen, M. Laiho: *Implication Logic Synthesis Methods for Memristors*, Proc. IEEE International Symposium on Circuits and Systems 2012, Seoul, May 2012

My contributions: Derivation of the NAND-OR synthesis method and a reformulation of the synthesis method presented in [62] in the case of multi-input implication logic.

- **Chapter 7. Memristive Crossbars**

  Large-scale memristive crossbar arrays are discussed. Two different approaches for interfacing memristive crossbars with CMOS circuitry are presented: the demultiplexer architecture and the so-called *CMOS / molecular hybrid* (CMOL) architecture. Inherent limitations of a memristive crossbar are discussed, and read and write operations within a crossbar are described.

- **Chapter 8. Memristor Applications**

  Various applications of memristive circuits are presented, including digital memories, field programmable gate arrays, parallel stateful logic circuits, chaotic circuits, and neural and cellular neural networks. Related publications:

  [60] E. Lehtonen, M. Laiho, J. H. Poikonen: *A Chaotic Memristor Circuit*, Proc. 12th IEEE CNNA – International Workshop on Cellular Nanoscale Networks and Applications, Berkeley, February 2010.

  My contributions: All of the new results presented in this paper.

  [59] E. Lehtonen, M. Laiho: *CNN Using Memristors for Neighborhood Connections*, Proc. 12th IEEE CNNA – International Workshop on Cellular Nanoscale Networks and Applications, Berkeley, February 2010.

  My contributions: An enhanced model of the memristor presented in [116]. Major contributions to the text.

  [67] E. Lehtonen, J. H. Poikonen, J. K. Poikonen, M. Laiho: *Grayscale CNN Computation of Boolean Functions*, Proc. First IEEE Latin American Symposium on Circuits and Systems, Iguassu Falls, February 2010.

My contributions: Theoretical derivation of the synthesis method for Boolean functions using grayscale CNN processors.

[61] E. Lehtonen, J. H. Poikonen, M. Laiho: *A CNN Approach to Computing Arbitrary Boolean Functions*, Proc. IEEE International Symposium on Circuits and Systems 2010, Paris, June 2010.

My contributions: All of the theoretical considerations and results.

[54] M. Laiho, E. Lehtonen: *Cellular Nanoscale Network Cell with Memristors for Synapses and Local Implication Logic*, Proc. IEEE International Symposium on Circuits and Systems 2010, Paris, June 2010.

My contributions: Small contributions in the theoretical aspects of memristor modelling and implication logic.

[63] E. Lehtonen, J. H. Poikonen, M. Laiho: *Applications and Limitations of Memristive Implication Logic*, Proc. 13th IEEE CNNA – International Workshop on Cellular Nanoscale Networks and Applications, Turin, August 2012.

My contributions: Discovery of the converse nonimplication operation, and formulation of stateful logic in memristive crossbars.

- **Chapter 9. Memristive Associative Memories** Memristive implementations of various associative memory architectures are presented. A related publication:

  [65] E. Lehtonen, J. H. Poikonen, M. Laiho, P. Kanerva: *Memristive associative memories*, submitted to IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2012.

  My contributions: Theoretical foundations of the memristive implementations of the ACAM, Willshaw, and SDM memories. Co-design of the CMOS cell used for these memories.

This thesis is then organized as follows. Chapters 2 and 3 provide the basic notions and insights into the theory of memristors and fabrication of physical devices, respectively. Chapter 4 describes different methods for programming memristors. These methods are used for the elementary computational operations presented in Chapters 5 and 6. Chapter 7 describes the interfacing of memristive crossbars to CMOS circuitry. Various applications of memristive computing are discussed in Chapters 8 and 9. Finally, Chapter 10 concludes this thesis.

The aim of this work is to present a thorough analysis of memristive computing. The scope of this thesis spans from the investigation of single device characteristics to the considerations relevant in large-scale applications.

Chapter 2

Memristor Fundamentals

In this chapter the central topic of the thesis, the *memristor*, is defined. First some circuit theoretical notions are fixed, and the fundamental circuit elements are reviewed. The second part of this chapter is devoted to the theory of memristors, beginning in Leon Chua's definition of a memristor from 1971. Then a more general class of components called memristive systems is investigated, and finally a new definition for a memristor is proposed.

2.1 Fundamental notions

In this section some fundamental quantities and functional relations in electronics and circuit theory are defined.

2.1.1 Circuit variables

Electric charge q is a physical property of matter which causes it to experience a force when near other electrically charged matter. The fundamental unit of electric charge is the magnitude of the charge of an electron or a proton, and has in SI units the approximate value [80]

$$e = 1.602176487(40) \times 10^{-19} \text{ coulombs.} \tag{2.1}$$

The *electric current* i through a closed surface is defined as the first derivative of the charge, *i.e.*,

$$i = \frac{\mathrm{d}q}{\mathrm{d}t} \quad \text{or} \quad q = \int_{t_1}^{t_2} i\mathrm{d}t. \tag{2.2}$$

The *electric voltage* v is the electrical force that would drive an electric current between those points. More precisely, electric voltage is the electrical

potential energy per unit charge. For a test charge q_0 whose potential energy is U_0, the electric voltage [119] equals

$$v = \frac{U_0}{q_0}. \tag{2.3}$$

The *magnetic flux* φ is defined as the time integral of the electric voltage:

$$\varphi = \int_{t_1}^{t_2} v \mathrm{d}t \quad \text{or} \quad v = \frac{\mathrm{d}\varphi}{\mathrm{d}t}. \tag{2.4}$$

In the following, the prefixes *electric* and *magnetic* are omitted, and the above defined quantities are addressed as the charge, the current, the voltage, and the flux.

2.1.2 Functional relations

The four circuit variables can be ordered in a diagram shown in Figure 2.1. A passive circuit element is called *fundamental*, if it cannot be written as a network of other circuit elements. The relations represented by the vertical lines in the diagram denote the definitions of the circuit variables given above, while the diagonal lines and the bottom horizontal line represent the relations given by the three familiar fundamental two-terminal circuit elements: the *capacitor*, the *inductor*, and the *resistor*. These relations can be mathematically written as

$$\mathrm{d}q = C(v)\mathrm{d}v \qquad \text{(capacitor)} \tag{2.5}$$

$$\mathrm{d}\varphi = L(i)\mathrm{d}i \qquad \text{(inductor)} \tag{2.6}$$

$$\mathrm{d}v = R(i)\mathrm{d}i \qquad \text{(resistor)} \tag{2.7}$$

When the proportionality factors C, L and R are constants, the corresponding circuit elements are linear.

2.2 Memristor

2.2.1 Chua's 1971 memristor

In the seminal paper [17], Chua introduced a new fundamental two-terminal circuit element, which is called *flux-charge memristor* in this thesis. Its existence was conjectured due to the previously missing relation between the flux and the charge, therefore yielding the defining relation

$$\mathrm{d}\varphi = M(q)\mathrm{d}q \tag{2.8}$$

represented by the top horizontal line in Figure 2.1. The multiplicative term $M(\cdot)$ is called the *memristance function*. Notice that $M(q) = \mathrm{d}\varphi/\mathrm{d}q$, which shows that a memristor is defined by a functional relation between φ and q.

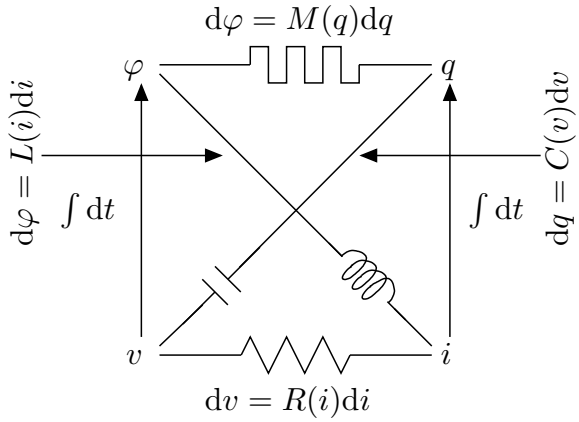

Figure 2.1: The four circuit variables connected by the fundamental circuit elements.

Dividing both sides of (2.8) by $\mathrm{d}t$ one obtains

$$v = M(q)i. \tag{2.9}$$

For a constant M, equation (2.9) is nothing but the defining relation of a linear resistor. However, with a non-constant M it describes a resistor with a memory, more precisely a resistor whose resistance depends on the amount of charge that has passed through the device. A typical response of a flux-charge memristor to a sinusoidal input is depicted in Figure 2.2. The fundamentality of the flux-charge memristor can also be deduced from this figure, as it is impossible to make a network of capacitors, inductors and resistors with an I-V behaviour forming a pinched hysteresis curve [18].

The following results are proved in [17].

Theorem 1. *A flux-charge memristor is passive if and only if its incremental memristance $M(q)$ is nonnegative;* i.e.*, if and only if $M(q) \geq 0$.*

Theorem 2. *A one-port containing only flux-charge memristors is equivalent to a flux-charge memristor.*

Theorem 3. *Any network containing only flux-charge memristors with positive incremental memristances has one, and only one, solution.*

2.2.2 Memristive system

In 1976, Chua and Kang generalized the original definition of a memristor to a more general class of dynamical systems called *memristive systems* [18]. An nth-order current-controlled memristive one-port is represented by

$$\begin{cases} v &= R(\mathbf{w}, i, t)i \\ \dot{\mathbf{w}} &= f(\mathbf{w}, i, t) \end{cases} \tag{2.10}$$

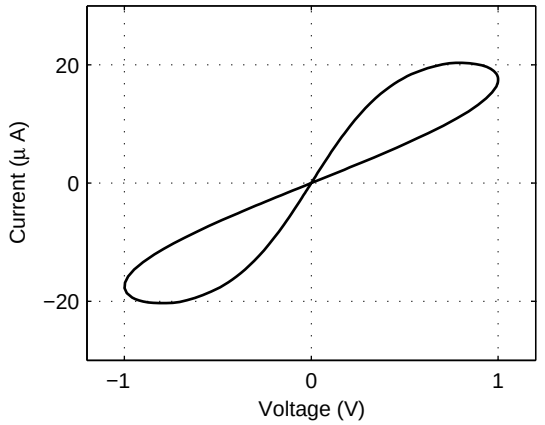

Figure 2.2: An I-V curve of a flux-charge memristor whose memristance function is linear $M(q) = \alpha q$, driven by a sinusoidal voltage input. The pinched hysteresis loop is a typical response to a periodic voltage input.

where $\mathbf{w} \in \mathbb{R}^n$ is the n-dimensional *state variable* of the system, and $\dot{\mathbf{w}}$ is its time derivative. Similarly, the nth-order voltage-controlled memristive one-port is defined as

$$\begin{cases} i & = G(\mathbf{w}, v, t)v \\ \dot{\mathbf{w}} & = f(\mathbf{w}, v, t). \end{cases} \tag{2.11}$$

A flux-charge memristor is a one-dimensional current-controlled time-invariant memristive system, for which

$$v = M(w)i \quad \text{and} \quad \dot{w} = \dot{q} = i, \tag{2.12}$$

where $w = q$ and $M(\cdot)$ is the memristance function.

Chua and Kang noted that memristive systems are capable of modelling for example such systems as the thermistor, and the Hodgkin-Huxley circuit model of the nerve axon membrane [29]. They also proved various properties for time-invariant memristive systems, the most important for this thesis being the no-energy discharge property: a time-invariant memristive system is passive if and only if $R(\mathbf{w}, i) \geq 0$ everywhere.

2.2.3 Memristor

In the previous subsection the I-V relationship of a memristive system was defined as

$$v = R(\mathbf{w}, i, t)i \quad \text{or} \quad i = G(\mathbf{w}, v, t)v. \tag{2.13}$$

For actual physical memristors manufactured so far this formulation is somewhat unnatural as there is no linear instantaneous relation between voltage

and current, and therefore for example one should define

$$R(\mathbf{w}, i, t) = R'(\mathbf{w}, i, t)/i \tag{2.14}$$

in order to get rid of the term i in (2.13). Since $R'(\cdot)$ is the function one is actually interested in, it is more convenient to devise a new definition which uses R' explicitly instead of R.

In the following I propose, for the purposes of this thesis, a definition for the memristor, which closely relates to the physical realizations discussed in Chapter 3. It should be noted that this definition does not generalize the flux-charge memristor, but rather defines a different subset of the set of memristive systems. Rationales for this definition are that the majority of the physical thin-film devices reported so far are not flux-charge memristors, and that the set of memristive systems is too broad to be simply called the set of memristors. The particular characteristics of the proposed memristor and its similarity and difference to the flux-charge memristor are discussed in the subsequent examples. For the definition, the following two elementary notions of real-valued functions are needed:

Let $f(x, y)$ be a real-valued two-variable function and let $c \in \mathbb{R}$ be fixed. Then

$$f|_{y=c}(x) : \mathbb{R} \to \mathbb{R}, \quad f|_{y=c}(x) = f(x, c)$$

is the *restriction* of f to the subset $\{(x, c)|x \in \mathbb{R}\}$. A one-variable function $f(x)$ is called *increasing*, if $x \leq y$ implies $f(x) \leq f(y)$.

Definition 4. A *memristor* is a dynamical system

$$\begin{cases} \dot{w} &= f(w, v) \\ i &= g(w, v), \end{cases} \tag{2.15}$$

with the following properties:

(i) for all values of w, $f(w, 0) = g(w, 0) = 0$

(ii) the restrictions

$$f|_w(v), \quad g|_w(v), \text{and} \quad g|_v(w)$$

are increasing for all values of v and w.

The variable $w \in \mathbb{R}$ is called the *state variable* of the memristor.

Remark 5. *From the monotonicities of $g|_w(v)$ and $g|_v(w)$ it follows that the state variable w is a measure of conductance of a memristor. The monotonicity of $f|_w(v)$ guarantees that positive voltages increase and negative voltages decrease a memristor's conductivity.*

Remark 6. *Definition 4 describes a voltage controlled memristor, as $\dot{w}$ is a function of v. Similarly one could define a current controlled memristor as*

$$\begin{cases} \dot{w} &= h(w,i) \\ v &= k(w,i), \end{cases} \tag{2.16}$$

for which $h(w,0) = k(w,0) = 0$ and the restrictions $h|_w(i)$, $k|_w(i)$, and $k|_i(w)$ are all increasing functions.

However, if $g_w(v)$ in Definition 4 is strictly *increasing and thus bijective, then a voltage controlled memristor can be seen as a current controlled memristor. Indeed, now $g|_w^{-1}(i)$ is also strictly increasing, and*

$$\begin{cases} \dot{w} &= f(w,v) = f(w, g|_w^{-1}(i)) \\ v &= g|_w^{-1}(i). \end{cases} \tag{2.17}$$

Section 3.4 describes a generic model of a physical analog memristor, which is widely used in the examples of this thesis. In this model, $g_w(v)$ is a strictly increasing function. Therefore it can be seen either as a voltage controlled or as a current controlled memristor, where the functions f and g are chosen to model the dynamics of a physical memristor.

Theorem 7. *A memristor is nonvolatile and passive. Moreover, it cannot be written as a network of capacitors, inductors and resistors.*

Proof. Nonvolatility follows from $f(w,0) = 0$. Since $g(w,0) = 0$ and $g|_w(v)$ is an increasing function, it follows that

$$v \cdot i = v \cdot g(w,v) \geq 0,$$

which proves the second claim. The third claim follows from the fact that a flux-charge memristor with an increasing memristance function $M(q)$ is a memristor. □

Example 8. *The sets of memristors and flux-charge memristors are incomparable. Indeed, choosing a non-linear g with respect to voltage in Definition 4 results in a memristor which is not a flux-charge memristor. On the other hand, a flux-charge memristor can provide negative differential resistance which is a forbidden characteristic of a memristor.*

Remark 9. *Why is the memristor in this thesis not defined as a generalization of the flux-charge memristor? Although logically appealing, this choice would not have lead to the simplicity of Definition 4 and its powerful corollaries of Theorem 7. The memristor is defined as it is to cover the physical realizations discussed in the following chapter.*

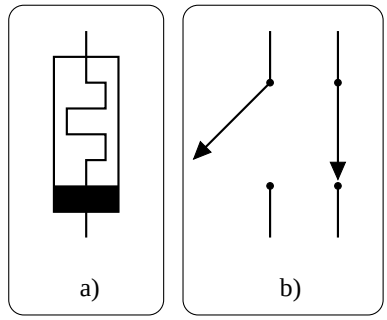

Figure 2.3: Memristor symbols used in this thesis.

Remark 10. *A flux-charge memristor is defined by function $\varphi = \varphi(q)$ that relates the flux to the charge. This is not the case with the memristor according to Definition 4. Indeed, $f(w, v)$ can be a very non-linear function of v, and therefore w does not have to be a function of φ (or q). This points out why the memristors in this thesis are not characterized by their $\varphi - q$ relationship, but rather by their I-V curves, when the input voltage is specified.*

Example 11. *GeSbTe alloy based phase-change memory [52] is not a memristor. This is because its conductance depends on the magnitude, not polarity, of the input voltage. A short, high magnitude pulse freezes the material to an amorphous state with high resistance. A longer pulse with medium magnitude of voltage is used to re-crystallize the material, yielding a low resistance. An even lower magnitude of voltage is used to read the state of the memory.*

On the contrary to the above, a memristor's conductance increases with an input voltage of one polarity and decreases with an input voltage of the other polarity. Still, a phase-change memory is a memristive system: it has a state which depends on the history of the input voltage and defines the conductivity of the system.

Although in Definition 4 the state variable is allowed to take any real value, in the rest of this thesis it is assumed to be normalized to the unit interval $w \in [0, 1]$, where $w = 0$ corresponds to the non-conductive off-state, and $w = 1$ corresponds to the conductive on-state. An example of an explicit, strictly increasing and continous normalization function is $N(x) = \arctan(x)/\pi + 1/2$. In physical realizations the off-state of a memristor is always slightly conductive, that is, $w \geq w_{\min}$ for some small constant $w_{\min}$.

Remark 12.

1. *Two different symbols for a memristor are used in this thesis. The symbol shown in Figure 2.3 a) is used in most cases. The symbol*

in Figure 2.3 b) is used when memristors with only two states are considered. A memristor in a low-conductance state is represented by an open switch, while a memristor in a high-conductance state is represented by a closed switch.

2. *The terms I-V behaviour or I-V curve are used throughout this thesis. The former means the overall dynamical relationship between current and voltage in a memristor, while the latter refers to a specific curve obtained from choosing some voltage input.*

Chapter 3

Thin-Film Memristor

A vast majority of physical memristors presented so far are thin-film solid state devices, and therefore it is appropriate to devote a chapter of this thesis to investigate their properties. Although resistive switches with thin-film structure have been observed and studied since the 1980s [36], it was not until 2008 that researchers from HP Labs announced to have found the "missing memristor" in rutile cross-point switches [104]. This article had a great impact on the research of memristive systems, as it was the first paper to associate the theory of memristors with thin-film solid state devices. After this initial breakthrough, a multitude of different thin-film implementations of memristors have been proposed, for example in [13, 33, 35, 36].

This chapter is organized as follows. First, in Section 3.1, common physical properties of thin-film memristors are investigated. In Section 3.2 the HP Labs' TiO_2 memristor is surveyed more closely, and its mathematical model is examined. Next, in Section 3.3 other implementations of thin-film memristors and their models are presented. Finally, in Section 3.4 a simplified and generic analog thin-film memristor model based on the discussed physical devices is derived. A SPICE netlist of this model is presented in Section 3.4.1. This model will be used in the rest of the thesis as a prototype model of a thin-film memristor.

3.1 Foundations of thin-film memristors

A schematic of a thin-film memristor cross section is presented in Figure 3.1. The device is sandwiched between a top and and a bottom electrode. Between the electrodes is an insulator or a semiconductor layer, inside which there are conducting filaments, whose number and lengths define the conductance of the device. The filaments consist of dopant ions or vacancies of ions, and their positions can be changed by applying a voltage between the top and the bottom electrodes.

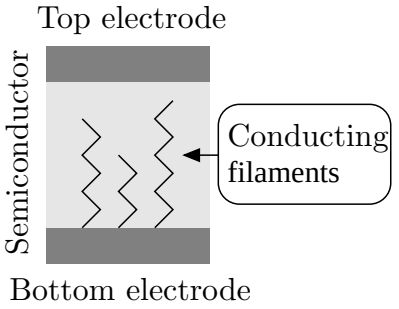

Figure 3.1: The cross section of a thin-film memristor.

Memristors can be divided roughly into two categories by their dynamical behavior. They can be either digital, which means that their states can assume a few different values, or analog if their states can take any value in a seemingly continuous range. A digital memristor, whose state can have only two distinct values is called *binary* or *bistable*.

3.1.1 Static electron transport

Metal/semiconductor contacts generally fall into one of two categories: they are either rectifying Schottky-type contacts in the case of low doping, or nonrectifying ohmic contacts in the case of heavy doping [87]. Assuming that the conducting filaments grow from the bottom electrode towards the top electrode as in Figure 3.1 it follows that the semiconductor contact at the bottom electrode can be assumed to be ohmic.

The semiconductor contact at the top electrode is Schottky type if the vicinity of this contact is devoid of conducting filaments. In this case it is said that the thin-film memristor is in off-state. The current i_S through the memristor can now be expressed in the form [87]

$$i_S \approx \alpha_S(\exp(\beta_S v) - 1), \tag{3.1}$$

where v is the voltage across the device, and α_S and β_S are constants.

If the conducting filaments penetrate the Schottky electron barrier, the memristor is said to be in on-state, where the electron transport is dictated by the tunnelling phenomenon through a thin residual barrier [116]. The tunnelling current i_T can be modeled [87] as

$$i_T \approx \alpha_T \sinh(\beta_T v), \tag{3.2}$$

where α_T and β_T are constants, and v is the voltage across the memristor.

In general, the current through the device can be written as a linear combination of the currents presented in (3.1) and (3.2), as a variable number of conducting filaments approach and penetrate the Schottky barrier.

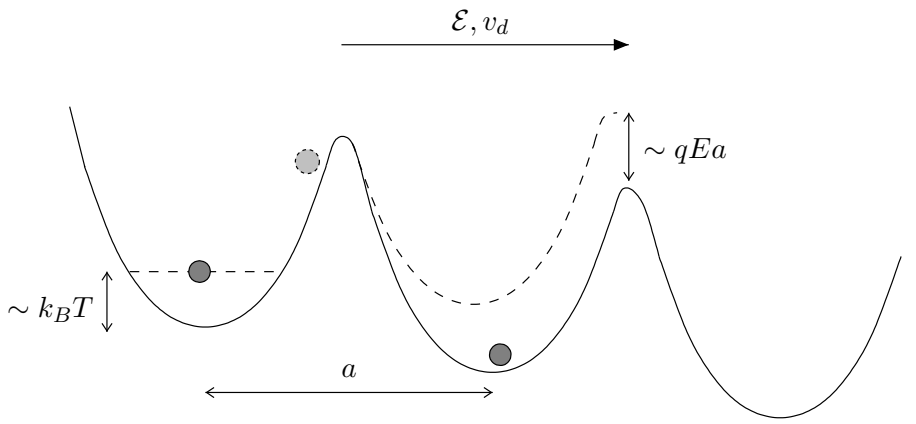

Figure 3.2: The rigid point-ion model with an applied electric field. Thermal heating lifts the ground state of the ions by k_BT, while the local electric field decreases the depth of the potential well in the direction of the applied field by $qEa/2$.

Therefore, the total current through the memristor can be written as

$$i \approx c_S(w)i_S + c_T(w)i_T, \tag{3.3}$$

where the state variable $w \in [0, 1]$ describes the effective influence of the conducting filament front, and $c_S(\cdot)$ and $c_T(\cdot)$ are some non-negative coefficient functions. With a suitable choice of coefficient functions the current in (3.3) satisfies requirements of Definition 4.

In [33] it was shown that the semiconductor contact at the top electrode can be prepared to be non-rectifying already at fabrication, thus weakening the coefficient α_S in (3.1) and making the total current of the device approach the tunnelling current of (3.2).

3.1.2 Ionic drift

Since a conducting filament consists of ions or ion vacancies, changing the state of the memristor means moving ions or vacancies within the semiconductor part of the memristor. In the following a model of ionic drift originally presented in [106] and [35] is investigated.

A rigid point-ion model for ionic crystals is illustrated schematically in Figure 3.2 for positive mobile ions. A drifting ion moves from one potential well to the next, where the net potential is determined by the constituent ions of the crystal.

An applied voltage V creates a local electric field E inside the material which changes the depth of the potential wells by $\sim qEa/2$, where a is the periodicity of the crystal, or the length between two consecutive potential

wells. The local electric field E is an effective quantity and can be much, for example 100 times [106], stronger than the calculatory average electric field $\mathcal{E} = V/L$, where L is the length of the semiconductor layer of the memristor. The overall effect of the field E on the average ionic drift velocity v_d can be written as

$$v_d \approx \begin{cases} \mu E & E << E_0 \\ \mu E_0 e^{E/E_0}, & E \sim E_0 \end{cases} \tag{3.4}$$

where μ is the ionic mobility at small electric fields, and $E_0 = 2k_B\mathcal{T}/(qa)$ is the characteristic electric field of the crystal [106]. Here k_B is the Boltzmann constant, and $\mathcal{T}$ is the temperature measured in Kelvins.

When the magnitude of the local electric field E is much smaller than E_0 the ionic drift behaves linearly. On the other hand, when E is of the same order or larger than E_0, the ionic drift depends exponentially on it. A typical magnitude for the characteristic field is $E_0 \sim 1$ MV at the room temperature $\mathcal{T} = 300$ K [106]. From (3.4) it can be seen that as the magnitude of the local electric field changes from 10 to 40MV, the drift velocity increases by more than 10 orders of magnitude. For a local field $E = 40$ MV the applied electric field $\mathcal{E}$ can still be well below the breakdown value of the semiconductor crystal.

In addition to the above, power dissipation in the small volume of the memristor yields significant heating, which in turn increases the ionic drift velocity. The overall effect makes the ionic drift a random Poisson process, whose rate is exponential or even super-exponential with respect to the applied voltage V [35]. This significant nonlinearity of the drift velocity makes it possible for a device to be practically nonvolatile, and still have a switching time in the nanosecond scale [106].

3.2 TiO_2 based memristor

The HP Labs' memristor [104, 116] consists of platinum electrodes enclosing a TiO_2 / TiO_{2-x} semiconductor bulk. In the following the fabrication process of this digital memristor is briefly reviewed. First, a crystal of rutile TiO_2 is annealed to create an oxygen-deficient layer near the bottom surface of the semiconductor [116]. Even a relatively minor stoichiometric ratio of 0.1% in TiO_{2-x} is enough to make the oxygen-deficient layer highly conductive, as oxygen vacancies in TiO_2 act as n-type dopants [106].

The conducting filaments are formed by applying a high voltage across the device. This *forming step* induces a permanent change to the oxide film by means of electroreduction [116]. In addition to creating the conducting filaments, it also produces a remnant tunnelling gap between the filaments and the top electrode [86].

After the forming step, the device is ready to be used as a memristor. In the on-state, the I-V behaviour of the memristor is dominated by the tunnelling phenomenon, while in the off-state its I-V curve is rectifying [116]. Since the oxygen vacancies are positively charged, positive voltages across the memristor tend to turn it off, while negative voltages can be used to turn the device on. The Pt / TiO_2 / Pt memristors have on/off conductance ratios of the order 1×10^3 [116]. Their switching behaviour is found to be insensitive to the device size from 5×5 μm^2 to the litography constricted size of 50×50 nm^2 [86].

When modeling the device, the state variable w of the TiO_2 based memristor can be chosen to be a normalized tunnelling gap width between the conducting filaments and the top electrode. The actual magnitude of the tunnelling gap modulation is a few nanometers.

The following equation describes the I-V behaviour of the TiO_2 based memristor [116]:

$$i = w^n \alpha_T \sinh(\beta_T v) + \alpha_S(\exp(\beta_S v) - 1), \tag{3.5}$$

where n, α_T, β_T, α_S, and β_S are positive constants.

The dynamics of the state variable w are modelled in [86] as

$$\dot{w} = \begin{cases} \frac{f_{\text{off}}}{w_c} \sinh\left(\frac{i}{i_{\text{off}}}\right) \exp\left[-\exp\left(w - \frac{a_{\text{off}}}{w_c} - \frac{i}{b}\right) - w\right], & i > 0 \\ \frac{f_{\text{on}}}{w_c} \sinh\left(\frac{i}{i_{\text{on}}}\right) \exp\left[-\exp\left(-w + \frac{a_{\text{on}}}{w_c} + \frac{i}{b}\right) - w\right], & i < 0 \end{cases} \tag{3.6}$$

where w_c, $f_{\text{on/off}}$, $i_{\text{on/off}}$, $a_{\text{on/off}}$, and b are material and dimension dependent positive constants.

Equations (3.5) and (3.6) satisfy the requirements of Definition 4. Clearly, both $\mathrm{d}i/\mathrm{d}w$ and $\mathrm{d}i/\mathrm{d}v$ are non-negative. Moreover, also $\mathrm{d}w/\mathrm{d}v$ is non-negative, since i is an increasing function of v, and therefore so is $\sinh(i/i_o)$, where i_o is either i_{on} or i_{off}.

3.3 Other memristor implementations

3.3.1 Analog memristor

In [13], Chang et al. investigate a tungsten based analog memristor. This device consists of a top palladium electrode, a tungsten oxide switching layer and a bottom tungsten electrode.

The fabrication process is roughly described as follows. First, tungsten is deposited on a thermally oxidized silicon substrate by sputtering at room temperature. The middle layer is obtained by partly annealing the tungsten film, forming thus a tungsten oxide (WO_3) film. Finally, the top palladium nanowire is formed by e-beam lithography. The reported $Pd/WO_3/W$ structure has dimensions 130nm $\times$ 130nm.

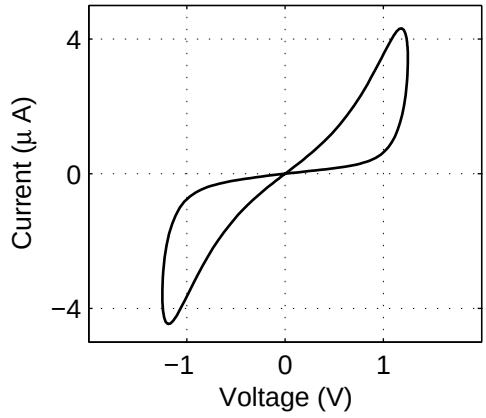

Figure 3.3: The I-V curve of the tungsten oxide-based analog memristor model with sinusoidal input voltage whose amplitude is 1.25V and frequency is 1Hz.

An important asset of this fabrication process is that it avoids electroforming. A Schottky barrier is formed between the WO_x film and the W bottom electrode. Application of a positive voltage accross the device results in a drift of the oxygen vacancies toward the bottom electrode thus resulting in an ohmic-like contact dominated by a tunnelling current. The state variable $w \in [0, 1]$ represents the normalized area index of the conducting region. More precisely, $w = 0$ indicates fully Schottky-dominated conduction while $w = 1$ indicates fully tunnelling-dominated conducting.

The defining equations of the memristor are

$$i = (1 - w)\alpha_S[1 - \exp(-\beta_S v) + w\alpha_T \sinh(\beta_T v) \tag{3.7}$$

$$\dot{w} = \lambda[\exp(\eta_1 v) - \exp(-\eta_2 v)], \tag{3.8}$$

where $\alpha_{S,T}$, $\beta_{S,T}$ and $\eta_{1,2}$ are positive constants. The rate of change of the state variable, described by (3.8), does not depend on w since in this model the existing conducting regions do not affect the formation of new conducting regions. An I-V curve of this memristor model is depicted in Figure 3.3.

Remark 13. *Recent results [2, 117] indicate that some analog memristors may operate by modulating the length and width of a single filament. This would allow for analog memristors with footprints of the same order than those of digital memristors. In [2] first results concerning an analog memristor with an active area less than* 20 nm^2 *were presented.*

3.3.2 M/a-Si/p-Si -memristor

In [36] Jo and Lu propose a memristor with a top metal — for example, silver — electrode, an active amorphous silicon layer, and a heavily doped p-type crystalline silicon layer as the bottom layer. The apparent advantage of such a structure is that only the standard CMOS process is required for the device fabrication. As with the analog memristor, electroforming is not required during fabrication of a M/a-Si/p-Si device. The resistance switching behaviour is explained by metal filament formation inside the a-Si matrix; the top electrode's metal ions drift inside the active layer towards the p-Si bottom electrode at positive applied voltages. The conducting filament size in M/a-Si/p-Si devices is found to be much smaller than the 50 nm$\times$ 50 nm metal electrode size limited by the standard lithography process, which implies that in principle even smaller devices can be fabricated using nanoimprint lithography.

The conductance of this digital memristor cannot be accurately programmed in a continous fashion, but it does have multibit capability; different resistivity levels can be programmed by controlling the maximum write programming current. The programmed resistivity levels differ from each other in an exponential rather than a linear fashion in contrast to the analog memristor discussed in the previous subsection.

Jo and Lu note that that the observed switching behaviour is insensitive to the fabrication method of the a-Si layer, but that the on/off conductance ratio does depend on the method. For example, a low pressure chemical vapor deposition process results in a conductance ratio 10^7. Moreover, two different types of switching behaviour, rectifying and nonrectifying, are possible for a M/a-Si/p-Si device in the on-state. The type of the switching behaviour depends on the fabrication method and the thickness of the a-Si layer.

3.3.3 Rectifying memristor

As noted above, the M/a-Si/p-Si memristor can be fabricated to exhibit diode characteristics. The rectifying behaviour is explained in [44] by the motion of the silver ions inside the a-Si matrix. When the device is in the on-state and a negative voltage is applied over it, the conducting filament within the a-Si matrix is partially retracted thus significantly suppressing the device conductance. The partially retracted mobile silver ions can be readily injected again to the a-Si/p-Si interface by applying a small positive bias. The actual programming of the memristor takes place at much larger absolute voltages.

In Figure 3.4, an I-V curve of a simplified model of this memristor is represented. It is assumed that the memristor has two states which corre-

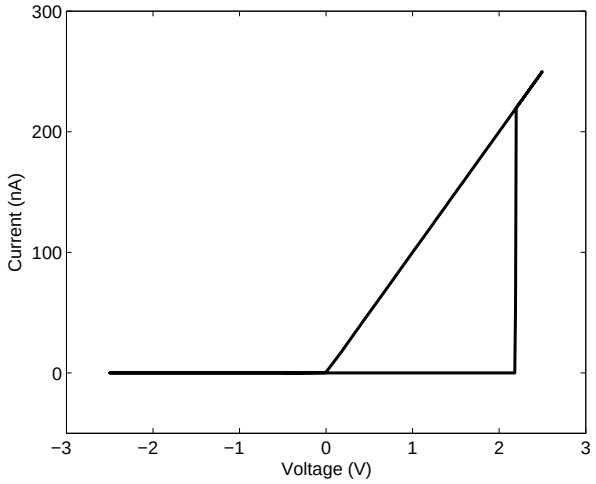

Figure 3.4: I-V curve of the rectifying memristor model. On-resistance of the device equals 10 MΩ, and the off-resistance is 10 TΩ. The device switches from off-state to on-state at approximately 2 V, and from on-state to off-state at approximately −2 V. Negative current is rectified with the plotted voltages below 1 pA.

spond to resistance values $R_{ON} = 10$ MΩ and $R_{OFF} = 10$ TΩ so that the resistance ratio $R_{OFF}/R_{ON} = 10^6$, as was reported in [44]. Furthermore, the device is assumed to switch from the off-state to the on-state at $V = 2$ V, and from the on-state to the off-state at $V = -2$V. The rectifying ratio of the device is assumed to be 10^6.

3.4 Generic model of an analog thin-film memristor

In this section a generic memristor model based on the previous survey of thin-film memristive devices is presented. To simplify the analysis of digital memristive circuits, in this thesis a digital memristor is generally modelled as a discrete-time binary switch:

$$w(t+1) = \begin{cases} 0, & v < V^{T-} \\ 1, & v > V^{T+} \\ w(t), & V^{T-} \leq v \leq V^{T+} \end{cases} \tag{3.9}$$

where v is the voltage across the device at time $t + 1$. The voltage values $V^{T-} < 0$ and $V^{T+} > 0$ are called the *negative and positive threshold voltage* of the memristor, respectively. The threshold voltage phenomenon is

properly analyzed in Chapter 4. When a more elaborate digital memristor model is required, for example the model for the TiO_2 based memristor can be used. Also the generic memristor model discussed in the rest of this chapter can be applied, when suitable parameters are chosen.

In the following a generic analog memristor model adopted from our publication [66] is presented. As was noted in Subsection 3.3.2, during the fabrication process it is possible to limit the Schottky effect at one of the interfaces making the I-V behaviour of a memristor depend mainly on the effective tunnelling barrier. Diminishing the rectification of a memristor makes its I-V behaviour more symmetrical regardless of its state. This property is beneficial in some applications, as is later seen for example in Chapter 5.

The following assumptions on the analog memristor model are made:

- The current-voltage behaviour of the device is dominated by the tunnelling phenomenon.
- The state variable $w \in [0, 1]$ represents the normalized area index of the conducting region — for example, the number of conducting filaments in the vicinity of the tunnelling barrier.
- The drift velocity of charged mobile ions in the active layer of the memristor is assumed to depend exponentially on the electric field, and thus, voltage. Moreover, the area of the conducting region does not affect the drift velocity of the mobile ions.

Definition 14. Given positive constants α, β, λ, and η, the generic analog memristor model is defined by the following equations:

$$i = w\alpha \sinh(\beta v) \tag{3.10}$$

$$\dot{w} = \lambda \sinh(\eta v), \tag{3.11}$$

with the constraint that if $w = 0$ and $v < 0$, or if $w = 1$ and $v > 0$, then $\dot{w} = 0$. In other words, the state variable w is hard-limited to the interval $[0, 1]$.

Remark 15. *The parameters α, β, λ, and η are determined by material properties of the modeled memristor, such as the barrier height for tunnelling, the effective tunnelling distance in the conducting region, and interface effects. However, in the following they are regarded as fitting parameters that yield I-V curves qualitatively similar to experimental data.*

Remark 16. *The generic analog memristor model satisfies the requirements of Definition 4. Indeed, for a fixed $w \in [0, 1]$, the current and $\dot{w}$ are increasing functions of voltage. On the other hand, for fixed voltage, the current is directly proportional to the state variable w.*

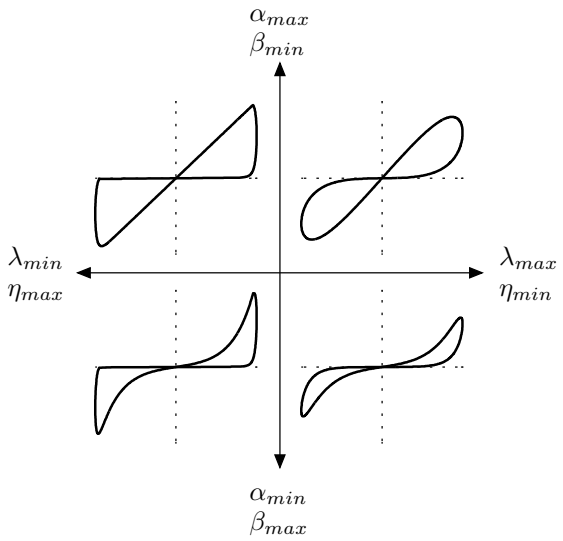

Figure 3.5: The generic analog memristor model with four choices of constants α, β, λ and η.

In the following example, the consequences for different choices of the constants α, β, λ and η are investigated.

Example 17. *Let the input voltage be sinusoidal with frequency 1MHz and amplitude 2V, and let the maximum current be 10μA and let w operate within the interval* $[0.01, 1]$. *Consider the following values for the constants:*

$$(\alpha, \beta) = (\alpha_{max}, \beta_{min}) \quad or \quad (\alpha, \beta) = (\alpha_{min}, \beta_{max}),$$
$$(\lambda, \eta) = (\lambda_{max}, \eta_{min}) \quad or \quad (\lambda, \eta) = (\lambda_{min}, \eta_{max})$$

where

$$\alpha_{min} = 4.2 \times 10^{-7}, \quad \alpha_{max} = 5 \times 10^{-4}, \quad \beta_{min} = 0.01, \quad \beta_{max} = 2,$$
$$\lambda_{min} = 0.06, \quad \lambda_{max} = 1 \times 10^{6}, \quad \eta_{min} = 1, \quad \eta_{max} = 10.$$

These values do not reflect any physical limits of the constants but rather serve simply as choices which yield visibly different I-V curves. In Figure 3.5 the I-V curves for the four possible different choices of the constants $(\alpha, \beta, \lambda, \eta)$ are depicted, while Figure 3.6 illustrates the operation of the state variable w for the two possible different choices of (λ, η).

Since α and λ can be regarded as scaling factors, it suffices to investigate the effect of the constants β and η on the behavior of the memristor model. The magnitude of the constant β defines how linear or exponential the I-V

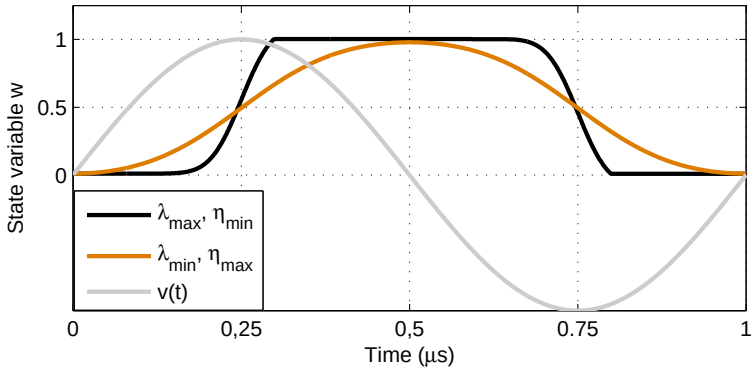

Figure 3.6: The dependence of the value of the state variable w on the input voltage $v(t)$. Depicted are the cases $(\lambda_{max}, \eta_{min})$ and $(\lambda_{min}, \eta_{max})$ and the phase of the input voltage whose amplitude is 2V.

relationship is:

$$\sinh(\beta v) \approx \begin{cases} \beta v, & \text{for } \beta v \approx 0, \\ \exp(\beta v)/2, & \text{for } \beta v > 1. \end{cases} \tag{3.12}$$

Similarly, the magnitude of the constant η defines how linear or exponential $\dot{w}$ is with respect to the input voltage.

The four different I-V curves of Figure 3.5 are summarized as follows:

- In the top right quarter of Figure 3.5 both β and η are close to zero, thus yielding a completely linear model. In other words this I-V curve corresponds to a flux-charge memristor whose memristance function M is linear with respect to charge.

- In the top left quarter, the constant β is close to zero but η is large. The resulting model corresponds to a memristor whose I-V relationship is linear for a fixed state variable w, but for which the change of the state variable depends exponentially on the input voltage. Such a device would make an ideal *binary memristor*, since it essentially has two states, and moving from one state to the other occurs abruptly when the magnitude of the input voltage is large enough. At the same time measuring the state of the memristor using a voltage of small magnitude would be easy due to the linearity of current w.r.t. voltage.

- In the bottom left quarter both β and η are large, thus yielding a completely exponential model. Again, the memristor has two distinct states, and moving from one state to the other occurs abruptly. However, in this model also the I-V relationship is exponential. The reader

may find some similarity in the behaviour of this model and the TiO_2 based memristor.

- Finally, in the bottom right quarter, β is large and η is close to zero. This makes the I-V relationship exponential, while the state variable w changes rather linearly with respect to the input voltage. If there exists a threshold voltage below which the memristor's state can be measured without disturbing it, then such a device could serve as an analog memristor.

3.4.1 A generic SPICE model

To conclude this chapter, a SPICE model of the above described generic analog memristor model is presented. The state variable is modeled as charge trapped in the capacitor Csv. The state variable changes due to the current coming from the voltage dependent current source Gsv. The amount of current coming from this source depends on the voltage between nodes T and B, which denote the top and bottom electrode of the memristor, respectively. The auxiliary functions are used to keep the state variable within the interval $[0, 1]$. Finally, between nodes T and B there exists a voltage dependent current source which is denoted by Gmem. Its value depends on the state variable and the voltage between T and B. A circuit diagram of this model excluding the auxiliary functions is depicted in Figure 3.7.

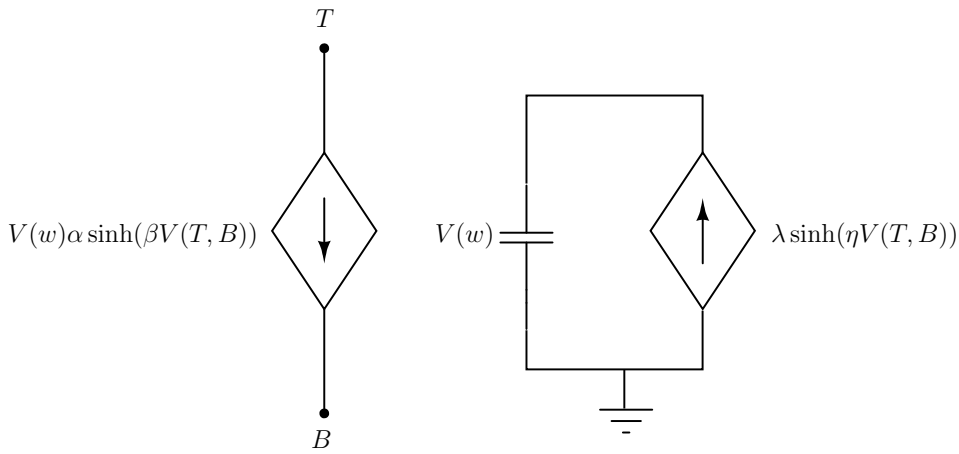

Figure 3.7: A circuit equivalent of the generic SPICE model. The voltage dependent current sources model the I-V behavior of the memristor and the rate of change of the state variable. The auxiliary functions limiting the value of the state variable — which is represented by the voltage $V(w)$ across the capacitor — are omitted here for simplicity.

The SPICE netlist of the generic memristor model is as follows:

```
.SUBCKT memristor T B w PARAMS:
+alpha=5e-4 beta=0.01 lambda=1e6 eta=1 lmin=0.01 lmax=1
+svinit = 0.01

*T and B are the top and bottom electrodes, respectively
*w is the state variable limited by lmin and lmax
*svinit is the initial value of the state variable

*State variable
Gsv 0 w value = {lambda*sinh(eta*V(T,B))*trunc(V(w),V(T,B))}
Csv w 0 1
.IC V(w) {svinit}

*Output
Gmem T B value = {V(w)*alpha*sinh(beta*V(T,B))}

*Auxiliary functions:
.func sign2(var) = {(sgn(var)+1)/2}
.func trunc(var1,var2) = {(sign2(var1-lmin)+sign2(var2))*
(sign2(lmax-var1)+sign2(-var2))/2}
.ENDS memristor
```

Chapter 4

Memristor Programming

In order to use memristors for computing, it is important to be able to rigorously change their states of conductance. Generally, this means either programming a memristor's state variable to a certain value, or changing its value by a certain amount. The programming task requires knowledge on the switching behavior of the memristor, for example it is essential to know the relationship between the input voltage and the switching time of the device. Also, when reading the memristor's conductance one needs to operate with a sufficiently low voltage or short time scale in order to keep the memristor's state unperturbated.

With these requirements in mind, the first part of this chapter is devoted to defining some basic quantities such as the switching time and energy of a memristor. While these concepts apply for arbitrary memristors, closed-form solutions are provided only for the generic memristor model. Applying the theory described in the first section, the second part of this chapter is devoted to memristor programming methods. The main references for this chapter are our publications [66] and [53].

4.1 Switching behavior of a memristor

In the examples of this section, the generic memristor model with the parameters $\alpha = 4.2 \times 10^{-7}$, $\beta = 2$, $\lambda = 0.06$, and $\eta = 10$ is assumed. An I-V curve of this model corresponding to a sinusoidal input voltage is depicted in Figure 4.1.

Remark 18. *In the simulations presented in this chapter, and the rest of this thesis, the non-memristive components are modeled as follows. A transistor symbol, such as the one depicted in Figure 4.6, in a circuit diagram implies that a Berkeley Short-channel IGFET Model (BSIM) [93] version 4.6 is used in the corresponding simulation. A switch symbol — see,* e.g. *Figure 4.8 — implies that an ideal switch model of a transistor is applied.*

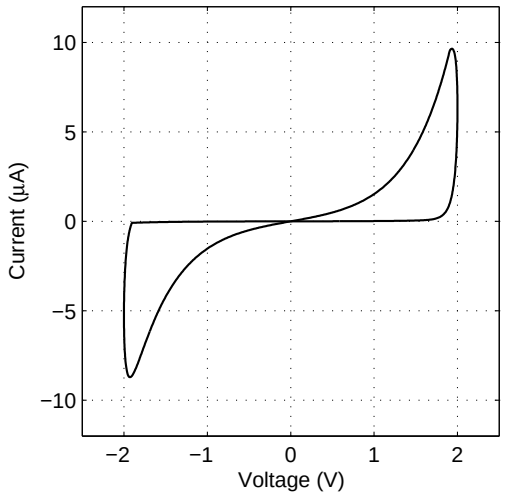

Figure 4.1: An I-V curve of the generic memristor model used in the examples. The input sinusoidal voltage has frequency 1MHz and amplitude 2V.

Operational amplifiers are simulated by an ideal high-gain model with finite output voltages. Wire resistances and capacitances have not been taken into account in the simulations.

The aforementioned ideal assumptions are made to keep the emphasis of the simulations on memristive dynamics.

4.1.1 Switching time and energy

First, the time and the amount of energy required to program a memristor's state using a fixed voltage v_{prog} are derived. Suppose a memristor is initially in the state $w = w_1$, and that it is to be programmed to the state $w = w_2$. Since the rate of change of the generic memristor model's state variable

$$\dot{w} = \lambda \sinh(\eta v) \tag{4.1}$$

is constant for the fixed input voltage v_{prog}, the switching time $T(w_1 \to w_2)$ can be written as

$$T(w_1 \to w_2) = \frac{w_2 - w_1}{\lambda \sinh(\eta v_{\text{prog}})}. \tag{4.2}$$

The assumption of a constant $\dot{w}$ at a given voltage is an ideality, which is not true for some physical devices, for example the TiO_2-memristor disussed in Subsection 3.2. For such devices the expression for the switching time must be derived by using a more appropriate memristor model. Note that if the

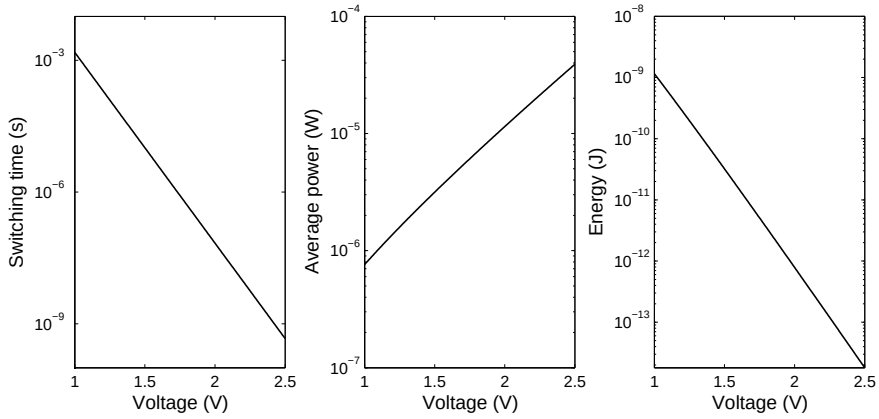

Figure 4.2: Programming the state variable from $w_1 = 0$ to $w_2 = 1$ with a constant voltage. In the subfigures the switching time, average power, and energy are plotted for the generic memristor model with the parameters $\alpha = 4.2 \times 10^{-7}$, $\beta = 2$, $\lambda = 0.06$, and $\eta = 10$.

evaluation of (4.2) results in a negative value, the polarity of the voltage is incorrect, and the switching time is infinite.

Assuming that the switching energy is consumed in Joule heating, the average power consumed during the switching process equals

$$\overline{P(w_1 \to w_2)} = \overline{v(t)i(t)} = v_{\text{prog}} \frac{w_1 + w_2}{2} \alpha \sinh(\beta v_{\text{prog}}) \tag{4.3}$$

again by virtue of $\dot{w}$ being constant. Multiplying Equations (4.2) and (4.3), the expression for the switching energy is obtained:

$$E(w_1 \to w_2) = \frac{1}{2} v_{\text{prog}} (w_2^2 - w_1^2) \frac{\alpha \sinh(\beta v_{\text{prog}})}{\lambda \sinh(\eta v_{\text{prog}})}. \tag{4.4}$$

From Equations (4.2) and (4.3) it is seen that the switching time decreases and average switching power increases exponentially with voltage. Switching energy, however, may either decrease or increase with the programming voltage v_{prog}. The exact behavior of the switching energy depends on the choice of the parameters α, β, λ, and η.

Example 19. *When $\beta < \eta$, the switching energy decreases exponentially with the programming voltage v_{prog}. This result is in accordance with the reported behavior of the TiO_2 memristor [86]. In Figure 4.2 the switching time, average power, and energy of the generic memristor model with the parameters fixed at the beginning of this section are plotted for $v_{prog} \in [1\,V, 2.5\,V]$.*

Example 20. *When $\lambda = 0$, the memristor model corresponds to a non-linear resistor. In this case the switching time and energy given by Equations 4.2 and 4.4 are infinite, as they should be.*

4.1.2 Threshold voltage

Next, the memristive *threshold voltage* phenomenon reported for example in [33,104,116] is investigated. In short, a threshold for the magnitude of the voltage across the memristor has been observed below which the devices do not significantly change their state and above which they can be programmed to a different state. As already noted in the first publication considering the TiO_2-based memristor [104], the threshold voltage is dynamical; any voltage across a memristor will change its state, but the rate of change depends on the voltage across the device. Thus, fixing the time scale of operation defines some soft voltage region within which the state variable w will stay for all practical purposes stationary. Here it should be noted that some physical devices, such the one investigated in [4], have so non-linear switching dynamics that their states stay practically unchanged, when a small enough voltage bias is set across them.

In the following the threshold voltage phenomenon is quantitatively analyzed for the generic memristor model. Originally these results were reported in [66]. Generally, a memristor has two threshold voltages, one for both polarities. Since in the generic memristor model the rate of change of the state variable is an odd function of voltage, the negative threshold voltage has equal magnitude to the positive threshold voltage.

Definition 21. Let $\epsilon > 0$ and $T > 0$ be fixed. Let $w(0)$ and $w(T)$ be the values of the state variable at time $t = 0$ and $t = T$, respectively. If the memristor is programmed with a constant voltage v_ϵ for the time T and

$$w(T) - w(0) = \epsilon,$$

then the voltage v_ϵ is called the *ϵ-threshold voltage at time scale T*.

In other words there is a memristor parameter dependent function $v_\epsilon(T)$ which relates a threshold voltage v_ϵ to a given time scale T. From Equation (4.1) one solves the ϵ-threshold voltage for the generic memristor model as

$$\frac{\mathrm{d}w}{\mathrm{d}t}T = \lambda \sinh(\eta v_\epsilon)T = \epsilon \tag{4.5}$$

$$\implies \quad v_\epsilon = \frac{1}{\eta}\sinh^{-1}\left(\frac{\epsilon}{\lambda T}\right). \tag{4.6}$$

Example 22. *Assuming the generic memristor model with the parameters $\alpha = 4.2 \times 10^{-7}$, $\beta = 2$, $\lambda = 0.06$, and $\eta = 10$, the 0.01-threshold voltage*

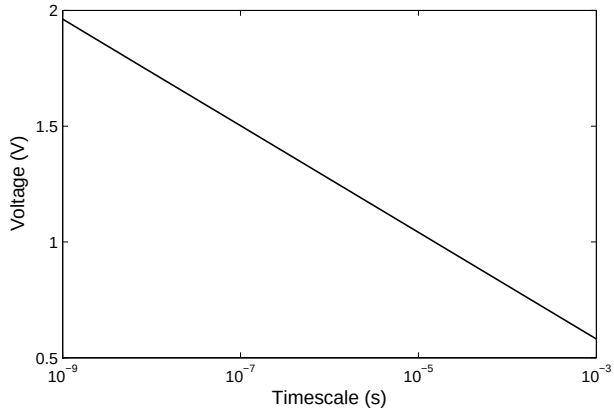

Figure 4.3: The ϵ-threshold voltage for $\epsilon = 0.01$ with different time scales.

for time scales $T \in [1e-9s, 1e-3s]$ *is plotted in Figure 4.3. The semilog graph of the threshold voltage is a straight line, as can be deduced from the fact that* (4.6) *can be accurately approximated as*

$$v_\epsilon = \frac{1}{\eta}\left(\ln\left(\frac{\epsilon}{\lambda T}\right) + \ln 2\right), \tag{4.7}$$

when $\epsilon/(\lambda T) \gg 1$.

The term threshold implies that there is an abrupt change in the programmability of the memristor at a given voltage. Analyzing the change of the state variable at voltage $v_\epsilon + \Delta v$, where v_ϵ is an ϵ-threshold at time scale T yields

$$\frac{\mathrm{d}w(v_\epsilon + \Delta v)}{\mathrm{d}t} T = \lambda \sinh(\eta(v_\epsilon + \Delta v))T.$$

If $\eta(v_\epsilon + \Delta v)$ is larger than 1 then the following approximation holds:

$$\frac{\mathrm{d}w(v_\epsilon + \Delta v)}{\mathrm{d}t} T \approx \lambda \exp(\eta(v_\epsilon + \Delta v))T/2 \tag{4.8}$$

$$= \epsilon \cdot \exp(\eta \cdot \Delta v). \tag{4.9}$$

Thus the memristor programming rate is increased by a factor of $\exp(\eta \cdot \Delta v)$ if the applied voltage across it is $v_\epsilon + \Delta v$ instead of v_ϵ. This shows that the constant η defines the sharpness of the threshold voltage.

Example 23. *With the chosen parameters,* $\exp(\eta \cdot \Delta v) = 10$ *for* $\Delta v = 0.23\,V$. *Moreover, this change in the programming rate is independent of the time scale* T. *In other words the threshold is equally sharp at every* ϵ *and* T. *In Figure 4.4 the change of the state variable* w *is plotted for two different time scales* $T =$ *1e-6s and* $T =$ *1e-9s. The steepness of the curves are identical regardless of the different time scales as was to be expected.*

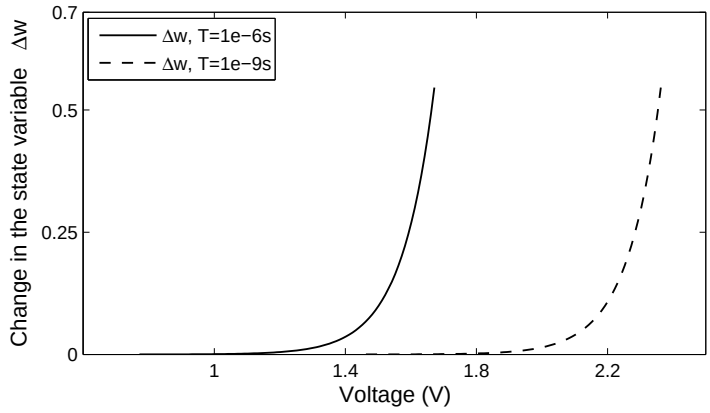

Figure 4.4: The change in state variable w for two different time scales, $T = 1\text{e-}6\text{s}$ and $T = 1\text{e-}9\text{s}$ as the function of the voltage across the device.

Effect of parameter variance on threshold voltage

In practice, the generic memristor model's parameters α, β, λ, and η may vary from their nominal values due to fabrication non-idealities. Variations $\Delta\lambda$ and $\Delta\eta$ in λ and η, respectively, cause an error Δv_ϵ in the threshold voltage v_ϵ, which can be upper bounded as follows [66]:

$$\frac{\Delta v_\epsilon}{v_\epsilon} \leq \frac{1}{v_\epsilon}\left(\left|\Delta\lambda\frac{\partial v_\epsilon}{\partial\lambda}\right| + \left|\Delta\eta\frac{\partial v_\epsilon}{\partial\eta}\right|\right) \tag{4.10}$$

$$\approx \left|\frac{1}{v_\epsilon\eta}\cdot\frac{\Delta\lambda}{\lambda}\right| + \left|\frac{\Delta\eta}{\eta}\right|, \tag{4.11}$$

given that $\epsilon/(\lambda T) \gg 1$.

This error has implications to programming methods sensitive to the exact magnitude of the threshold voltage, as is for example the case with the self-terminating programming described in Section 4.2.2.

4.2 Programming methods

In the remaining part of this chapter different programming methods for memristors are investigated. First, in Section 4.2.1 methods for programming a memristor relative to its current state are studied. Section 4.2.2 concentrates on absolute programming methods, in which a memristor is programmed to an explicitly given state.

4.2.1 Relative methods

Programming with a square pulse

Let a memristor be in state $w \in [0, 1]$, let $\epsilon > 0$, and let v_ϵ be the ϵ-threshold voltage of this memristor for time scale T. A square voltage pulse across the memristor with amplitude v_ϵ and duration T changes by definition the state of the memristor from w to $w + \epsilon$, if this value is within the interval $[0, 1]$. Similarly, a square voltage pulse with amplitude $-v_\epsilon$ decreases the state variable from w to $w - \epsilon$.

Since the rate of change of the state variable is constant for a fixed voltage, the amount of programming is controlled by the duration of the voltage pulse: A pulse with amplitude v_ϵ and duration T' changes the state variable by an amount of $\epsilon(T'/T)$. On the other hand, a train of n pulses with amplitude v_ϵ and duration T increases the state variable from w to $w + n\epsilon$.

Square voltage pulses are used in the cyclical programming method described in Section 4.2.2.

Spike-timing-dependent plasticity

More complex programming behavior is obtained by assuming other shapes than just a rectangular one for the programming pulse. Here a programming scheme called *spike-timing-dependent plasticity (STDP)* [11] is investigated. STDP is a learning mechanism that is postulated to exist in some synapses of mammalian brains. It causes the synaptic weight to change as a function of the relative spike times of pre-synaptic and post-synaptic neurons: If a pre-synaptic spike precedes the post-synaptic one, the synaptic weight is increased. Vice versa, if the post-synaptic spike precedes the pre-synaptic spike, the synaptic weight is decreased. The amount of change of the synaptic weight depends exponentially on the time difference between the spikes. The closer the spikes occur in time, the larger the change in the synaptic weight.

Originally STDP for memristive devices was proposed in [98]. There the STDP mechanism was implemented using time slots, and therefore a global clock signal for the neurons was required. Generalizations of this idea for asynchronous computation were investigated in [3, 72, 83]. In the following the approach proposed in [83] is presented.

Let a memristor be located between nodes n_1 and n_2. When at rest, both nodes are at ground voltage, $v(n_1) = v(n_2) = 0$. In spiking phase, the voltage at a node changes according to the pulse depicted in Figure 4.5 a). The amplitude of the spike is chosen to be smaller than a given ϵ-threshold voltage of the memristor related to the time scale of the pulse. Thus, when a node in one end of the memristor fires a single spike and the node in the

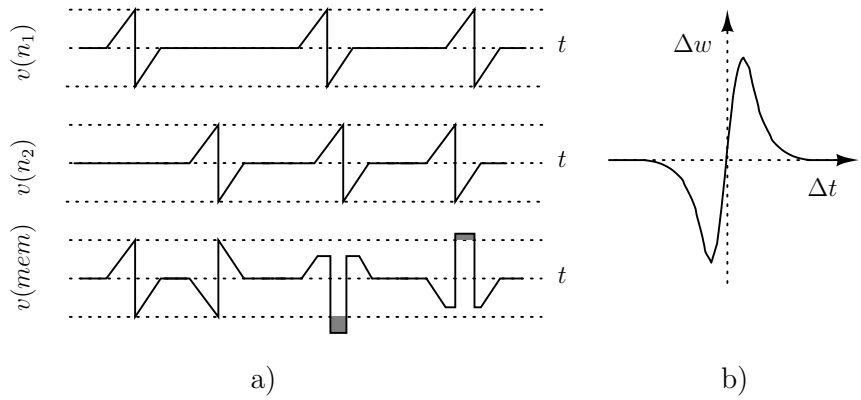

Figure 4.5: Spike-timing-dependent plasticity based programming of a memristor. a) Spike shapes that realize STDP. Dashed lines correspond to the ϵ-threshold voltages. b) Change of memristor state as a function of spike-timing.

other end is grounded, the state variable of the memristor changes less than ϵ. However, when both nodes spike almost simultaneously, the magnitude of the voltage across the memristor exceeds the ϵ-threshold, and the state variable of the memristor is changed by Δw, whose value depends on the relative timing of the spikes, as depicted in Figure 4.5 a) and b). Since the rate of change of the state variable depends exponentially on the voltage, the resulting programming curve decays exponentially with increasing time difference between the spikes.

4.2.2 Absolute methods

Self-terminating programming

Self-terminating programming means a programming method where the state of the memristor relaxes within a predetermined time interval to a targeted value. The programming is not monitored as the dynamics of the circuit ensure that the programming of the state variable is terminated once the target value is reached. An advantage in using self-terminating programming is the simplicity of the required circuitry. As a downside the accuracy of the programming is generally inferior to the other absolute programming method presented in this section. Experimental results concerning self-terminating programming are reported in [35].

A memristive current memory based on self-terminating programming is depicted in Figure 4.6. Using the simple circuit depicted in Figure 4.6 a) it is possible to program a specific current into the memristor, assuming that it is originally in a low-conductance state $w \approx 0$. A memristor and a p-type

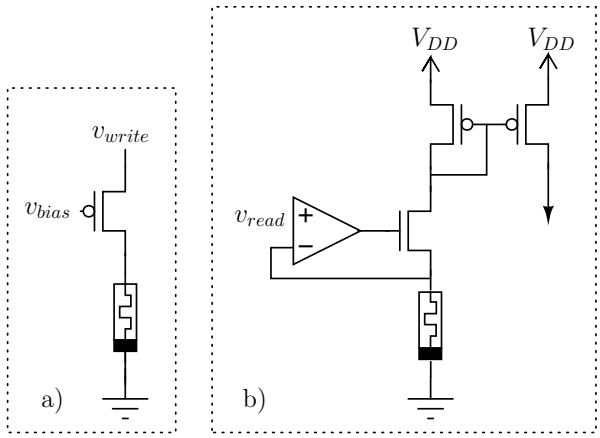

Figure 4.6: a) Write and b) read configuration of a memristive self-terminating current memory.

transistor are placed in series. The PMOS–transistor is acting as a current source that is biased with v_{bias} to carry a current i_{write} in saturation.

Forcing a constant current through the memristor makes the voltage across it at first larger than the ϵ-threshold voltage v_ϵ. Thus the memristor programs into a more conductive state. As the voltage over the memristor approaches v_ϵ, the rate of change of the state variable decreases. As a result, the state w of the device converges approximately to a value which satisfies

$$i_{write} = w\alpha \sinh(\beta v_\epsilon). \tag{4.12}$$

In the appropriate time scale, the programming effectively terminates as the voltage across the memristor becomes sufficiently close to the threshold voltage.

A nondestructive read operation can be carried out at the ϵ-threshold voltage, provided that the time of the read operation is much shorter than the write operation. In Figure 4.6 b) the read phase is illustrated. In this circuit the operational amplifier drives the NMOS–transistor so as to keep its source voltage at $v_{read} = v_\epsilon$. The read-out current is directed through the current-mirror.

In Figure 4.7 simulation results for the self-terminating memristive current memory's write and read operations are depicted.

Remark 24. *For the self-terminating memristive current memory it is advantageous that the I-V behavior of the memristor is close to linear,* i.e.*, the ratio α/β is large, and that the programming behavior is exponential,* i.e.*, the ratio λ/η is small. The latter requirement follows since the threshold voltage should be made as sharp as possible in order to have the programming continue with high rate until the target value is reached.*

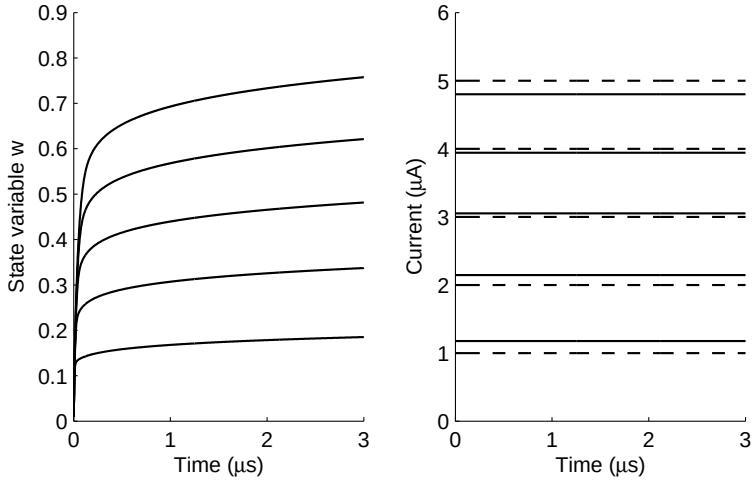

Figure 4.7: Using the memristor as a current memory. Left subfigure: Change of the state variable with 0.01-threshold voltage at timescale 1μs in the write phase. The generic memristor model is used with parameters $\alpha = 5 \times 10^4$, $\beta = 0.01$, $\lambda = 0.06$, and $\eta = 10$. The different curves correspond to the target currents 1μA, ..., 5μA shown in the right subfigure. Right subfigure: A comparison of the stored currents (solid lines) versus target currents (dashed curves) in the read phase.

To see why a linear I-V behavior is desired, recall that the programming of the memristor begins at a low-conductance state $w < 0.1$. When the PMOS–transistor is in saturation, voltage v across the memristor satisfies

$$i_{write}/w = \alpha \sinh(\beta v). \tag{4.13}$$

During the programming the state variable w increases and thus the left hand side of (4.13) *decreases. If the I-V behavior of the memristor were exponential, the parameter β would be large, and hence v would become close to v_ϵ for values of w relatively far from the targeted value thus slowing down the programming. On the other hand, with a linear I-V behavior the voltage v across the memristor stays further above v_ϵ as the state variable w changes towards the targeted value, and thus also the programming rate stays higher.*

In this thesis, self-terminating programming is applied in Chapters 5 and 6, which discuss memristive arithmetic operations and logic computing, respectively.

Continuous monitoring

In *continuous monitoring* the current through a memristor, when a programming voltage V_{prog} is set across it, is continuously compared to a desired value. As long as the current is below (above) the desired value, the memristor is allowed to program towards a more (less) conductive state. In Figure 4.8 a circuit realizing this programming method is depicted. An operational amplifier is used to set a virtual ground to one electrode of the memristor in order to convert the current through the memristor to the voltage across the resistor R_{ref}.

Suppose that the memristor is initially at a low-conductance state $w \approx 0$, the SR-latch is set to state $Q = 1$, and that the switch s_1 is closed. The output of the inverting operational amplifier is connected to the input of the comparator. The memristor is programmed towards a more conductive state until the current through it equals $I = V_{\text{ref}}/R_{\text{ref}}$. At that point, the state of the comparator is flipped to logical 1. This resets the SR-latch and opens the switch s_1 thus ending the programming. A simulation of the continuous monitoring method is depicted in Figure 4.9. A problem with this programming method is the complicated circuitry it needs, including an operational amplifier and an SR-latch. Moreover, the desired current is measured at a programming voltage, and thus if the $I - V$ behaviour of the memristor is very nonlinear, a look-up table may be required to associate targeted read current values to the programming currents. On the other hand, this method should be more reliable than the self-terminating method, since the state of the memristor is being continuously monitored during the programming. As continuous monitoring does not require clocking, it should

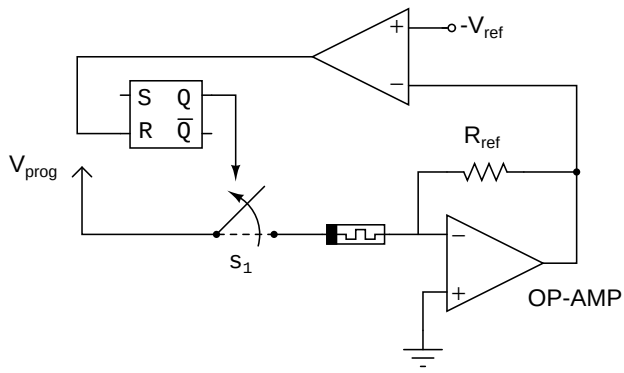

Figure 4.8: Programming the memristor with continuous monitoring. The comparator at the top of the circuit outputs logical 1 when the current through the memristor reaches the desired value $I = V_{\mathrm{ref}}/R_{\mathrm{ref}}$. This switches the state of the SR-latch and opens the switch s_1 which connects the programming voltage V_{prog} to one electrode of the memristor, thus ending the programming.

also be faster than the cyclical programming method discussed next. For these reasons it is suggested to be used in the memristive CNN application presented in Subsection 8.6.1.

Cyclical programming

To conclude this chapter, the cyclical programming method previously proposed for programming floating-gate transistors [7] is investigated. For memristors, this programming scheme was originally proposed in our publication [53]. In [118], a similar programming method was proposed and simulated with a model of the TiO_2-memristor, while [4] presents experimental data of a feedback-based programming method that closely resembles cyclical programming, which achieves 7 bit precision on a TiO_2-based memristive device.

The idea in cyclical programming is to divide the programming process into two phases: the monitoring phase and the programming phase. In the monitoring phase the state of a memristor is measured in a nondestructive fashion, for example using a low read voltage. Depending on the result obtained from the monitoring, the state of the memristor is then programmed to the correct direction by a small constant amount in the programming phase. After programming the system returns to the read phase, and the two phases are alternated until the state of the memristor converges near to its targeted value. In contrast to self-terminating programming, cyclical programming can begin at any initial state of a memristor. Cyclical

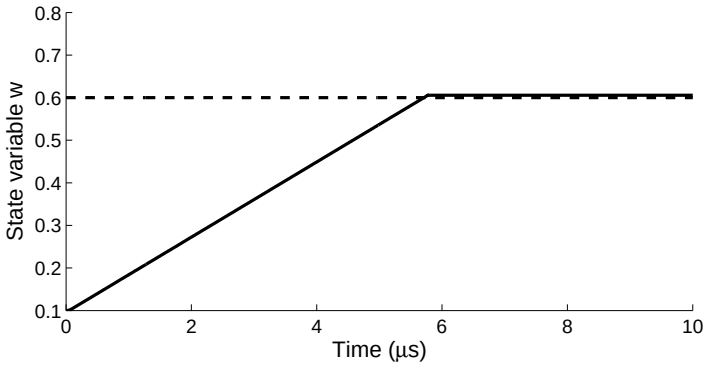

Figure 4.9: Simulation of the continuous monitoring method. The generic analog memristor model with parameters $\alpha = 5e-4$, $\beta = 0.01$, $\lambda = 0.06$, $\eta = 10$, yielding a threshold voltage 1.27 V at 1 μs timescale was used. The reference voltage and resistor were $V_{\text{ref}} = 4.5$ V and $R_{\text{ref}} = 330$ kΩ, respectively, corresponding to a current $I = V_{\text{ref}}/R_{\text{ref}} \approx 13.6$ μA through the memristor, or the value $w = 0.6$ of state variable at the programming voltage $V_{\text{prog}} = 1.5$ V.

programming can be made as precise as needed or physically possible by adjusting the lengths and amplitudes of the programming pulses, as is shown in [4]. The downside of this programming method is the additional circuitry needed in comparison with self-terminating programming. It should also be noted that cyclical programming requires clocking unlike the continuous monitoring method.

In the left subfigure of Figure 4.10 the circuitry needed for the monitoring and programming phases is depicted. In the monitoring phase, the switch $s1$ is open and the switch $s2$ is closed, and thus the voltage IN is compared to ground value and the result is stored into the capacitor. In practice the current source i_m is realized for example with an NMOS–transistor. In the programming phase the switch $s1$ is closed and the switch s_2 is opened, and the memristor is programmed either towards a more conductive or a less conductive state, depending on the voltage v_c sampled into the capacitor. Switching cyclically between the two operation phases, the state of the memristor eventually converges to satisfy

$$i(V_m) = i_m, \tag{4.14}$$

where $i(V_m)$ is the current through the memristor at voltage V_m. A simulation of the cyclically programmed current memory is depicted in the right subfigure of Figure 4.10. Here a memristor is programmed cyclically to store currents from 1μA to 4μA.

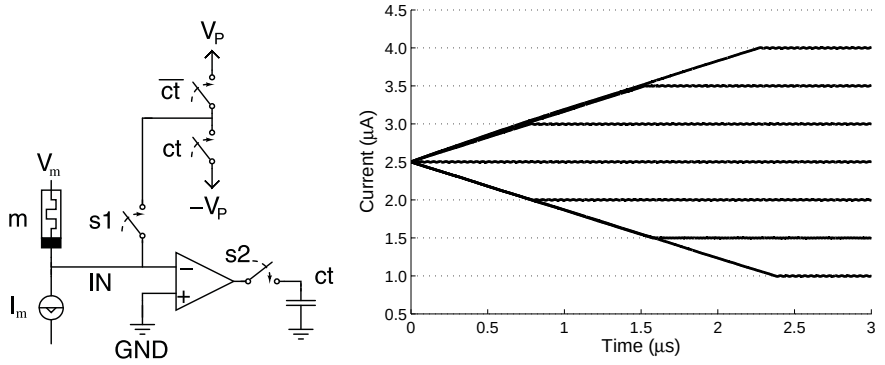

Figure 4.10: Cyclical programming of a memristor. Left subfigure: The current memory configuration. Right subfigure: The current through the memristor corresponding to input voltage of 1V during the monitoring phase of the cyclical programming. Initially, the value of the state variable is $w = 0.5$ corresponding to a current of 2.5μA. The different curves correspond to the different currents (1μA, 1.5μA, ..., 4μA) to be stored. The generic memristor model with parameters $\alpha = 5 \times 10^{-4}$, $\beta = 0.01$, $\lambda = 0.06$, and $\eta = 10$ is used. The sampling rate of the switch s_2 is 100 Msps.

Chapter 5

Analog Arithmetic Operations

In this chapter the absolute programming methods discussed in Section 4.2.2 are used to perform analog arithmetic operations with memristors. Three basic arithmetic operations are discussed: copying, addition, and multiplication.

Copying and addition do not assume any specific memristor model, and these operations can be modified to operate also on signed numbers, as is described in Section 5.2. The variable to be copied or added is the current through a memristor, when a fixed voltage is set across it. Hence, the memristors acting in these operations do not need to have the same parameters or threshold voltages. If they do have same parameters, then these operations act on the state variables of the memristors.

A memristive multiplication operation is described in Section 5.3. It requires a memristor whose I-V behaviour is exponential or sinh-type, and it operates only on positive numbers which are represented by the state variable of a memristor. Since it operates on state variables, it is less robust against parameter variations and device mismatch than the other memristive arithmetic operations described in this chapter.

The presented arithmetic operations are further used in this thesis in the memristive applications of Chapters 8 and 9. The main references for this chapter are our publications [53] and [55].

5.1 Copying the state of a memristor

In the following, two different methods for copying the current of a memristor are described. The first of the proposed methods uses self-terminating programming, while the second uses cyclical programming, as described in Section 4.2.2.

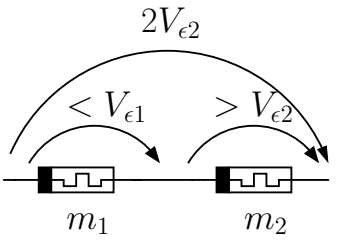

Figure 5.1: Copying the current from one memristor to another using self-terminating programming.

5.1.1 Self-terminating copying of the current of a memristor

Let V_{ϵ_i} be the threshold voltage for some small ϵ and some timescale T of the memristor m_i, where $i = 1, 2$. For the self-terminating copy operation, assume that the threshold voltages satisfy $V_{\epsilon 1} \geq V_{\epsilon 2}$. Now, copying the current through memristor m_1 at voltage $V_{\epsilon 2}$ is achieved by setting the voltage $2V_{\epsilon 2}$ over the memristors connected in series as shown in Figure 5.1. Assuming that the memristor m_2 is initially in a low-conductance state, the voltage across it lies within the interval $(V_{\epsilon 2}, 2V_{\epsilon 2}]$. While this is true, it follows that the voltage across the memristor m_1 satisfies

$$V(m_1) < V_{\epsilon 2} \leq V_{\epsilon 1}. \tag{5.1}$$

Ideally, the self-terminating programming continues until the voltage across m_2 reaches $V_{\epsilon 2}$, when the duration of the programming is comparable to T. By (5.1) the state of m_1 stays approximately constant during the programming. The currents through the memristors thus converge to satisfy

$$I_{m_2}(V_{\epsilon 2}) = I_{m_1}(V_{\epsilon 2}), \tag{5.2}$$

where $I_m(V)$ stands for the current through the memristor m at a voltage V. In other words, the current through m_1 at voltage $V_{\epsilon 2}$ is programmed to be the current of m_2 at the same voltage. Should the parameters of the memristors m_1 and m_2 be identical, this means that in the end of the programming $w_2 = w_1$, that is, the state of m_1 is copied as the state of m_2.

There are some practical problems in self-terminating copying. As the state of m_2 approaches the desired value, the voltage across the device decreases and so does the programming rate. For example, as shown in the left subfigure of Figure 5.2, copying state values close to $w = 0.9$ is not accurate. Also, the closer the voltage across m_1 is to $V_{\epsilon 1}$, the faster the undesired change of the state of m_1. Regardless of the model parameters, copying small state values results in more accurate results as shown in the right subfigure of Figure 5.2. The right subfigure of Figure 5.2 demonstrates that a more nonlinear model with ($\lambda =$ 1e-9, $\eta = 50$) yields fairly accurate

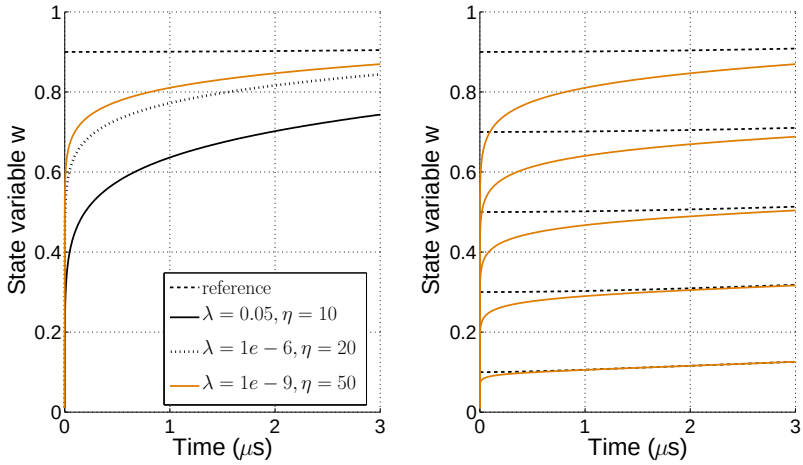

Figure 5.2: Copying the current from a memristor to another. Left subfigure: Copying the current corresponding to the value of state variable $w = 0.9$ with three different memristor models, $\eta = 10, 20, 50$. Right subfigure: Copying different currents with a memristor model with $\lambda =$ 1e-9, $\eta = 50$.

results also with values of w close to 1. It should be noted that the value $\eta = 50$ yields an extremely exponential, and possibly unrealistic, model with respect to programming voltage.

5.1.2 Cyclical copying of the current of a memristor

A more accurate although also more complex copy operation is obtained by using the cyclical programming method. A cyclical copying circuit is depicted in Figure 5.3. It is assumed that the read voltage V_R satisfies $2V_R < V_{\epsilon 1,2}$ in order to keep the states of the memristors unchanged during the read phase.

The cyclical copying circuit operates similarly to the one depicted in Figure 4.10 of Chapter 4. In the read phase, $V_1 = V_R$, $V_2 = -V_R$, and the voltage IN is compared to the ground potential at the comparator. Then V_1 is driven to high impedance in order to keep the state of m_1 unchanged during the programming phase. During the programming phase, the voltage V_2 is set to ground, and depending on the polarity of the node ct — which corresponds to the polarity of the node IN during the read phase — either V_P or $-V_P$ is connected to IN by the switch $s1$, where V_P is a programming voltage suitable for the memristor m_2. This changes the state of m_2 by a small amount either to the more conductive or the less conductive direction, and hence the circuit gradually converges to satisfy $I_{m_2}(V_R) = I_{m_1}(V_R)$, where $I_{m_i}(V_R)$ is the read current through the memristor m_i. Again, in the

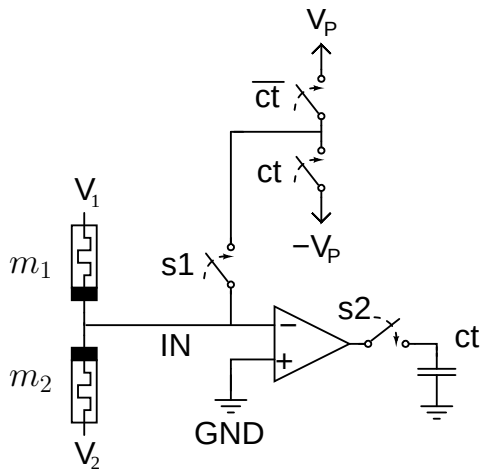

Figure 5.3: A circuit for the cyclical current copy operation.

case of identical memristor parameters, this means that the state variable w_2 is programmed to equal w_1.

5.2 Addition of currents of memristors

Next, a memristive circuit performing addition is presented. Due to the inherent inaccuracy of the self-terminating programming, only cyclical programming is considered in the more demanding arithmetic operations discussed in the rest of this chapter. In Figure 5.4 a memristive arithmetic unit capable of setting a constant current as the read current of a memristor, copying a memristor's read current to another memristor, and summing the read currents of two memristors and storing the result to a third one, is depicted. All of these arithmetic operations are realized by monitoring the voltage of the node IN and programming the memristors according to its polarity.

In the cyclical addition the goal is to change the state of m_1 until $I_{m_1}(V_R) = I_{m_2}(V_R) + I_{m_3}(V_R)$. The circuit of Figure 5.4 can be configured to read and programming phases with switches s_1 and s_2. In the read phase the driver voltages are $V_1 = V_R$ and $V_2 = V_3 = -V_R$ for a small non-destructive read voltage V_R. The operational amplifier is used to compare the node IN to the ground voltage. If the read conductance of m_1 is larger (smaller) than the sum of the read conductances of m_2 and m_3, the operational amplifier produces a LO (HI) comparison voltage, which is then sampled with $s2$ to the capacitor. In the programming phase the capacitor voltage ct controls whether the programming increases or decreases the conductance of m_1. More precisely, the voltage V_1 is set to ground, and

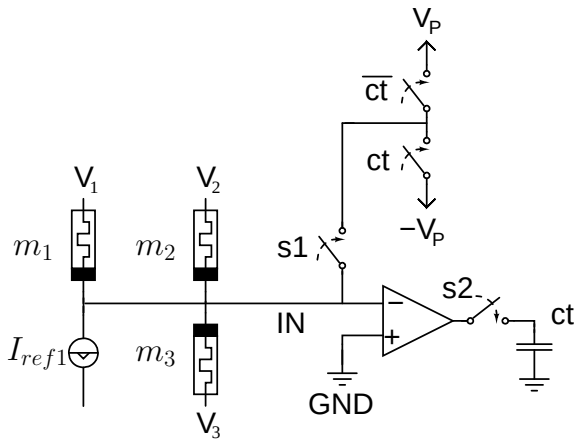

Figure 5.4: A basic arithmetic circuit for cyclical constant current programming, copying and (signed) addition.

depending on the polarity of ct, either V_P or $-V_P$ is set to IN by the switch $s1$. During the programming phase the drivers V_2 and V_3 are set to high impedance in order to prevent unwanted programming of the memristors m_2 and m_3.

The cyclical programming forces the voltage at node IN during the read phase to approach zero. The amount of programming during one cycle (step size) can be reduced by shortening the duration of a programming cycle, or lowering V_P, as was experimentally shown in [4]. If the step size is small enough, a sequence of read and programming phase cycles assures that the read currents satisfy

$$I_{m_1}(V_R) \approx I_{m_2}(V_R) + I_{m_3}(V_R). \tag{5.3}$$

In other words, the sum of the read currents of the two memristors m_2 and m_3 is programmed as the read current of the memristor m_1. In Figure 5.5, a simulation of this addition operation is depicted with identical memristors thus resulting in $w_1 = w_2 + w_3$.

5.2.1 Representing signed numbers

The memristive addition described above operates only on positive quantities $I_{m_i}(V_R)$. In order to perform signed addition, the circuit of Figure 5.4 can be complemented with a current source, as is described in the following.

Let $w_{\min}$ and $w_{\max}$ be the minimum and maximum value of the state variables of the input memristors of the addition, and let $I_{\min}$ and $I_{\max}$ be the corresponding read currents. Let us denote by $I_0 = (I_{\min} + I_{\max})/2$ the read current corresponding to the value zero: read currents which are

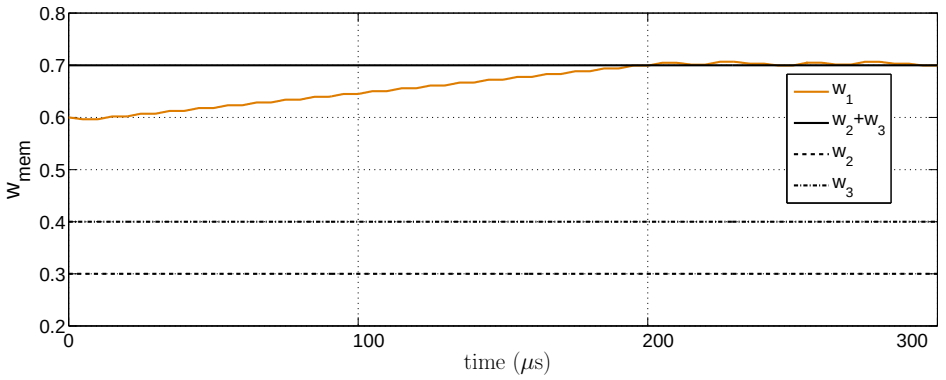

Figure 5.5: SPICE simulation of summing two memristor currents (states) into a third memristor. Memristor characteristics were simulated using the SPICE model presented in Chapter 3.

smaller than I_0 are regarded as negative values, and read currents which are larger than I_0 are correspondingly regarded as positive values. As a result, a read current I_x is associated with a numerical value x by the formula

$$x = (I_x - I_0)/I_s, \tag{5.4}$$

where I_s is a scaling constant corresponding to a unit current.

Suppose that the read currents of the memristors m_1 and m_2 are I_x and I_y, respectively, and that their signed sum I_{x+y} is to be stored to the memristor m_3. From (5.4) it follows that

$$x + y = (I_x + I_y - 2I_0)/I_s \implies I_{x+y} = I_x + I_y - I_0. \tag{5.5}$$

In other words, setting the current source I_{ref1} in the circuit depicted in Figure 5.4 to output the negative current $-I_0$ allows signed addition of two inputs. In practice, this current source can be implementd by a pair of CMOS-transistors to allow both positive and negative currents. Signed addition of n inputs requires the additional current of $-(n-1)I_0$.

If the read current is a linear function of the state variable of the memristor as is the case with the generic analog memristor model of Section 3.4, and if the parameters of the memristors are identical, then this operation corresponds to the signed addition of the state variables of the memristors. In Figure 5.6 a simulation of such a case is plotted. Initially the state variables satisfy $w_1 = 0.1$, $w_4 = 0.6$, and $w_5 = 0.2$. Since $w = 0.5$ corresponds to the zero value, the state variables w_4 and w_5 represent numbers 0.1 and -0.3, when the scaling constant I_s equals to the read current at $w = 1$. After the cyclical programming, $w_1 = 0.3$, yielding thus the result $-0.2 = 0.1 + (-0.3)$.

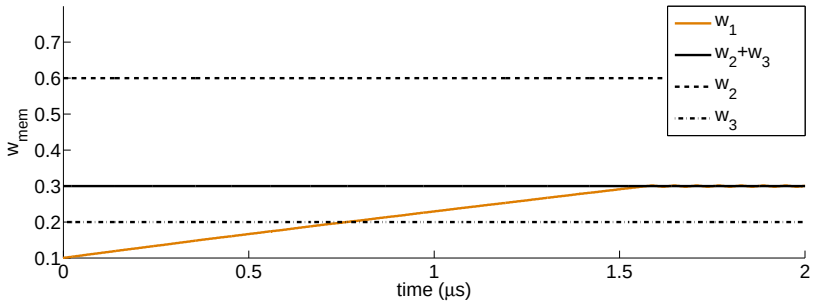

Figure 5.6: Simulation of the cyclical signed addition. The generic analog memristor model was used with parameters $\alpha = 5e-4$, $\beta = 0.01$, $\lambda = 0.06$ and $\eta = 10$. Here $I_0 \approx 1.25$ μA.

5.3 Multiplication of states of memristors

There are various options to carry out multiplication and division in analog CMOS circuits [27]. One method of interest in the context of this section is the translinear principle. Translinear circuits operate in the logarithmic domain where multiplication and division can be carried out by addition and subtraction, respectively. For example, in a diode-connected CMOS transistor biased to the subthreshold region, the input current is converted to a logarithmic voltage. With memristors it is the state variable that needs to be converted to a logarithmic voltage.

Current monitoring provides a way to perform the translinear principle with the generic analog memristor model. The operation is based on the observation that the analog memristor model has-sinh type I-V relationship which can be approximated by an exponential function with sufficiently large voltages. Consider the circuit shown in Figure 5.7. Let the current I_{ref2} through the analog memristors m_2 and m_3 be chosen to be sufficiently large so that their $I-V$ relationship can be approximated as

$$I_{ref2} = w_i \alpha \sinh(\beta V_i) \approx w_i \alpha \exp(\beta V_i)/2 \quad i = 2, 3. \tag{5.6}$$

Solving the voltages yields

$$V_i \approx \frac{1}{\beta} \ln\left(\frac{2I_{ref2}}{\alpha w_i}\right) \quad i = 2, 3, \tag{5.7}$$

and thus,

$$V_{\text{ref}} = V_2 + V_3 = \frac{1}{\beta} \ln\left(\frac{4I_{ref2}^2}{\alpha^2 w_2 w_3}\right). \tag{5.8}$$

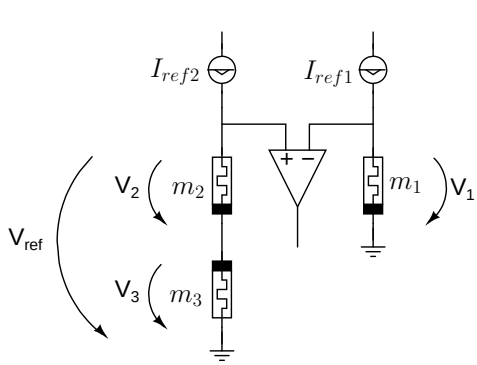

Figure 5.7: A circuit configuration used in cyclical multiplication of state variables of memristors m_2 and m_3.

Setting $I_{\text{ref1}} = 2I_{ref2}^2/\alpha$ yields

$$V_1 = \frac{1}{\beta} \ln\left(\frac{2I_{\text{ref1}}}{\alpha w_1}\right) = \frac{1}{\beta} \ln\left(\frac{4I_{ref2}^2}{\alpha^2 w_1}\right). \tag{5.9}$$

Now if m_1 is cyclically programmed until $V_1 = V_{\text{ref}}$, it follows by (5.8) and (5.9) that $w_1 = w_2 w_3$. Division of two state variables is obtained by programming w_3 instead of w_1. This multiplication scheme works as long as the memristors have sinh-type I-V behavior, mismatch between devices is small, and the result of the multiplication (or division) lies within the interval $[w_{\min}, w_{\max}]$.

The current I_{ref2} should be large enough to enable accurate approximation of the exponential function as described above, while the monitoring time should be short enough to avoid unwanted programming of m_1 and m_2. Figure 5.8 shows a simulation of the multiplication. Memristor m_2 is at state 0.5, while the state of memristor m_3 is at 0.7. After the cyclical programming, w_1 converges to approximately $w_2 w_3 = 0.35$.

5.4 Memristive arithmetic unit

A memristive arithmetic unit featuring cyclical constant current programming, current copying, signed addition of currents, and multiplication of state variables is depicted in Figure 5.9. When s_3 is closed, the circuit is configured as the signed addition circuit of Figure 5.4. When s_3 opened, the circuit is configured as the multiplication circuit of Figure 5.7. Additional switches enabling subtraction and division are left out of this circuit for simplicity. Moreover, for convenience it is assumed that the current source I_{ref1} can output both positive and negative currents. It should be noted that

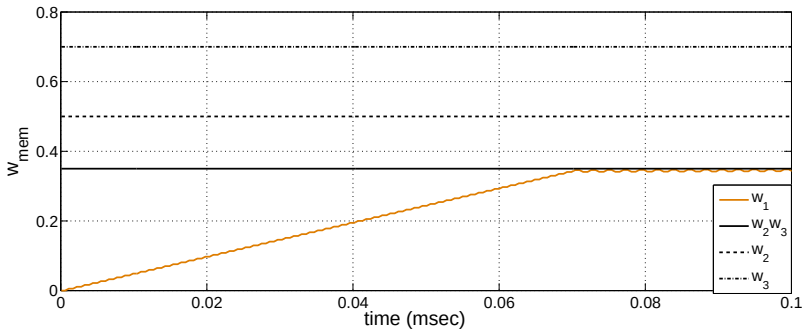

Figure 5.8: Simulated multiplication of analog memristor state variables.

more memristors can be added to this circuit — for example the subcircuit containing memristors m_2 and m_3 can be duplicated.

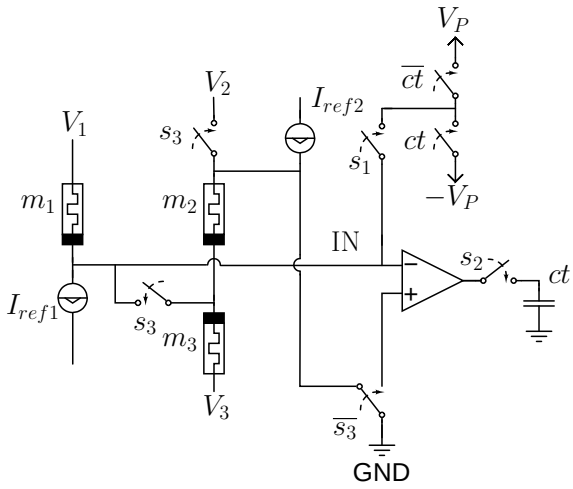

Figure 5.9: A memristive arithmetic unit. The switch s_3 determines whether the circuit operates in additive or multiplicative configuration, while the switches s_1 and s_2 are used to control the cyclical programming.

5.5 Remarks on implementation

The feasibility of memristive arithmetic depends on the programming behaviour of the memristor, and on the resolution of its state variable. For example, it is beneficial that the memristors have a steep programming threshold, and it is important to be able to change the memristor's state by small steps by using suitable voltage pulses. So far in this thesis the state has been assumed to take continuous values within $[w_{\min}, w_{\max}]$, an

assumption which may not hold in reality, where the state corresponds for example to a distance of ions in crystalline semiconductor bulk as discussed in Chapter 3.

The analog memristor presented in [13] seems to be a suitable candidate for implementing arithmetic operations. Indeed, its state variable can be changed by small steps in a seemingly continuous range. Moreover, the model proposed for this tungsten oxide -based memristor closely resembles the generic analog memristor model of Section 3.4, thus making the multiplication operation at least in theory possible. However, no arithmetic experiments using this memristor have been yet conducted.

As a conclusion, the following remarks on memristive arithmetic operations can be made. If analog memristors with sufficiently high resolution and low device variability can be fabricated, they could be used for analog addition and subtraction, where the variables in these arithmetic operations are presented as currents through the memristors. Moreover, if the I-V characteristic of the memristor has sinh-type relationship, then approximate multiplication could be obtained by using the translinear method proposed in Section 5.3.

Chapter 6

Memristive Implication Logic

Implication logic was proposed as a natural form of logic for memristors by Phil Kuekes in a talk presented at the first Memristor and Memristive Systems Symposium at University of California, Berkeley in 2008 [50]. The message of this presentation was that a binary memristor is capable of not only storing information but also of performing logic on it: a memristor can act both as a latch and a gate. However, since memristors are two-terminal devices, they cannot be used as typical Boolean logic gates such as the OR and the AND gates, and thus a different approach for logic computing must be taken. As it turns out, the logic operation called *material implication*, which was studied by Whitehead and Russell in early 1900s in *Principia Mathematica* [112], can be straightforwardly implemented on memristors with self-terminating programming. Combined with the logical constant false — with memristors, this means unconditionally resetting a memristor to the off-state — implication logic is functionally complete, which means that all Boolean functions can be synthesized using it.

This chapter is organized as follows. In Section 6.1, self-terminating memristive logic operations are described, and their digital abstractions are presented. In the next four sections, different synthesis methods for memristive implication logic are described. Section 6.2 presents a synthesis method based on the 2–depth NAND–form of a Boolean function. This method requires the most computational steps of the synthesis methods discussed in this chapter, but on the other hand, uses only the elementary single-input implication operation. Section 6.3 presents a single-input synthesis method which uses the minimum number of memristors. A variation of this method using multi-input implication operation is described in Section 6.4; with multi-input implication logic the number of computational steps required to synthesize a Boolean function can be significantly reduced. Another solution for shortening the computational sequences is to use complementary representation of variables as is discussed in Sections 6.2.1 and 6.4.2. Accordingly,

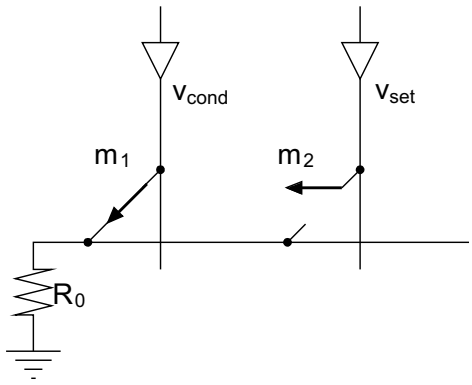

Figure 6.1: Stateful memristor logic. The resulting state of m_2 depends of its previous state and the state of m_1. The state of m_1 remains unchanged during the operation.

the shortest computational sequences are then obtained by applying both the complementary representation and the multi-input implication operation, as is the case with the NAND–OR method described in Section 6.4.2. Section 6.5 summarizes the synthesis methods, and Section 6.6 discusses the limitations and possible improvements of memristive implication logic. This chapter is concluded by Section 6.7, wherein a stateful logic operation requiring rectifying memristors and enabling multi-output operation is described.

The literature on memristive implication logic is rather limited and hence this chapter is based mainly on our original contributions [58, 62–64, 89]. Other references used for this chapter are [12], an empirical article on implication logic with the titanium dioxide-type memristor discussed in Section 3.2, and the more theoretically inclined publications [28, 84].

As a technical note, in the following the logical operators AND and OR are either written as Boolean functions AND$(\cdot)$ and OR$(\cdot)$, or as logical connectives $\wedge$ and $\vee$. A Boolean function is a function $f : B^n \to B$, where n is some positive integer and $B = \{0, 1\}$. Moreover, in this chapter memristors are identified with their states, *e.g.*, $m_1 = 0$ denotes $w_1 = 0$.

6.1 Memristive stateful logic

Kuekes [50] proposed in 2008 that the circuit depicted in Figure 6.1 would enable self-terminating stateful memristor logic. 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 $\sqrt{R_{\mathrm{OFF}}/R_{\mathrm{ON}}} \gg 1$. The series resistance R_0 is chosen as $R_0 = \sqrt{R_{\mathrm{ON}} R_{\mathrm{OFF}}}$, so that $R_0/R_{\mathrm{ON}} = R_{\mathrm{OFF}}/R_0 \gg 1$. Memristor m_1 is driven with a con-

m_1	m_2	$m_2 := m_1 \rightarrow m_2$
0	0	1
0	1	1
1	0	0
1	1	1

Table 6.1: The truth table of material implication operation, resulting from the choice $v_{\text{cond}}, v_{\text{set}} > 0$.

ditional voltage v_{cond} whose magnitude is smaller than the programming threshold voltage of the memristor, $|v_{\text{cond}}| < V^T$. For simplicity it is assumed here that $V^T = V^{T+} = -V^{T-}$, where the notation of Section 3.4 is used. Memristor m_2 is driven with a programming voltage v_{set}, whose value depends on the chosen stateful logic operation and is determined in the following. As a result, a stateful logic operation is performed on m_2. In the following subsections, all possible combinations of polarities for the conditional and programming voltages and the resulting logical operations are considered.

6.1.1 Material implication

First, suppose that $v_{\text{cond}} > 0$ and that $v_{\text{set}} > V^T$ satisfies

$$v_{\text{set}} - v_{\text{cond}} < V^T. \tag{6.1}$$

As before, the binary values 0 and 1 are used to denote the low-conductance and the high-conductance states of a memristor, respectively. Now the voltage divider in circuit of Figure 6.1 ensures that m_2 is programmed to the on-state exactly when $m_1 = 0$, since the voltage across m_2 is not enough to program it if $m_1 = 1$. On the other hand, the voltage across the memristor m_1 is maintaned below V^T at all times, and therefore its state remains unchanged during the operation. The resulting state of $m_2 := \text{OR}(\neg m_1, m_2)$ can be read from the truth table presented in Table 6.1. This truth table corresponds to the logical connective called *material implication* that is denoted by the symbol $\rightarrow$. Material implication and the logical constant false, corresponding to resetting a memristor's state to 0, form a universal set of logical connectives, which means that any Boolean function can be synthesized using them. It is crucial to note that the result of the implication operation is stored to the state of the memristor m_2, that is, to one of the inputs of the operation. The synthesis of Boolean functions for memristive implication logic is further investigated in Sections 6.2, 6.3 and 6.4.

In [12] the circuit of Figure 6.1 was shown to perform memristive implication logic, when the TiO_2–based memristor was used. The switching rate of this device is a highly nonlinear, even super-exponential, function

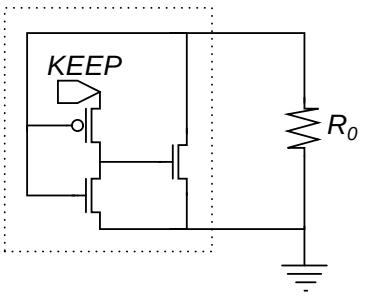

Figure 6.2: The keeper circuit added in parallel with R_0 in the implication circuit of Figure 6.1.

of the input voltage as was noted in Section 3.2. Such nonlinearity is advantageous for the implementation of memristive implication logic, as the threshold voltage of the memristor should be made as sharp as possible in order to minimize undesirable programming of the memristors.

In general however, the simple circuit of Figure 6.1 may not perform the implication operation correctly. If m_1 and m_2 are in the low-conductance state while performing the implication operation $m_2 = m_1 \to m_2$, R_0 should keep the voltage over m_2 large until m_2 has changed state. However, as m_2 becomes more conductive, the current through R_0 increases, and correspondingly the voltage across m_2 decreases. This in turn slows down and possibly even terminates the programming of m_2 before the state $m_2 = 1$ is reached. This problem is dealt with in [54] by adding a keeper subcircuit in parallel with R_0. The keeper circuit depicted in Figure 6.2, activated by a KEEP signal, maintains an initially low voltage over R_0 if $m_1 = 0$.

In Figure 6.3 the implication circuit of Figure 6.1 augmented with the keeper subcircuit is simulated. Once again, the generic memristor model is assumed, this time with parameters $\alpha = 0.001$, $\beta = 0.1$, $\lambda = 0.06$, and $\eta = 10$. Using time scale $T = 1\mu$s, the threshold voltage of this model for $\epsilon = 0.01$ is

$$v_\epsilon = \frac{1}{\eta} \sinh^{-1}\left(\frac{\epsilon}{\lambda T}\right) \approx 1.27\text{V}. \tag{6.2}$$

The exact values of the parameters are not important, but as was noted above, it is advantageous to have as sharp a threshold voltage as possible, and therefore a large value of the parameter η. The control signals are operated as follows. At $t = 0$ s the voltage v_{cond} is set to 1.2 V. Then, at $t = 0.5$ μs the KEEP signal is activated. Finally, at $t = 1$ μs, the voltage v_{set} is set to 1.8 V. The resistor R_0 is chosen to have a resistance value of 10 kΩ.

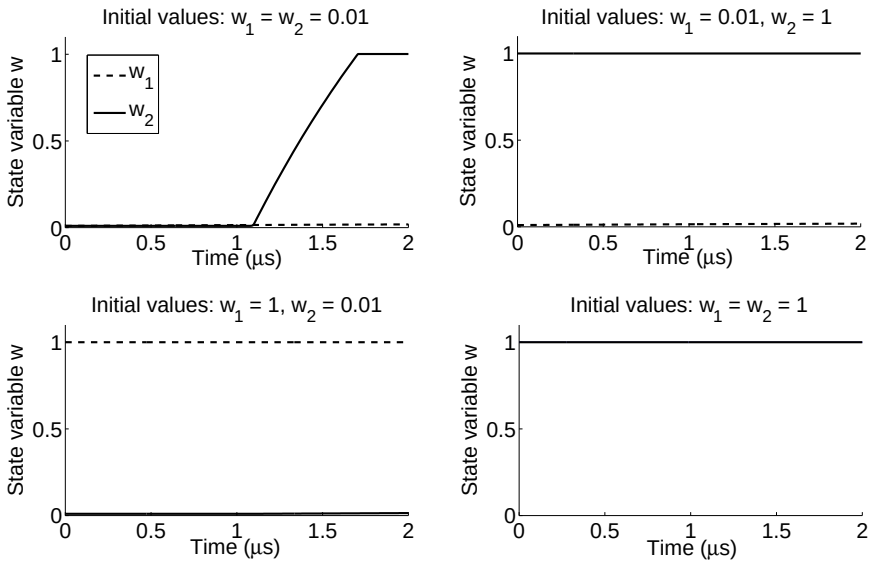

Figure 6.3: Simulation of the implication circuit with four different initial conductance states of the memristors m_1 and m_2. The initial and final states of m_1 and m_2 satisfy the truth table of the material implication.

6.1.2 Other stateful logic operations

Consider then the case where the polarities of the programming and conditional voltages are both negative:

$$\begin{aligned} &-V^T < v_{\text{cond}} < 0, \\ &\quad v_{\text{set}} < -V^T, \quad \text{and} \quad v_{\text{set}} - v_{\text{cond}} > -V^T. \end{aligned}$$

Analogously to the above, one might falsely assume that in this case the memristor m_2 is programmed to the off-state exactly when $m_1 = 0$, resulting in $m_2 := \text{AND}(m_1, m_2)$. To see why this is not the case, suppose that the memristors are in states $m_1 = 0$ and $m_2 = 1$. Then the voltage across R_0 becomes close to v_{set} due to voltage division, and thus the voltage across m_2 becomes too small to change its state to $m_2 = 0$. Consequently, this choice of voltages does not change the state of m_2 at all.

Choosing $-V^T < v_{\text{cond}} < 0$ and $0 < v_{\text{set}} < V^T$ with the constraint

$$v_{\text{set}} - v_{\text{cond}} > V^T \tag{6.3}$$

programs m_2 to the on-state exactly when $m_1 = 1$, thus yielding $m_2 := \text{OR}(m_1, m_2)$. Again a keeper subcircuit may be required for correct operation. It should be noted that the OR operation together with 'set' or 'reset' do not form a universal set of logical connectives. Therefore at least some

other stateful logic operation must be used together with OR to synthesize all Boolean functions.

The fourth possible choice of polarities $0 < v_{\text{cond}} < V^T$ and $-V^T < v_{\text{set}} < 0$ with the constraint

$$v_{\text{cond}} - v_{\text{set}} > V^T \tag{6.4}$$

yields also the OR-function, but this time on m_1. As a result of the operation, $m_1 := \text{OR}(m_1, m_2)$. However, the logical operation changes if rectifying memristors are used. Then this fourth choice of polarities results in a operation called *converse nonimplication*. The use of rectifying memristors for converse nonimplication is discussed in more detail in Section 6.7.

6.1.3 Computational sequence

The NAND function is universal, which means that all Boolean functions can be written in a form containing NANDs of input variables. Thus to show that memristive implication logic is universal, it is sufficient to synthesize the NAND function with it. Consider thus three memristors, m_1, m_2 and m_3, and assume that m_3 is initially in the off-state $m_3 = 0$. Suppose then that two implication operations are performed in succession, first from m_1 to m_3, and then from m_2 to m_3. The resulting state of m_3 can then be written as

$$\begin{aligned} m_3 = m_2 \to (m_1 \to m_3) &= \neg m_2 \vee (\neg m_1 \vee m_3) && (6.5)\\ &= (\neg m_1 \vee \neg m_2) = \neg(m_1 \wedge m_2), && (6.6) \end{aligned}$$

since by definition $p \to q = (\neg p) \vee q$. This sequence of operations can briefly be written as a computational sequence

$$\text{NAND}(m_1, m_2) : m_3 = 0, \quad m_1 \to m_3, \quad m_2 \to m_3, \tag{6.7}$$

where the result is stored as the state of the memristor m_3. Recall, that in memristive stateful logic the result of the logical operation is stored into the second operand of the operation, thus $m_i \to m_j$ stands for $m_j := m_i \to m_j$.

From now on, the memristors are divided into three sets: the set P of *input memristors* p_i, the set A of *auxiliary memristors* a_j, and the set R of *result memristors* r_k. The states of the input memristors represent the values of the input variables of the corresponding Boolean function, and they remain constant at all times. Therefore in a computational sequence, operations of the form $p_i = 0$ and $x \to p_i$ are forbidden, as input memristors can be located only on the left side of the material implication as in $p_i \to a_j$.

The auxiliary memristors act as the memory needed to store intermediate results of the computation, and the value of the Boolean function is

finally stored into the result memristors. If non-complementary representation of variables is used, the set $R = \{r\}$ contains only a single memristor. It is assumed that initially all the auxiliary memristors and the result memristors are set to state 0. Now the computational sequence of NAND can be rewritten as

$$\mathrm{NAND}(p_1, p_2) : p_1 \to r, \quad p_2 \to r. \tag{6.8}$$

Other commonly used Boolean functions can be synthesized as

$$\mathrm{NOT}(p_1) : p_1 \to r \tag{6.9}$$

$$\mathrm{OR}(p_1, p_2) : p_1 \to a, \quad a \to r, \tag{6.10}$$

$$a = 0, \quad p_2 \to a, \quad a \to r \tag{6.11}$$

$$\mathrm{AND}(p_1, p_2) : p_2 \to a, \quad p_1 \to a, \quad a \to r. \tag{6.12}$$

Notice that the OR function can be also obtained from the stateful logic operation for which $v_{\mathrm{cond}} < 0$ and $v_{\mathrm{set}} > 0$. Here its synthesis by material implication is given for completeness, and also because if rectifying memristors are used, the OR operation may not be available. From the above it follows that no auxiliary memristors are needed for the synthesis of NAND and NOT functions, while one auxiliary memristor a is required for the synthesis of OR and AND functions.

6.2 Synthesis with 2–depth NAND form

All Boolean functions can be written in the disjunctive normal form

$$f \equiv \mathrm{OR}(\mathrm{AND}(\ldots), \ldots, \mathrm{AND}(\ldots)), \tag{6.13}$$

which is useful for the synthesis of Boolean functions in conventional CMOS–based logic. For memristive implication logic, a more useful form is the *2–depth NAND form*

$$f \equiv \mathrm{NAND}(n_1, \ldots, n_k), \tag{6.14}$$

where each n_i is a NAND–clause of (possibly inverted) input variables. This 2–depth NAND form is easy to establish by substituting the OR and all ANDs in the disjunctive normal form expression of f by NANDs. Notice, that any NAND–clause n_i can be synthesized by using a single auxiliary memristor. For example, the following computational sequence yields $r = \mathrm{NAND}(p1, \neg p_2)$, when initially $a = r = 0$:

$$p_1 \to r, \quad p_2 \to a, \quad a \to r, \quad a = 0. \tag{6.15}$$

The last step $a = 0$ is performed to allow possible subsequent computational sequences. In the next Theorem the computational sequence (6.15) is generalized for the 2–depth NAND form (6.14), as proposed in [64].

Theorem 25. *Two auxiliary memristors suffice to synthesize any Boolean function* $f : B^n \to B$.

Proof. Let f be written in the 2–depth NAND–form (6.14). Consider the ith NAND–clause

$$n_i = \mathrm{NAND}(p_1^{\alpha_1}, p_2^{\alpha_2}, \ldots, p_n^{\alpha_n}), \tag{6.16}$$

where $\alpha_j \in \{0, 1\}$ for all $j = 1, \ldots, n$ and the notation $p^0 = \neg p$ and $p^1 = p$ is used. For $j = 1, \ldots, n$, perform the computational sequence

- $p_j \to a_1, \quad a_1 \to a_2, \quad a_1 = 0$, if $\alpha_j = 0$
- $p_j \to a_2$, if $\alpha_j = 1$

After this, the value of n_i is stored as the state of the auxiliary memristor a_2. Then perform the computational sequence

$$a_2 \to r, \quad a_2 = 0. \tag{6.17}$$

When the above procedure is iterated for all $i = 1, \ldots, k$, the state of the result memristor r equals

$$r = \mathrm{NAND}(n_1, \ldots, n_k) = f(p_1, \ldots, p_n). \tag{6.18}$$

□

Example 26. *In the examples of this chapter, the synthesis of the three-input parity function* $S_3(p_1, p_2, p_3) = p_1 + p_2 + p_3 \pmod 2$ *using different methods is considered. The function* S_3 *is chosen since it cannot be reduced with conventional methods such as the Karnaugh map [40] and the Quine-McCluskey algorithm [77], and since significantly different lengths of computational sequences are obtained for this function with different synthesis methods. Moreover,* S_3 *is required for constructing a full-adder circuit, which makes it practically important. Let* S_3 *be written in the 2–depth NAND form as*

$$\begin{aligned} S_3 \equiv\ & NAND(NAND(p_1, p_2, p_3), NAND(p_1, \neg p_2, \neg p_3), \\ & NAND(\neg p_1, p_2 \neg p_3), NAND(\neg p_1, \neg p_2, p_3)). \end{aligned} \tag{6.19}$$

The corresponding computational sequence is obtained from the constructive proof of Theorem 25, and it has length $5 + 3 \cdot 9 = 32$.

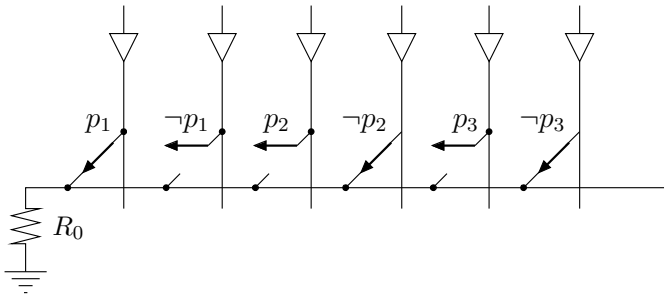

Figure 6.4: Complementary representation of input variables.

6.2.1 Complementary NAND–method

In complementary representation two memristors are reserved for each input and result variable: one memristor holds the value of the variable while the other holds its inverted value. This reduces the number of steps required for synthesizing a Boolean function, as no additional steps are required for the inverted input variables. On the other hand, complementary representation doubles the number of memristors required for computation, as is depicted in Figure 6.4.

Complementary representation can be used to reduce the lengths of the computational sequences obtained from the 2–depth NAND form. Now a general NAND–clause n_i of (6.16) does not require using an auxiliary memristor, but can instead be simply computed as

$$p_1^{\alpha_1} \to r_1, \quad p_2^{\alpha_2} \to r_1, \quad \ldots, \quad p_n^{\alpha_n} \to r_1. \tag{6.20}$$

Since one of the result memristors can be used as an auxiliary memristor during the computation, no additional auxiliary memristors are required in this method, and thus $A = \emptyset$.

Example 27. *Recall the 2–depth NAND form* (6.19) *of* S_3*. With complementary representation the corresponding computational sequence has length* $4 \cdot 5 + 1 = 21$*, and it consists of the following steps:*

$$\begin{aligned}
&p_1 \to r_2, \quad p_2 \to r_2, \quad p_3 \to r_2,\\
&r_2 \to r_1, \quad r_2 = 0,\\
&p_1 \to r_2, \quad \neg p_2 \to r_2, \quad \neg p_3 \to r_2,\\
&r_2 \to r_1, \quad r_2 = 0,\\
&\neg p_1 \to r_2, \quad p_2 \to r_2, \quad \neg p_3 \to r_2,\\
&r_2 \to r_1, \quad r_2 = 0,\\
&\neg p_1 \to r_2, \quad \neg p_2 \to r_2, \quad p_3 \to r_2,\\
&r_2 \to r_1, \quad r_2 = 0, \quad r_1 \to r_2.
\end{aligned}$$

The final step $r_1 \rightarrow r_2$ is performed to maintain the complementary representation. As a result, $r_1 = S_3(p_1, p_2, p_3)$ and $r_2 = \neg r_1$.

6.3 Minimizing the number of auxiliary memristors

No auxiliary memristors are required in the 2–depth NAND–method using complementary representation, while the number of input and result memristors must be doubled in order to maintain the complementary representation of variables. On the other hand, in the non-complementary NAND–method described in Section 6.2.1, two auxiliary memristors and a single result memristor are required. In such a non-complementary setting the number of auxiliary memristors must be non-zero, since without auxiliary memristors only Boolean functions of the form

$$f \equiv \mathrm{NAND}(p_{i1}, \ldots, p_{ik}) \tag{6.21}$$

can be synthesized. This raises the question of exactly how many auxiliary memristors are required to synthesize any given Boolean function$f : B^n \rightarrow B$, when a non-complementary representation of variables is assumed. The answer is one, and this result, adopted from our publication [62], is presented in the following.

6.3.1 Synthesis with a single auxiliary memristor

To simplify notations of this section, let the singleton sets $A = \{a\}$ and $R = \{r\}$ be combined into a set of *work memristors*

$$Q = \{q_1, q_2\} := A \cup R. \tag{6.22}$$

In the following it will be shown that all Boolean functions $f : B^n \rightarrow B$ can be synthesized by using the set Q so that the value of the function is finally stored into one of the work memristors q_i. This proves that all Boolean functions can be synthesized by using a single auxiliary memristor, when non-complementary representation of variables is used.

Let $i, j \in \{1, 2\},\ i \neq j$. An implication operation from the set of input memristors P is allowed only into q_i, while q_j is used to maintain the result of the computational sequence computed so far. The following three computational subsequences are allowed at any stage of computation:

$$SO_1 = (p \rightarrow q_i) \text{ for any } p \in P \tag{6.23}$$

$$SO_2 = (q_i \rightarrow q_j,\ q_i = 0) \tag{6.24}$$

$$SO_3 = (q_i \rightarrow q_j,\ q_i = 0,\ q_j \rightarrow q_i,\ q_j = 0,\ i \leftrightarrow j). \tag{6.25}$$

In SO_3, the expression $i \leftrightarrow j$ means that the indices of q_i and q_j are interchanged. Repeated application of SO_1 results in expressions $q_i = \neg\pi_k$, where π_k is a positive product term of the form $\pi_k = \mathrm{AND}(p_{k1}, p_{k2}, \ldots, p_{kr})$, where each $p_{kh} \in P$. SO_2 flushes the content of q_i to q_j, resulting in $q_j = \mathrm{OR}(\pi_k, q'_j)$, where q'_j is the state of q_j before applying the rule. SO_3 first flushes q_i to q_j, then q_j to q_i, and then interchanges the roles of q_i and q_j. In other words, SO_3 results in $q_j = \mathrm{NOR}(\pi_k, q'_j)$.

Using all of the three computational subsequences defined above yields Boolean functions representable by the following *recursive conjunctive form*

$$\begin{aligned} f &\equiv ((\ldots((\pi_L\ )^{\alpha_L} \vee \pi_{L-1})^{\alpha_{L-1}} \ldots \vee \pi_2)^{\alpha_2} \vee \pi_1)^{\alpha_1} \\ &= (\neg\pi_1 \rightarrow (\neg\pi_2 \rightarrow \ldots(\neg\pi_{L-1} \rightarrow \neg\pi_L^{\alpha_L})^{\alpha_{L-1}} \ldots)^{\alpha_2})^{\alpha_1}, \end{aligned} \tag{6.26}$$

where π_k are positive product terms, $\alpha_k \in \{0, 1\}$, and the notation $x^1 = x$, $x^0 = \neg x$ is used. Conversely, all Boolean functions of the form (6.26) can be computed using the set of computational subsequences $\{SO_1, SO_2, SO_3\}$. The following Theorem is proved in our publication [62].

Theorem 28. *Every Boolean function $f : B^n \rightarrow B$ can be written in the recursive conjunctive form of Equation* (6.26)*, where the terms π_i for $i = 1, \ldots, 2^n$ are the positive product terms of n input propositions ordered in reversed lexicographical order, that is, $\pi_1 = AND(p_1, p_2, \ldots p_n)$, $\pi_2 = AND(p_2, p_3, \ldots, p_n)$, $\pi_3 = AND(p_1, p_3, \ldots, p_n)$, ...,*
$\pi_{n+1} = AND(p_1, p_2, \ldots, p_{n-1})$, ..., $\pi_{2^n-2} = p_2$, $\pi_{2^n-1} = p_1$, and $\pi_{2^n} = \mathbf{F}$ is the logical constant false.

Proof. Follows immediately from Theorem 31. □

Corollary 29. *All Boolean functions $f : B^n \rightarrow B$ can be synthesized using two work memristors and thus only one auxiliary memristor.*

The reason for leaving the proof of Theorem 28 to Section 6.4 is that the proof is constructive, and it will be formulated in the case of multi-input implication logic, which is practically more important than the synthesis method discussed here. Moreover, the resulting computational sequence is optimized in Section 6.4.1 by a procedure resembling the Karnaugh map and the Quine-McCluskey algorithm. This procedure, with some changes that are discussed in [89], can also be used to reduce the length of the of the recursive conjunctive form (6.26).

Example 30. *Continuing Example 26, the parity function $S(p_1, p_2, p_3)$ can be written in the recursive conjunctive form as*

$$\begin{aligned} S \equiv (\neg\pi_1 \rightarrow (\neg\pi_2 \rightarrow (\neg\pi_3 \rightarrow (\neg\pi_4 \rightarrow \\ \rightarrow (\neg\pi_5 \rightarrow (\neg\pi_6 \rightarrow (\neg\pi_7 \rightarrow \neg\pi_8^0)^1)^1)^0)^1)^1)^0)^1, \end{aligned} \tag{6.27}$$

where

$$\pi_1 = AND(p_1, p_2, p_3), \quad \pi_2 = AND(p_2, p_3), \quad \pi_3 = AND(p_1, p_3),$$
$$\pi_4 = AND(p_1, p_2), \quad \pi_5 = p_3, \quad \pi_6 = p_2, \quad \pi_7 = p_1, \quad \textit{and} \quad \pi_8 = \mathbf{F}.$$

As was the case with the NAND–form, this expression cannot be reduced. Assuming that initially $a = r = 0$, the corresponding computational sequence has length 30*. To see this, suppose that at some point of the computation $q_1 = z$ and $q_2 = 0$, where $\{q_1, q_2\} = \{a, r\}$. In order to have $q_1 = \neg\pi \to z$, one needs first to perform the implications $p_i \to q_2$ corresponding to the input propositions in π (SO1), then to perform $q_2 \to q_1$, and finally to set $q_2 = 0$ (SO2). For example,* 5 *computational steps are required for $q_1 = \neg\pi_1 \to z$. Moreover, each inversion except $\neg\pi_8^0$ requires two additional computational steps according to the subsequence SO3. In total, the number of computational steps required to synthesize S using two work memristors is*

$$\underbrace{5 + 4 + 4 + 4 + 3 + 3 + 3}_{\neg\pi_i \to} + \underbrace{2 + 2}_{inv.} = 30.$$

6.4 Multi-input implication logic

Apparently, a practical problem with the synthesis methods discussed so far is the relatively large number of computational steps required to synthesize a given Boolean function. However, the lengths of the computational sequences can be substantially reduced with a multi-input implication operation, as will be shown in the following. In [54] we noted, that implication operations of the form

$$\mathrm{OR}(p_{i1}, p_{i2}, \ldots\ p_{ik}) \to a \tag{6.28}$$

can be realized in a single computation step by applying the voltage v_{cond} on all of the memristors in the OR–clause simultaneously as depicted in Figure 6.5. As long as the on/off conductance ratios of the memristors are large enough compared to the number of simultaneously used input memristors, the multi-input implication operation is available.

In order to maximally benefit from the multi-input implication operation, one would like to express Boolean functions in a form which consists mainly of multi-input implications, whose OR–clauses contain as many input variables as possible. Moreover, the number of implication operations should be minimized in order to minimize the total number of computational steps. These requirements are satisfied by the following Theorem, which combines results from our publications [64,89], and presents a multi-input implication form for Boolean functions.

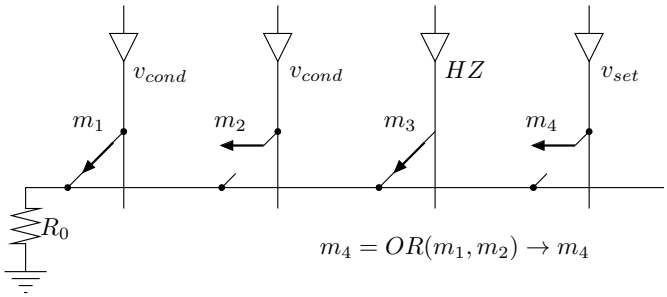

Figure 6.5: A multi-input implication operation yielding $m_4 = \mathrm{OR}(m_1, m_2) \rightarrow m_4$. The driver of memristor m_3 is set to a high impedance state.

Theorem 31. *Every Boolean function $f : B^n \rightarrow B$ can be written in the recursive multi-input implication form*

$$f \equiv \left(\sigma_1 \rightarrow \left(\sigma_2 \rightarrow \ldots \left(\sigma_{L-1} \rightarrow \sigma_L^{\alpha_L}\right)^{\alpha_{L-1}} \ldots\right)^{\alpha_2}\right)^{\alpha_1}, \tag{6.29}$$

*where the terms σ_k for all $k = 1, \ldots, L$ are OR–clauses of positive (*i.e.*, not inverted) input variables of the form*

$$\sigma_k = OR(p_{i1}, p_{i2}, \ldots, p_{ik}).$$

Proof. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of input variables of a Boolean function $f : B^n \rightarrow B$. First, all possible OR–clauses of positive input variables are ordered in a reversed lexicographical order, that is, $\sigma_1 = \mathrm{OR}(p_1, p_2, \ldots, p_n)$, $\sigma_2 = \mathrm{OR}(p_2, p_3, \ldots, p_n)$, $\sigma_3 = \mathrm{OR}(p_1, p_3, \ldots, p_n)$, …, $\sigma_{n+1} = \mathrm{OR}(p_1, p_2, \ldots, p_{n-1})$, …, $\sigma_{L-2} = p_2$, $\sigma_{L-1} = p_1$, and $\sigma_L = \mathbf{T}$, where $L = 2^n$ and $\mathbf{T}$ denotes logical constant *true*.

Denote the value of $f(0, 0, \ldots, 0)$ by $\alpha_1 \in \{0, 1\}$. Now f can be written in the form

$$f \equiv (\sigma_1 \rightarrow f')^{\alpha_1}, \tag{6.30}$$

where $f' : B^n \rightarrow B$ is some Boolean function. Equation (6.30) follows from the following facts:

1. $\sigma_1 = 0$ if and only if $p_1 = p_2 = \ldots = p_n = 0$
2. $0 \rightarrow q = 1$ for all $q \in \{0, 1\}$
3. If $\sigma_1 = 1$, the truth value of the right hand side expression of Equation (6.30) depends on f'.

Next concentrate on $f(1, 0, \ldots, 0)$. The OR–clause $\sigma_2 = 0$ if and only if $p_2 = p_3 = \ldots = p_n = 0$, but p_1 can be either 0 or 1. Since the case with all

zeros was already dealt with σ_1, the OR–clause σ_2 corresponds to the input vector $(1, 0, \ldots, 0)$. It follows as above that

$$f \equiv (\sigma_1 \to (\sigma_2 \to f'')^{\alpha_2})^{\alpha_1}, \tag{6.31}$$

where $\alpha_2 = 1$ if $f(1, 0, \ldots, 0) = f(0, 0, \ldots, 0)$ and 0 otherwise.

Next, define the set of input vectors

$$\{\mathbf{x}_i \in \{0,1\}^n | i = 1, \ldots, L\}$$

by the rule

$$\mathbf{x}_i(j) = 0 \iff p_j \in \sigma_i, \tag{6.32}$$

where $\mathbf{x}_i(j)$ denotes the jth component of the vector $\mathbf{x}_i$ and $p_j \in \sigma_i$ means that the input proposition p_j occurs in the OR–clause σ_i. For example $\mathbf{x_1} = (0, 0, \ldots, 0)$ and $\mathbf{x}_L = (1, 1, \ldots, 1)$. Let then $b_i = f(\mathbf{x}_i)$ for all $i = 1, \ldots, L$, and let $\alpha_i = 1 - \mathrm{XOR}(b_i, b_{i-1})$ for all i, where $b_0 = 1$. Now, by a similar reasoning than above with σ_1 and σ_2, one obtains

$$f \equiv \left(\sigma_1 \to \left(\sigma_2 \to \ldots \left(\sigma_{L-1} \to \sigma_L^{\alpha_L}\right)^{\alpha_{L-1}} \ldots\right)^{\alpha_2}\right)^{\alpha_1}. \tag{6.33}$$

□

Remark 32. *One auxiliary memristor is required to synthesize all Boolean functions with the multi-input implication logic. However, the only purpose of the auxiliary memristor is to invert the intermediate result at given points of the computation, and thus if some additional CMOS circuitry can be used to invert the state of the result memristor then no auxiliary memristors are required.*

Example 33. *The parity function $S(p_1, p_2, p_3)$ can be written in the multi-input implication logic form as*

$$\begin{aligned} S \equiv (\sigma_1 \to (\sigma_2 \to (\sigma_3 \to (\sigma_4 \to \\ \to (\sigma_5 \to (\sigma_6 \to (\sigma_7 \to \sigma_8^0)^1)^1)^0)^1)^1)^0)^0, \end{aligned} \tag{6.34}$$

where

$$\begin{aligned} &\sigma_1 = p_1 \vee p_2 \vee p_3, \quad \sigma_2 = p_2 \vee p_3, \quad \sigma_3 = p_1 \vee p_3, \quad \sigma_4 = p_1 \vee p_2, \\ &\sigma_5 = p_3, \quad \sigma_6 = p_2, \quad \sigma_7 = p_1, \quad \textit{and} \quad \sigma_8 = \mathbf{T}. \end{aligned}$$

As before, this expression cannot be reduced. The corresponding computational sequence has length 13, since each implication operation $\to$ takes one computational step, and each inversion takes two computational steps.

6.4.1 Reducing the disjunctive form

The synthesis methods described in this chapter which are based on the 2–depth NAND form or the conjunctive normal form benefit from the conventional reduction procedures such as the Karnaugh map or the Quine-McCluskey algorithm. Similar procedure for the recursive multi-input implication form (6.29) would be very useful, as the constructive proof of Theorem 31 uses the maximal number of $2^n - 1$ multi-input implication operations to synthesize any Boolean function $f : B^n \to B$. In the following, a reduction procedure from our publication [89] is presented.

To motivate this procedure, consider a Boolean function $f : \{0,1\}^n \to \{0,1\}$ that satisfies

$$f(0,0,\ldots,0) = f(1,0,\ldots,0). \tag{6.35}$$

By the proof of Theorem 31, f can be written in the form

$$f \equiv (\sigma_1 \to (\sigma_2 \to f'')^{\alpha_2})^{\alpha_1}. \tag{6.36}$$

However, now $\alpha_2 = 1 - \mathrm{XOR}(f(0,0,\ldots,0), f(1,0,\ldots,0)) = 1$. This allows to rewrite Equation (6.36) as

$$\begin{aligned} f &\equiv (\sigma_1 \to (\sigma_2 \to f'')^1)^{\alpha_1} \\ &= ((\neg\sigma_1) \vee (\neg\sigma_2) \vee f'')^{\alpha_1} \\ &= (\neg\sigma_2 \vee f'')^{\alpha_1} \\ &= (\sigma_2 \to f'')^{\alpha_1}, \end{aligned} \tag{6.37}$$

since $\sigma_1 = p_1 \vee \sigma_2$. Thus the disjunctive term σ_1 can be eliminated, and the number of implication operations is reduced by one.

This simplification procedure is generalized as follows. Let the sets $P(\sigma_i)$ for $i = 1,\ldots,L = 2^n$ be defined as

$$P(\sigma_i) = \{\mathbf{x} \in \{0,1\}^n | \mathbf{x}(j) = 0 \ \text{ if } \ p_j \in \sigma_i\}. \tag{6.38}$$

For example, $P(\sigma_1) = \{(0,0,\ldots,0\}$ and $P(\sigma_2) = \{(0,0,\ldots,0),(1,0,\ldots,0)\}$. Perform then the following steps.

1. Initialize $k = 1$ and $Q = \{0,1\}^n$.

2. While $Q \neq \emptyset$, denote $S_i = P(\sigma_i) \cap Q$ for all $i = 1,\ldots,L$. Choose among the sets S_i a set S_j with maximal cardinality which satisfies

   $$f(\mathbf{x}) = c \ \ \forall \mathbf{x} \in S_j,$$

   where c is a constant (either 0 or 1). Set $Q = Q \setminus S_j$, $b_k = c$, $r_k = j$, and $k = k + 1$.

   Repeat this step until $Q = \emptyset$.

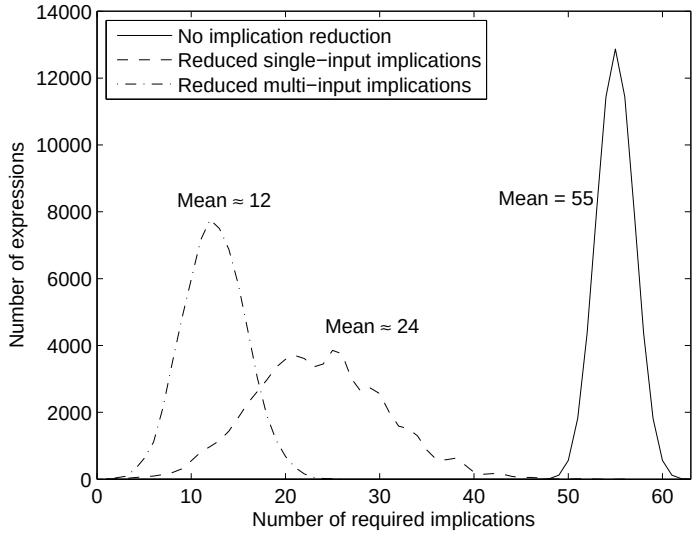

Figure 6.6: Distributions of lengths of computational sequences required for Boolean functions with four variables.

3. Denote the number of terms b_k obtained above as K. Set $b_0 = 1$. Calculate $\alpha_k = 1 - \mathrm{XOR}(b_k, b_{k-1})$ for every $k = 1, \ldots, K$. While $\alpha_K = 1$, set $K = K - 1$.

Now the Boolean function f can be written in the form

$$f \equiv (\sigma_{r_1} \to (\sigma_{r_2} \to \ldots (\sigma_{r_{K-1}} \to \sigma_{r_K}^{\alpha_K})^{\alpha_{K-1}} \ldots)^{\alpha_2})^{\alpha_1}. \tag{6.39}$$

Note the following:

i) In step 2) a set S_j always exists. To see this, let $\mathbf{x} \in Q$ have a minimal Hamming weight, and let σ be the corresponding disjunctive term (cf. proof of Theorem 31). Then $P(\sigma) \cap Q = \{\mathbf{x}\}$.

ii) In step 3) the reduction of terms can be made since $\sigma_L = \mathbf{T}$ and $q \to \mathbf{T} = \mathbf{T}$ regardless of the value of q.

iii) The possible term $(\sigma \to \mathbf{T}^0)^0$ in Equation (6.39) simplifies to σ.

Remark 34. *The reduction procedure described above can be modified for the single-input implication method discussed in Section 6.3. Figure 6.6 shows the numbers of computational steps required for synthesizing Boolean functions with four variables, with and without using the reduction procedure.*

6.4.2 NAND–OR method

As the final synthesis method described in this chapter, a multi-input method using complementary representation of variables is presented. In this NAND–OR synthesis method [64] the idea is to write a Boolean function $f : B^n \to B$ in the "inverted conjunctive normal form"

$$f \equiv \mathrm{NAND}(o_1, \ldots, o_k), \tag{6.40}$$

where each o_i is an OR-clause of variables and their negations. The computation of the value of f can then be performed as a sequence

$$o_1 \to r, \quad o_2 \to r, \quad \ldots, \quad o_k \to r, \tag{6.41}$$

where r is the result memristor and the operations $o_i \to r$ are multi-input implication operations of the form (6.28). As the OR–clauses contain input variables and their negations, these must be available for the implication operation. In practice, one may use the complementary representation discussed in Subsection 6.2.1, in which case no auxiliary memristors are needed, but then two result memristors r_1 and r_2 must be used in order to maintain the representation. An alternative solution is to use n auxiliary memristors, and to store the negations of the input variables to these memristors at the beginning of the computation. This solution requires n additional computational steps when compared to the complementary version, but may reduce the overall number of memristors in an implication circuit almost to half, as the auxiliary memristors form a shared resource that can be used in subsequent Boolean computations with new sets of input and result memristors.

Example 35. *The parity function $S_3(p_1, p_2, p_3)$ can be written in the NAND–OR form as*

$$\begin{aligned} S_3 \equiv\ & NAND(OR(p_1, p_2, \neg p_3), OR(p_1, \neg p_2, p_3), \\ & OR(\neg p_1, p_2, p_3), OR(\neg p_1, \neg p_2, \neg p_3)). \end{aligned} \tag{6.42}$$

Thus the corresponding computational sequence has the following five steps:

$$\begin{aligned} & OR(p_1, p_2, \neg p_3) \to r_1, \quad OR(p_1, \neg p_2, p_3) \to r_1, \\ & OR(\neg p_1, p_2, p_3) \to r_1, \quad OR(\neg p_1, \neg p_2, \neg p_3) \to r_1, \quad r_1 \to r_2, \end{aligned}$$

if complementary representation of all variables is used. When $n = 3$ auxiliary memristors is used, the computational sequence is

$$\begin{aligned} & p_1 \to a_1, \quad p_2 \to a_2, \quad p_3 \to a_3, \quad OR(p_1, p_2, a_3) \to r, \\ & OR(p_1, a_2, p_3) \to r, \quad OR(a_1, p_2, p_3) \to r, \quad OR(a_1, a_2, a_3) \to r, \\ & a_1 = a_2 = a_3 = 0. \end{aligned}$$

When the auxiliary memristors are all reset simultanously, this approach requires eight computational steps.

Method	CR	MI	# Aux. memr.	# Steps
NAND			2	32
Complementary NAND	X		0	21
Recursive			1	30
Recursive multi-input		X	1	13
NAND–OR with n aux. memr.		X	n	8
NAND–OR	X	X	0	5

Table 6.2: Comparison of the presented synthesis methods. CR and MI refer to using complementary representation and multi-input implication respectively. The entry # Steps corresponds to the number of computational steps required to synthesize the parity function S_3.

6.5 Summary of the synthesis methods

In the above, several different synthesis methods for memristive implication logic were presented. In Table 6.2 their properties are summarized, and the number of computational steps — implications and resets — required for computing the parity function S_3 are given. The number of required computational steps depends strongly on whether or not multi-input implication and complementary representation of variables can be used. The availability of multi-input implication depends on the on-off ratio of the devices, while complementary representation doubles the number of memristors.

6.5.1 Worst-case algorithmic complexities

The complementary NAND–OR method requires at most $2^{n-1} + 1$ steps. This follows from the fact that the minimum number of OR–clauses in the conjunctive normal form of a Boolean function and its complement equals 2^{n-1}. Moreover, one computational step is required for maintaining the complementary representation of the result of the computation. When a non-complementary version with n auxiliary memristors is used, the worst-case number of computational steps is $2^{n-1} + n + 1$, as each of the input variables must to be implicated to the corresponding auxiliary memristor.

The complementary 2–depth NAND–method requires at most $(n+2)2^{n-1}+1$ operations. This follows from the above and the fact that each clause in the 2–depth NAND–form contains n propositions, that is, variables or their negations.

The non-complementary 2–depth NAND–method requires at most $(2n + 2)2^{n-1}$ operations. This follows from the fact that representing each proposition inside a NAND clause requires on average $(1/2) \cdot 1 + (1/2) \cdot 3 = 2$ computational operations — each variable requires 1, and each negated variable requires 3 computational operations.

The multi-input recursive method requires at most $3(2^n - 1)$ computational operations, corresponding to the case $\alpha_i = 0$ for all i. Without multi-input, all the positive product terms must be constructed as sequences of implications thus requiring in total at most $(8 + n)2^{n-1} - 4$ computational steps. This follows from the fact that each positive product term of length i requires i implications and a reset operation. Since

$$\sum_{i=1}^{n}(i+1)\left(\begin{array}{c} n \\ i \end{array}\right) = 2^n + n2^{n-1} - 1, \tag{6.43}$$

the maximum number of computational steps equals

$$3(2^n - 1) + 2^n + n2^{n-1} - 1 = (8 + n)2^{n-1} - 4. \tag{6.44}$$

One may thus conclude that asymptotically the NAND–OR method and the recursive multi-input method belong to the same complexity class, while the other synthesis methods are asymptotically more complex.

Figure 6.7 shows the numbers of computational steps required for synthesizing Boolean functions with four variables, when no reduction procedure is used. Results for the NAND, complementary NAND, and NAND-OR methods were obtained by choosing the minimum of the numbers of steps required to synthesize a Boolean function and its complement.

6.6 Limitations and improvements

Even when the multi-input operation and the complementary representation of variables are used, the foremost disadvantage of memristive implication logic is the necessity of performing the computations as a sequence of operations. Sequences take processing time, and require rather complicated control signals, which must be stored into and retrieved from some additional memory. On the other hand, an advantage of stateful logic is that no long-range data transfer is required, as the computation is performed directly at the memory.

Another major limitation is the lack of multi-output operations. Indeed, suppose that one would like to perform simultanously the implication operations $m_2 \to m_3$ and $m_2 \to m_4$ in the circuit of Figure 6.5. This is not possible, since as initially $m_3 = 1$, it follows that the voltage v_{set} is driven on the horizontal wire, and thus the voltage over m_4 is insufficient to program it to the on-state. In [42], a fan-out method which copies the state of a memristor to the states of other memristors was proposed. However, this method uses switches to divide the horizontal wire into smaller pieces, and thus trades off some of the density available in the stateful logic circuit.

As was noted in Remark 32, multi-input implication logic benefits from the addition of a CMOS subcircuit which implements the inversion of the

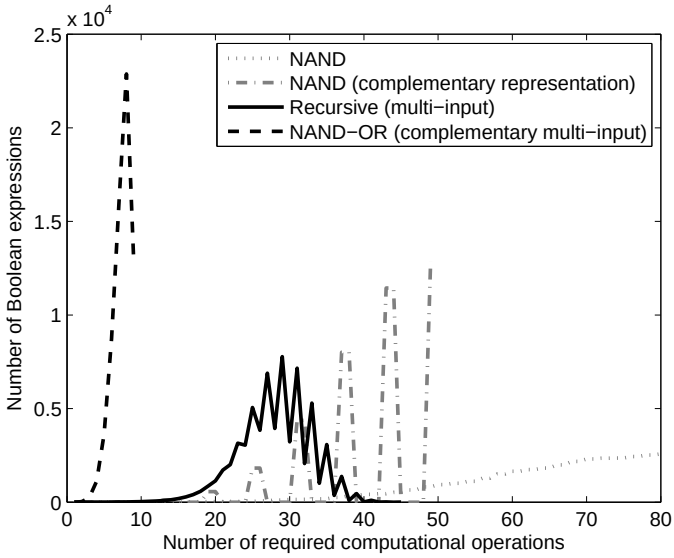

Figure 6.7: Distributions of the total numbers of implication and reset operations required for computing the values of all Boolean functions with four inputs. The maximum number of computational operations required by the NAND–OR method equals 9.

state of a memristor, as this eliminates the need for auxiliary memristors. One could also consider other auxiliary circuits to shorten the computational sequences. In [84], Pershin and di Ventra suggest adding a capacitor to the common horizontal line of the circuit depicted in Figure 6.1. This allows to store intermediate results of the computing as the charge of the capacitor and it also shortens the computational sequences.

For example, to compute AND(m_1, m_2), one charges the capacitor for a short time first through the memristor m_1 and then through the memristor m_2. Then the capacitor is connected to the memristor m_3, which is initially at state $m_3 = 0$. If both of the memristors m_1 and m_2 are in the conducting state, $m_1 = m_2 = 1$, the voltage across the capacitor becomes large enough to switch the state of m_3. In all other cases the voltage across the capacitor is too small to change the state of m_3.

This approach reduces the number of computational steps, and thus seems promising for practical applications. Moreover, Pershin and di Ventra suggest the capacitor to be replaced by a memcapacitor, whose state can evolve during the computation thus yielding an additional computational resource. However, it seems that such hybrid circuits may not cope as well with the physical variances of the memristors as the "pure" memristive implication logic does. For example, in the computation of the AND-function

m_1	m_2	$m_2 := m_1 \not\leftarrow m_2$
0	0	0
0	1	1
1	0	0
1	1	0

Table 6.3: The truth table of converse non-implication, which is implemented by choosing $v_{\mathrm{cond}} > 0$ and $v_{\mathrm{set}} < 0$.

described above, a mismatch of 100% in the value of the conductance at the on-state of a memristor will ruin the computation. Indeed, if the conductance of the memristor m_1 in state $m_1 = 1$ is twice of its nominal value, the charge in the capacitor becomes large enough to switch m_3 even if m_2 is in off-state. In pure memristive implication logic, even as large mismatches as the above may be tolerated, as long as the on/off conductance ratio is large enough.

In any case, to make memristive implication logic truly worthwhile, it should be parallelized as suggested in [12, 84]. This idea is further discussed in Chapter 8.

6.7 Converse nonimplication

To conclude this chapter, the use of rectifying binary memristors for stateful logic is discussed. As noted in Section 6.1, rectifying memristors are required to enable the stateful logic operation which corresponds to the following choice of the driving voltages:

$$0 < v_{\mathrm{cond}} < V^T, \quad -V^T < v_{\mathrm{set}} < 0, \quad \text{and} \quad v_{\mathrm{set}} - v_{\mathrm{cond}} < -V^T. \tag{6.45}$$

When rectification is used to prevent negative currents in the circuit of Figure 6.1, it follows that m_2 is programmed to the off-state exactly when $m_1 = 1$. The corresponding logical connective is called converse nonimplication, denoted by the symbol $\not\leftarrow$, and its truth table is given in Table 6.3. The resulting state of m_2 can now be written as

$$m_2 := m_1 \not\leftarrow m_2 = \mathrm{AND}(\neg m_1, m_2). \tag{6.46}$$

The truth table of converse nonimplication equals that of material implication when the logical interpretations of the memristors' states are interchanged, or when in the truth table zeros are substituted with ones and vice versa. Therefore the synthesis results described in Sections 6.2 and 6.3 can be straightforwardly translated for converse nonimplication. However, the multi-input operation for converse nonimplication is not as useful as it

was for implication logic, since

$$\mathrm{OR}(m_1, \ldots, m_k) \not\leftarrow m_2 = \mathrm{AND}(\neg m_1, \ldots, \neg m_k, m_2), \tag{6.47}$$

that is, only AND-clauses result from multi-input converse nonimplication.

A practical advantage of memristive implementation of converse nonimplication is that the keeper subcircuit described in Section 6.1 is no more required. This follows from the fact that the programming voltage v_{set} never interferes with the conditional voltage v_{cond} due to the rectification. This fact has another, very significant corollary: in contrast to the memristive implication operation, converse nonimplication allows for multi-output operations. Indeed, a multi-output converse nonimplication operation is performed simply by driving the programming voltage v_{set} on multiple memristors simultaneously. This allows for example copying the negated state of a memristor simultaneously to multiple memristors. Multi-output converse nonimplication will be further discussed in Section 8.3 which describes the parallelization of memristive stateful logic.

Chapter 7

Memristive Crossbars

So far in this thesis circuits and systems consisting of only small numbers of memristors have been considered. However, one of the main driving forces of memristor technology is the prospect of crossbar architectures with very large numbers of memristive devices.

This chapter discusses the design and operation of memristive crossbar arrays. First, general properties of a memristor crossbar are described in Section 7.1. Section 7.2 presents two different approaches to interface CMOS circuitry with a memristive crossbar. In particular, in Subsection 7.2.1 a demultiplexer architecture is briefly described, while the rest of Section 7.2 concentrates on the CMOS/molecular hybrid approach, as it is currently the most promising interface design. Section 7.3 concludes this chapter with an investigation on write and read operations in a memristive crossbar.

7.1 Memristive crossbar

A memristive crossbar is depicted in Figure 7.1. 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 [34, 43, 57, 105]. A typical nanowire half pitch of the nanowires in the reported physical memristive crossbars so far is in the range 30 nm – 100 nm. The memristive material is laid between the two nanowire layers, and as a result, a memristor is formed at each crosspoint of two nanowires. Some significant advantages of a memristive crossbar structure are listed in the following.

- The memristive crossbar is a memory structure whose each memory cell consists of a single device.
- It has the highest possible device integration density $1/(4F^2_{\text{nano}})$. Here

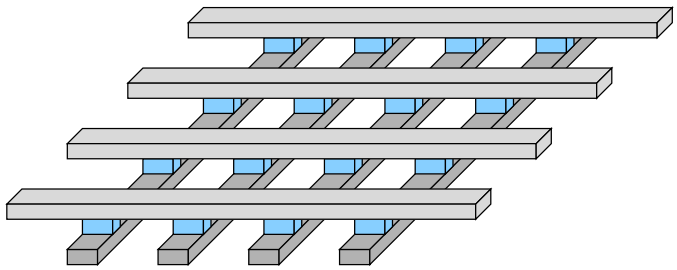

Figure 7.1: Memristive crossbar. Here the horizontal and vertical nanowire layers are depicted in gray, whereas the memristive layer is depicted in light blue.

F_{nano} is the lithographic feature size, or half-pitch, of a nanowire, which can be potentially scaled down to a few nanometers [34, 107].

- The crossbar structure is self-aligned: only the dimension between the two electrodes of a memristor needs to be cricitically controlled [34].

Only a few fabricated large-scale memristive crossbar arrays have been reported so far. In [16], a metal-oxide memory with a capacity of 64 Mb was presented. To my best knowledge this is the largest memristive memory crossbar that has been reported so far. In [34] Jo et al. described the fabrication process of a 1 kb crossbar array consisting of the M/a-Si/p-Si -memristors, which were discussed in Section 3.3.2. An advantage of a M/a-Si/p-Si type memristor is that it is fabricated using standard semiconductor materials, and therefore it is CMOS compatible. Moreover the a-Si switching medium allows high-yield fabrication: for the 1 kb crossbar the reported yield was 98% [34].

7.2 Interfacing memristive crossbars with CMOS circuitry

Since memristors are passive devices, nanowires must be driven from outside of the crossbar. This can be realized by complementing the memristive crossbar array with a CMOS layer that provides signal restoration and gain. The large feature size of the CMOS layer compared to the considerably smaller one of the nanowire crossbar does not need to be a problem, since the number of memristors in the nanowire crossbar is quadratic with the number of nanowires to be driven, as can be seen from Figure 7.1. In this section, two different interfacing approaches, namely, the demultiplexer and the CMOL architectures, are described.

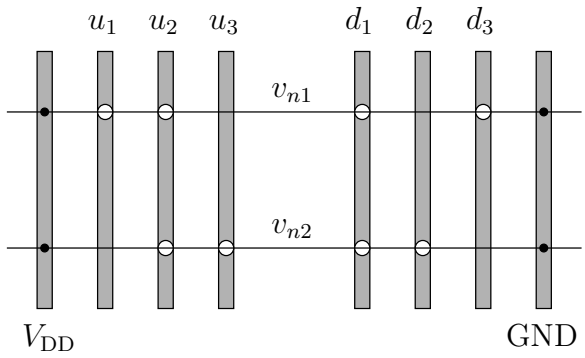

Figure 7.2: Nanowire demultiplexer using field-effect-transistors as the junction components. Microwires are depicted as the wide vertical gray lines, while the nanowires are depicted as the horizontal black lines. The FETs are depicted as white circles; microwires supplying the rail voltages are connected by vias to the nanowires.

7.2.1 Demultiplexers

Demultiplexer architectures for interfacing memristive crossbars with CMOS circuitry were first studied in [15, 22, 120]. By definition, a demultiplexer is a logic component that uses a small number M of input address lines to select exactly one of N output data lines [99]. A demultiplexer for driving the nanowires of a memristive crossbar can be implemented as a crossbar structure consisting of microwires and nanowires. At chosen junctions of a microwire and a nanowire there exists an electrical component which is used to control the voltage on the nanowire.

Two nanowire demultiplexers, one for each of the rail voltages V_{DD} and GND, are depicted in Figure 7.2. Here the components located at some of the junctions of microwires and nanowires are assumed to be field-effect-transistors (FETs), which are gated by the microwires. The FETs reside on the nanowires and are connected in series so that all the FETs located on a given nanowire must be gated in order to pass a rail voltage onto that nanowire. For example, a high voltage on the microwires u_1 and u_2 passes the rail voltage V_{DD} onto the nanowire n_1.

Other demultiplexer architectures based on nanoscale resistors and diodes have been proposed for example in [99]. In general it is advantageous that the microwire-nanowire junction components have nonlinear $I - V$ characteristics [99]. Furthermore, coding theory can be used to decide optimal junction patterns for these components, as has been considered in [51] and [90].

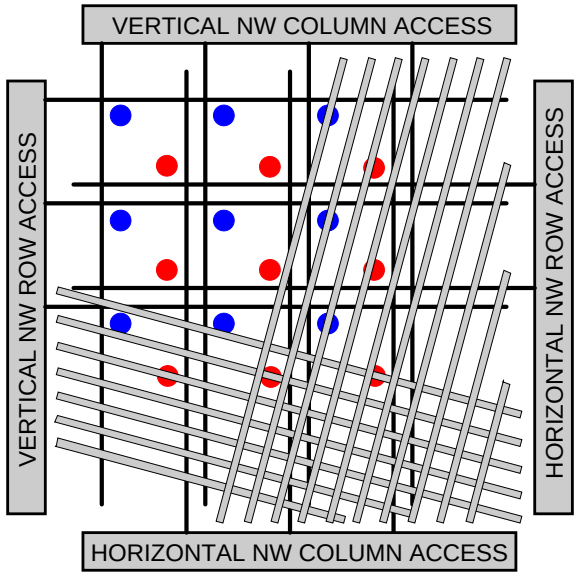

Figure 7.3: Layout of a CMOL circuit. The CMOS cells are depicted as square tiles, with circular interfaces to the nanowire crossbar. The nanowire crossbar is depicted as a mesh of line segments; notice that only part of the crossbar is depicted to make the CMOS layer visible. Each CMOS cell is addressed by four microwires which are connected to the address decoders depicted 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 Figure 7.4 b). Although not depicted here, there exists a memristor at each crossing of the nanowires. The abbreviation NW stands for nanowire.

7.2.2 CMOL-type architectures

In the CMOS/molecular hybrid [71] (CMOL) architecture the memristive crossbar is fabricated on top of a layer of CMOS cells. The cells are comprised of pass-transistors, CMOS-to-crossbar pins, and inverters. Each of the horizontal nanowires is connected to the input of one of the inverters, while the vertical nanowires are similarly connected to the outputs of the inverters. The memristive crossbar is slightly rotated with respect to the CMOS layer so that each nanowire can be electrically connected to exactly one pin extending up from the CMOS layer [100]. An abstract view of the memristive CMOL architecture is depicted in Figure 7.3.

Figure 7.4a) provides a more detailed view of the pins connecting the CMOS layer to the memristive crossbar. As can be seen, the pins must grow thinner towards the nanowires, and the tips of the pins are of the same size with the nanowire half-pitch. Such a design may pose some difficulties

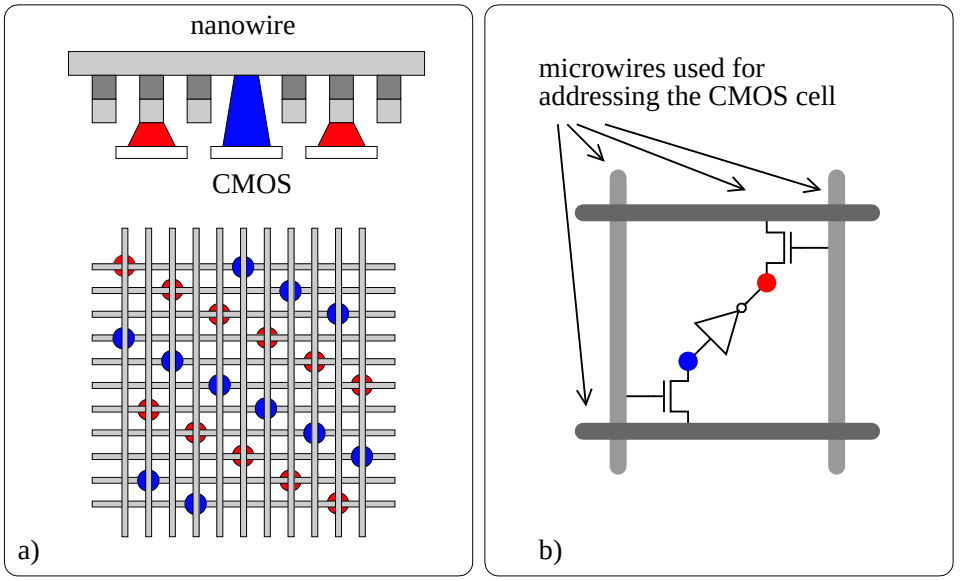

Figure 7.4: Memristive CMOL architecture. a) A nanowire crossbar on top of a CMOS layer. The CMOS pins are depicted as red and blue circles. The red pins are connected to the lower layer or the crossbar, while the blue pins are connected to the upper layer. b) A CMOS cell used in the CMOL architecture, consisting of two pass transistors, CMOS pins, and an inverter.

to the fabrication, and may require also nanoscale alignment. Possible solutions to these problems are described later in this section, where the field programmable nanowire interconnect architecture and the CMOL variant with segmented nanowires are presented.

A CMOS cell used in the CMOL architecture is depicted in Figure 7.4b). Each cell is addressed by four microwires which are connected to the address decoders depicted in Figure 7.3. The four-line addressing is used in order to independently control the horizontal and the vertical nanowires. Any memristor in the crossbar can be selected by an appropriate choice of a horizontal and a vertical nanowire. Addressing in CMOL architecture is discussed in more detail in Subsection 7.2.3.

Example 36. *Assuming that the crossbar consists of rectifying binary memristors, the CMOL architecture can be used to implement wired-OR logic as depicted in Figure 7.5. In this example it is assumed that the vertical nanowires are connected to the horizontal nanowire by memristors that are in the high-conductance state. Then the voltage on the horizontal nanowire equals the OR of the voltages on the vertical nanowires, and this result is inverted at the CMOS cell thus yielding* $NOR(A, B)$*. Here it is also assumed that there exists a resistive path to ground at the input of this inverter to*

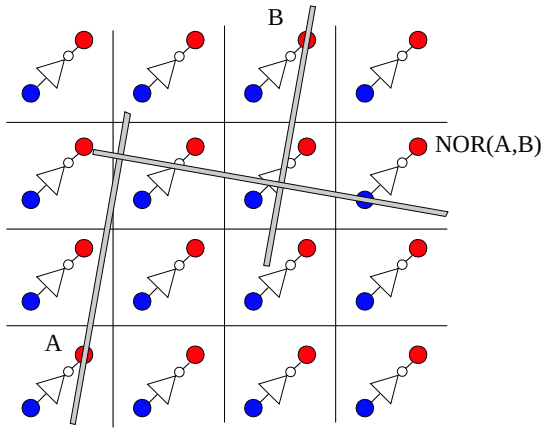

Figure 7.5: Using a CMOL architecture for wired-OR logic.

enable such resistive logic. Wired-OR and inversion make the CMOL architecture functionally complete, since all Boolean functions can be synthesized by using only OR-gates and inverters.

The pins on the surface of the CMOS layer are distributed uniformly, which makes CMOL an extremely dense architecture. However, the aligment of the nanoscale pins poses also a challenge for fabrication. Moreover, due to the non-complementary nature of the wired-OR logic, a low supply voltage V_{DD} is required to keep the static power dissipation within reasonable limits. An initial estimate for a large-scale CMOL supply voltage V_{DD} is about 0.3 V, which is lower than projected for CMOS circuits by the International Technology Roadmap for Semiconductors (ITRS) through the year 2020 [1, 100].

To overcome the aforementioned challenges, a generalization of the CMOL architecture called the *Field Programmable Nanowire Interconnect* (FPNI) was proposed by Snider and Williams in [100]. In FPNI, logic is performed at the CMOS layer, and only signal routing is realized by the memristive crossbar. Accordingly, other logical primitives than just inverters are needed at the CMOS layer. In FPNI, the memristive crossbar is connected to CMOS logic components such as AND and OR gates, latches and buffers — in principle, any multi-input logic gate could be used. To facilitate the fabrication, the nanowires are suggested to form CMOS feature sized pads, to which the CMOS pins are connected to. A FPNI-type nanowire layer is depicted in Figure 7.6.

Remark 37. *To simplify the nomenclature, in the rest of this thesis any architecture consisting of a memristive crossbar fabricated on top of an array*

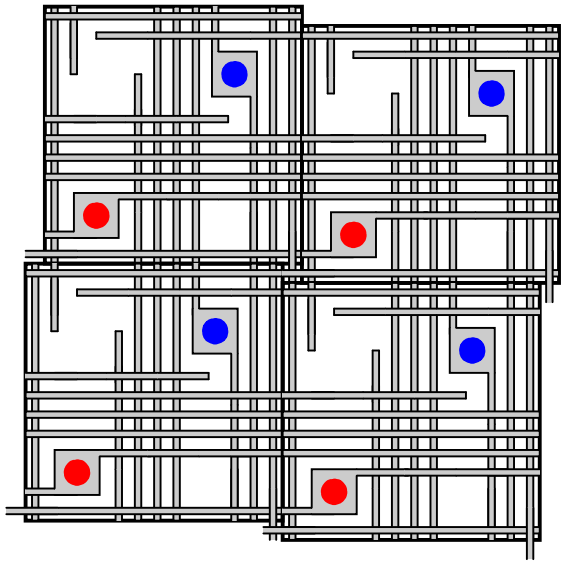

Figure 7.6: FPNI-type nanowire crossbar. Instead of nanoscale CMOS pins, larger pads are formed onto the nanowires. The CMOS cells are shown by black square tiles. Although not shown here, they can contain arbitrary CMOS logic circuitry.

of CMOS cells is called CMOL-type, regardless of the design of the cells or the CMOS pins.

7.2.3 Addressing in a CMOL-type architecture

As noted above, 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 to select any pair of a horizontal and a vertical nanowire. This selection is then used for reading from and writing to the memristive crossbar as is further described in Section 7.3.2.

Suppose that the CMOS cells be arranged as a $\sqrt{N} \times \sqrt{N}$ grid consisting of in total N cells. Then $4\sqrt{N}$ microwires are required for addressing the cells. To retain unique addressing of memristors, the memristive crossbar can then have at most N^2 memristors; one for each crosspoint of N horizontal and N vertical nanowires, where each nanowire is connected by a via to a CMOS cell. In practice, the number of memristors within a single memristive crossbar is determined by the nanowire pitch, and is typically smaller than N^2 — this and the effect of segmenting the nanowires on the number of memristors is discussed in the next subsection.

A limitation of this addressing scheme is that only a single horizontal and a single vertical nanowire can be selected at a time. Indeed, a simultaneous

selection of two CMOS cells that are not on the same CMOS row or column unavoidably leads to the selection of two undesired CMOS cells. This restriction to a single-junction selection may not be a problem in pure memory architectures such as the ones discussed in Section 8.1, but in other applications it would be advantageous to be able to select multiple nanowires at once. For example, parallel implication logic described in Section 8.3 benefits significantly from multi-input and multi-output operations.

A direct, although area consuming, solution to this problem is to duplicate the CMOS wiring and the pass transistors to allow additional addressing. Another straighforward solution is to add a local memory bit to each of the CMOS cells to indicate whether or not those cells should participate into the driving of the nanowires. These bits can be written in row-parallel manner — one CMOS cell row at a time — thus requiring at most $\sqrt{N}$ programming cycles for the whole CMOS layer.

7.2.4 Segmented nanowires

In the initial version of CMOL architecture [71], interface pins to the upper nanowire layer were supposed to pass between nanowires on the lower level. This approach requires precise nanoscale aligment and poses challenges for the fabrication. To facilitate the fabrication, in [102] an improved version of the CMOL architecture was proposed, in which the upper-layer pins intentionally interrupt the lower-layer nanowires. In theory, such a modification improves the circuit yield substantially, raising its theoretical upper bound to 100% even without any nanoscale aligment [102].

In the following it is assumed that both of the nanowire layers consist of segmented nanowires. This mitigates the fabrication requirements even further and is well-suited for computing architectures that operate with locally connected processor elements, as is for example the case with the Cellular Neural Networks discussed in Chapter 8. In principle, such segmentation reduces the density of the crossbar, as the connectivity domains of the CMOS cell become smaller. A countermeasure to this density reduction is presented in Section 7.2.5, where the vertical stacking of memristive crossbars is discussed.

In the left inset of Figure 7.7 a few different choices for the placement of nanowire segments with respect to CMOS cells are presented. Here the CMOS cell half-pitch is assumed to be F. For the simplicity and replicability of the connectivity domain it is assumed that each upper layer nanowire is centered on a blue pin as depited in Figure 7.7, and that these nanowires end just before a lower level pin depicted by red circles. Moreover, to simplify the geometry, it is assumed that a nanowire segment does not pass horizontally across any other upper level vias. Now the upper half of the nanowire segment corresponds to the hypotenuse of a right-angled triangle, whose

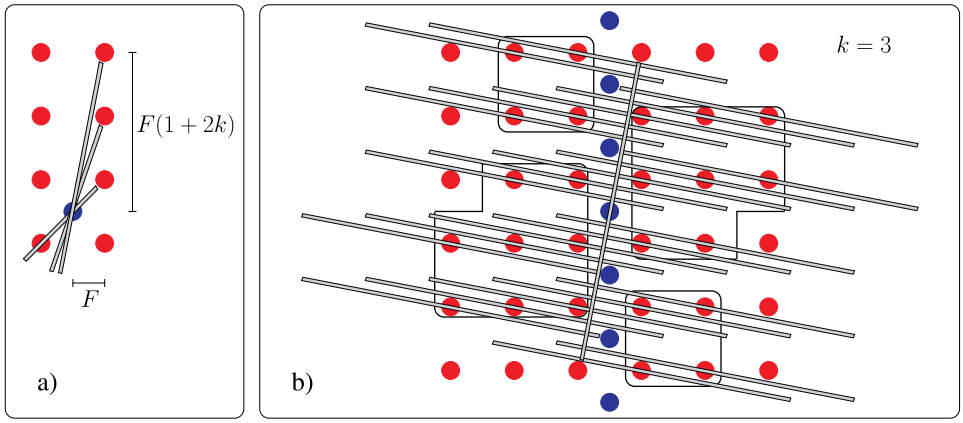

Figure 7.7: CMOL-type memristive crossbar with segmented nanowires. Left inset: different nanowire segments corresponding to different choice of the parameter k. Right inset: The connectivity domain of single vertical nanowire, with $k = 3$, is depicted by solid black polygons.

shorter edges have lengths F and $F(1+2k)$, where k is an integer. Choosing the length d of a nanowire segment corresponds to fixing the parameter k, as

$$d = 2F\sqrt{1 + (1 + 2k)^2}. \tag{7.1}$$

The tilting angle α of the memristive crossbar now equals

$$\alpha = \arctan(1 + 2k), \tag{7.2}$$

and as can be seen from right the inset of Figure 7.7, the number γ of memristors on a single nanowire segment equals

$$\gamma = 2(k^2 + (k - 1)^2 - 1). \tag{7.3}$$

Ignoring the clipping effect at the fringe of the memristive crossbar which results from the fact that the nanowire segments at the fringes have smaller connectivity domains, this implies that altogether γN memristors can be addressed with the $4\sqrt{N}$ microwires at the CMOS layer. The larger the parameter k, the larger the number of addressed memristors, and correspondingly the smaller the required feature size F_{nano} of the memristive crossbar.

Remark 38. *The tilting angle of the memristive crossbar can also be written in the form*

$$\alpha = \arcsin(F_{nano}/\beta F), \tag{7.4}$$

where $\xi > 1$ is a dimensionless constant, whose value depends on the CMOS cell complexity [102]. As shown in [107], the number of memristors on a

single segmented nanowire crossbar using this notation equals

$$N(\beta F/F_{nano})^2. \tag{7.5}$$

7.2.5 Vertical stacking of memristive crossbars

In [107], Strukov and Williams suggest vertical stacking of multiple memristive crossbars on top of the CMOS layer. They note that some of the major problems related generally to vertical stacking in integrated circuits do not necessarily apply here, as no active components are situated on the vertical stack. Their main idea, as illustrated in Figure 7.8, is to translate the connectivity domains of the CMOS cells while ensuring that only one memristor in all of the crossbars can be addressed by any allowed four-dimensional address. As a result, a vertical stack of M memristive crossbars allows for addressing γMN memristors, when the clipping effect at the fringes of the crossbar is neglected.

Translating the connectivity domain requires via translation layers. Using the notation introduced in the previous section, each upper level via could be translated at each via translation layer a length of $2F(2k-1)$ in the horizontal direction to the right and a length of $4F$ in the vertical direction downwards as depicted in Figure 7.8, while the lower level vias should not be translated at all.

Strukov and Williams predict that over the long term at least of the order $M = 100$ memristive crossbars could be vertically stacked with the feature size $F_c = 10$ nm yielding a theoretical memory density as high as 100 terabits per square centimeter.

7.3 Accessing and programming memristors within a crossbar

In this section it is assumed that an interface between the CMOS layer and the memristive crossbar is established, and that this interface allows for driving all of the nanowires independently and simultaneously to a chosen voltage. This best-case scenario assumption is used to investigate the inherent performance limitations of a memristive crossbar. First, in Subsection 7.3.1, the so-called half-select problem and a related sneak current phenomenon in memristive crossbars are discussed. Subsection 7.3.2 describes different schemes for writing to and reading from a memristive crossbar, and Subsection 7.3.3 addresses the problem of nanowire resistance.

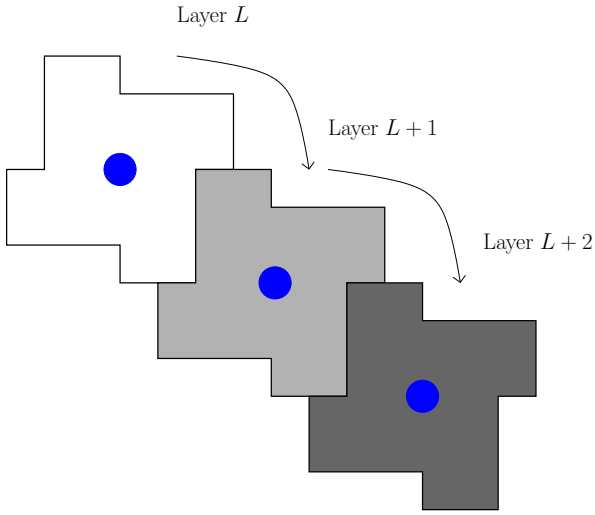

Figure 7.8: Translation of the connectivity domain in the vertically stacked crossbar structure by lower level via translation. The blue circles represent the vias from a single CMOS cell to layers L, L+1 and L+2 of the memristive crossbar stack. The connectivity domains resulting from the translation are depicted as gray polygons. Each of these domains corresponds to a different set of CMOS cells connected to horizontal nanowires as depicted by the red circles in Figure 7.6.

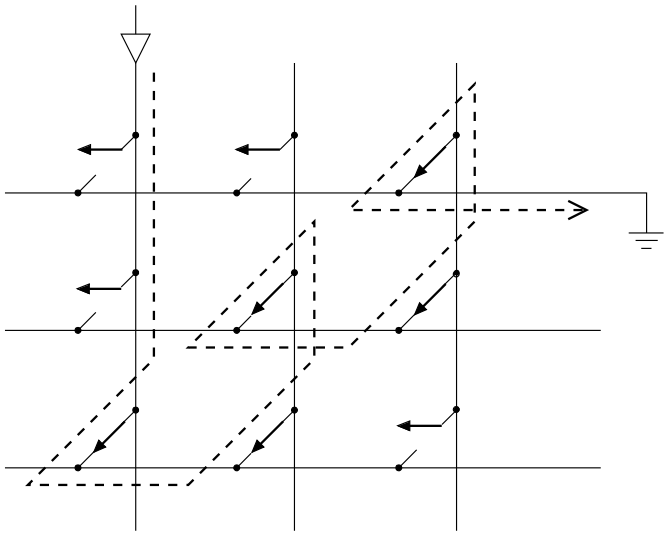

Figure 7.9: Illustration of the half-select problem. The leftmost vertical nanowire is driven with a positive write or read voltage, and the first horizontal nanowire is set to ground. A sneak current path, depicted as a dashed line, driving the other vertical nanowires and disturbing the read operation on the first horizontal nanowire is formed.

7.3.1 Half-select problem and sneak paths

The *half-select problem* is inherent to all memristive crossbar structures. This problem arises when a memristor is selected by driving the nanowires between which the device is located. Then all the other memristors connected to these nanowires are subject to the same voltage biases, and are defined as *half-select cells* [115].

As an example of the half-select problem, consider the crossbar circuit depicted in Figure 7.9, and suppose that the memristor at the upper left corner of the crossbar should be programmed to the on-state. For this, the leftmost vertical nanowire is driven to a programming voltage V_{prog}, while the first horizontal nanowire is set to ground, and all the other nanowires are left floating. Now the memristors on the driven nanowires are half-selected, and a *sneak current path* is formed from the driven vertical nanowire to the middle vertical nanowire. Now the middle vertical nanowire is biased towards V_{prog}, and the second memristor on the first horizontal nanowire may be programmed inadvertently.

Read sneak current paths denote a similar phenomenon as discussed above in the case when a memristor's state is read. Indeed, currents flowing through the sneak paths may disturb the measuring of the current through the selected memristor. In Figure 7.9, a read sneak current path disturbing

the read operation of the upper left memristor is depicted by the dashed line.

There are several solutions to these problems. One is to use rectifying memristors such as the one discussed in Subsection 3.3.3 to allow current flow only to one direction in the crossbar. Another solution is to use specific biasing schemes which do not allow nanowires to float. Such schemes are discussed in detail in the following.

7.3.2 Writing to and reading from a memristive crossbar

This subsection considers different write and read configurations of a memristive crossbar as suggested in [26,68,111]. Using the nomenclature common in memory architecture literature, the horizontal and vertical nanowires of the crossbar are called *word lines* and *bit lines*, respectively. The write and read configurations are

(W) To program a memristor, a $V/2$ writing scheme is used, where the selected word line is biased at a programming voltage V_{prog}, the selected bit line is grounded, and all the unselected word lines and bit lines are biased at $V_{\mathrm{prog}}/2$.

(R) To read a memristor's state, the selected word line is biased with a non-destructive read voltage V_{read}, and all the other word lines and bit lines are grounded. The bit line currents are then fed to sense amplifiers, which act as current-to-voltage converters, to determine the state of the selected memristor.

In the read operation (R), all memristors on the selected word line dissipate power. To reduce this power dissipation, the read currents should be minimized, which implies that the memristors should have as high resistance as possible. To further reduce the power dissipation, other read configurations have been proposed [26], two of which are presented below.

(R') Select one word line and ground one bit line corresponding to the memristor to be accessed. Ground the other word lines and leave the other bit lines floating.

(R") Select one word line and ground some of the (*e.g.*, every 5th) bit lines simultaneously. Ground the other word lines and leave the other bit lines floating.

These two read configurations suffer from an RC delay due to the capacitive coupling of the floating nanowires. It is important that all the word lines except the selected line are grounded to prevent the existence of sneak currents in the memristive crossbar, which would disturb reading the correct current at the bit line.

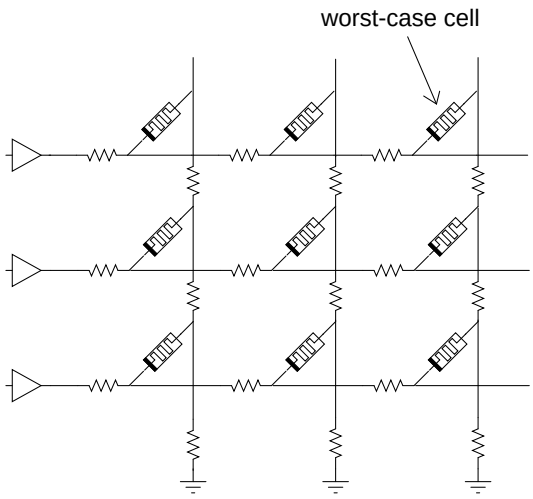

Figure 7.10: Effect of nanowire resistance in accessing memristors in a crossbar. Here the top-right memristor is accessed with the least voltage since it is the furthest from the drivers.

However, even with no sneak current paths, not all of the memristors in the crossbar are accessed with equal programming or read voltage. The reason to this is the nonzero resistance of the nanowires. Its effect on the write and read operations is discussed in the following subsection.

7.3.3 Nanowire resistance

The worst-case selected memristor under the write and read operations is the one that sees the smallest voltage across it when accessed [68]. Voltage reduction in selected memristors results from the voltage division due to the nonzero resistance of the nanowires. In Figure 7.10 the worst-case selected memristor is depicted; it is the device at the rightmost junction of a selected word line, when the drivers driving the horizontal nanowires are assumed to be located at the left ends of the nanowires.

During the write operation, the worst-case selected memristor sees the smallest voltage across it when all the other memristors on the word line are in the on-state. This is because the voltage drop along the word line is maximized by the large current flow.

As reported in [68], the fidelity of the read operation depends on the selected memristor's state, on the specific device parameters, and on the states of the other memristors in the crossbar. It was found that if the selected memristor is in the on-state, the read operation performs the worst when all the other memristors on the selected word line are in the off-state and all the other memristors in the crossbar are in the on-state. On the other

hand, if the selected memristor is in the off-state, the worst case happens when all the other memristors in the crossbar are in the on-state, as is the case with the write operation.

The maximum size of the memristive crossbar depends on the performance of the worst-case cell. The conclusion given in [68] is that to maximize the memristive crossbar size, one needs to minimize its power consumption. Thus the on-state and off-state resistance values of the memristors must be scaled up, while simultaneously keeping the on/off conductance ratio sufficiently large. However, this may degrade the write and read speeds of the circuit. Moreover, only uniformly distributed binary data should be stored into the memory in order to improve the reliability of the read operation [111].

Chapter 8

Memristor Applications

This chapter presents various applications of memristive computing. Sections 8.1 and 8.2 discuss digital memristive memory and reconfigurable logic, which are some of the most commercially viable applications described in this chapter. Section 8.3 presents how the memristive stateful logic operations of Chapter 6 can be parallelized to implement logical vector operations within memristive crossbars. Section 8.4 describes two memristive chaotic circuits, and Sections 8.5 and 8.6 conclude this chapter with a discussion on memristive neural and cellular nanoscale networks.

8.1 Digital memory

The prospect of fabricating highly dense nonvolatile digital memories is the main driving force of memristor technology [105]. A few different design candidates for a memristive implementation of a Resistive Random Access Memory (RRAM) have been proposed. Probably the simplest to implement is the *1T1R* architecture [14], where a memory cell consists of a CMOS transistor and a memristor which are connected in series. The transistor is used to select the memory cell while the memristor is used to store one or multiple bits of information. Since this architecture requires a transistor for each memory cell, it cannot provide as high a memory density as the demultiplexer [23] or CMOL [101, 103, 105] architectures, whose memory cells consist of single memristors.

In the CMOL RRAM architecture, the CMOS layer is used for coding, decoding, line driving, sensing and input/output functions [105]. The CMOS cell in this architecture has the simplest possible structure and consists of two pass transistors. Since demultiplexers have architectural challenges such as the need for nonlinear devices for the microwire-nanowire interfacing, it seems likely that the CMOL RRAM will overcome the demultiplexer design.

To make a CMOL-type memory defect tolerant, memory array reconfigu-

ration — bad bit exclusion — together with error correction code techniques have been proposed [101]. However, when using a computationally affordable *repair most* reconfiguration protocol complemented with Hamming-code error correction, only a defect rate of the order of 0.1% can be tolerated while attaining an order-of-magnitude advantage in density when compared to conventional CMOS memory architectures [101]. This defect rate can be increased to a more practicable 2% when the Hamming error-correction codes are replaced by the much more powerful BCH codes [103]. Such a defect rate has already been attained in [33], where the fabrication of a 1 kb memristive crossbar memory array was reported.

As noted in [105], it is natural to expect quick progress in CMOL type RRAM technology due to its very attractive density scaling properties.

8.2 Reconfigurable logic circuits

A memristive CMOL circuit can be used to implement reconfigurable Boolean logic circuits such as Field Programmable Gate Arrays (FPGAs) [105]. Memristive crossbars are advantageous in the implementation of FPGA–like circuits, since memristors can act as configuration-bit flip-flops and associated data-routing multiplexers, as proposed in [114]. Since more than 90% of the area in contemporary FPGAs is consumed by the SRAM-based configuration bits [105], a memristive implementation can yield much higher logic gate density than is available in a pure CMOS implementation. Another advantage in a memristive CMOL implementation is that since the memristors are non-volatile, the FPGA retains its state when unpowered.

From the fabrication point of view, the requirements for the memristors in an FPGA implementation are rather relaxed, as typically only a small fraction of the devices are in ON state, and their states need not be changed rapidly [105]. With appropriate defect-finding and control circuitry, the redundant data paths in the memristive crossbar allow for highly defect-tolerant operation. Indeed, if a memristor or a nanowire does not function properly, it can be bypassed [114] by choosing an alternative route. The first fabricated CMOL-type memristive reconfigurable logic circuit was reported in [114].

8.3 Parallel stateful logic

Parallel stateful logic, as proposed by our research group in [63], enables the implemention of bitwise vector operations on the columns and rows of a memristive crossbar. As described in Section 7.3.1, rectifying memristors are used to prevent the half-select problem and stray currents to arise within

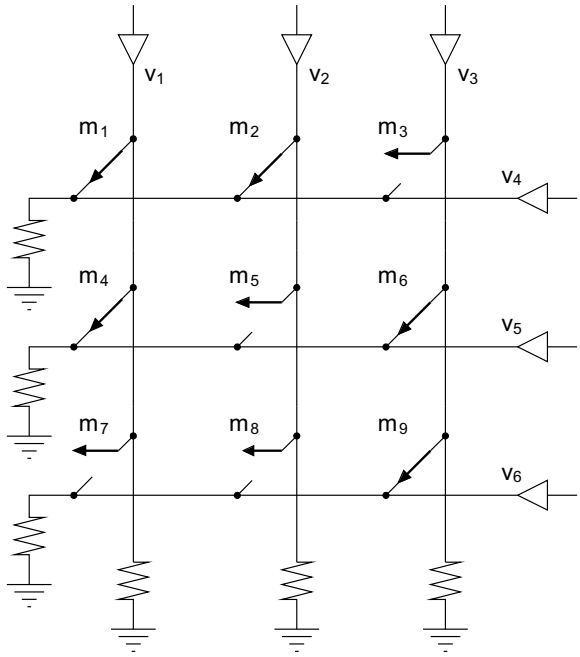

Figure 8.1: Parallel stateful memristor logic within a crossbar.

the crossbar. Figure 8.1 illustrates a parallel implementation of the stateful logic circuit presented in Figure 6.1.

Parallel stateful logic could be useful for example in image processing, where local filtering operations are applied simultanously on the pixel matrix. Another possible application is the emulation of binary neural network -type processing within a memory, such as the Willshaw associative memory model described in Section 9.3. For example, it is straighforward to implement some basic operations of *hyperdimensional arithmetic* [39] with parallel stateful logic. Hyperdimensional arithmetic is a cognitive computing method which allows to create data structures within an associative memory.

8.3.1 Column-wise operations

For columnwise stateful logic operations the horizontal nanowire drivers (v_4, v_5, and v_6 in Figure 8.1) are set to high impedance, as are the vertical drivers of those nanowires not participating in the stateful logic operation. Otherwise the implication and converse nonimplication operations are performed

as described in Chapter 6. For example, to obtain in parallel

$$\begin{cases} m_3 := m_2 \to m_3 \\ m_6 := m_5 \to m_6 \\ m_9 := m_8 \to m_9 \end{cases}$$

the nanowires should be driven to $v_2 = v_{\text{cond}}$, and $v_3 = v_{\text{set}}$, while the drivers of the horizontal nanowires and v_1 should be set to high impedance, $v_1 = v_4 = v_5 = v_6 = \text{HZ}$. The rectification prevents the current flowing through m_1 in the "wrong" direction. If in this example the memristors were not rectifying, a stray current path would be formed through $m_2 \rightsquigarrow m_1 \rightsquigarrow m_4$, disturbing the implication operation $m_6 := m_5 \to m_6$.

8.3.2 Row-wise operations

Analogously to the above, to implement row-wise implication and converse nonimplication operations, all the vertical drivers (v_1, v_2, and v_3 in Figure 8.1) and the horizontal drivers of the not participating nanowires are set to high impedance. However, in row-wise operations it is necessary to reverse the polarities of the conditional and programming voltages, assuming that the ground voltage equals 0 V.

For example, consider the parallel implementation of the implication operations

$$\begin{cases} m_7 := m_4 \to m_7 \\ m_8 := m_5 \to m_8 \\ m_9 := m_6 \to m_9 \end{cases}$$

in the crossbar circuit of Figure 8.1. These operations are performed by setting $v_1 = v_2 = v_3 = v_4 = \text{HZ}$, $v_5 = -v_{\text{cond}}$, and $v_6 = -v_{\text{set}}$. To justify these choices of conditional and programming voltages, consider first the operation $m_7 := m_4 \to m_7$. Since $m_4 = 1$ and the series resistance satisfies $R_0 \gg R_{\text{ON}}$, the voltage on the leftmost vertical nanowire is pulled close to $-v_{\text{cond}}$. It follows that the voltage across m_7 approximately equals $v_{\text{set}} - v_{\text{cond}} < V^T$, and consequently m_7 remains in the off-state. This is in accordance with the truth table of material implication presented in Table 6.1. On the other hand, since $m_5 = 0$ the voltage on the middle vertical nanowire is close to 0. Therefore the voltage across m_8 approximately equals v_{set}, thus m_8 is programmed to the on-state.

Remark 39. *In the parallel operations described above, computations are performed bitwise over all bits in the chosen rows or columns. In practice it is necessary to be able to operate on selected bits in vectors while keeping other bits unchanged. This can be achieved by driving the rows or columns corresponding to the nonparticipating bits to a voltage which depends on the*

operation being performed. For example for columnwise material implication these rows are driven to v_{cond}, while for columnwise converse nonimplication they are driven to ground voltage.

8.3.3 Example: Parallelized synthesis of a Boolean function

In Section 6.5 it was shown that with multi-input implication logic using complementary representation of variables, up to $2^{n-1}+1$ computational steps were required to synthesize an n-input Boolean function. To demonstrate the power of the multi-output converse nonimplication operation and parallelized stateful logic, in the following it is shown that the corresponding upper limit of operations is linear in n when parallel stateful operations within a memristive crossbar are used for the synthesis. Naturally, in return for the exponential speed-up, the amount of memristors required in this method is exponential with n.

Consider the evaluation of the Boolean function $S_3(p,q,r) = p+q+r \pmod 2$ within a 5×4 crossbar, where the inputs and the result of the function are represented by the states of the memristors on the bottom row of the crossbar. A natural description of such a crossbar is given as a 5×4 matrix, and initially the crossbar is assumed to be in the configuration

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ p & q & r & 0 \end{pmatrix}. \tag{8.1}$$

The function S_3 can be written in the form

$$\begin{aligned} S_3 \equiv & \mathrm{NOR}(\mathrm{AND}(p,q,\neg r), \mathrm{AND}(p,\neg q,r), \\ & \mathrm{AND}(\neg p,q,r), \mathrm{AND}(\neg p,\neg q,\neg r)), \end{aligned} \tag{8.2}$$

which provides a way to synthesize it using parallelized stateful logic. First the contents of the AND-clauses are synthesized by six multi-output converse nonimplication operations on chosen columns of the crossbar, while keeping the other columns unchanged. The first two computational steps are used to modify the column containing p as follows:

$$\mapsto \begin{pmatrix} \neg p & 1 & 1 & 0 \\ \neg p & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ p & q & r & 0 \end{pmatrix} \mapsto \begin{pmatrix} \neg p & 1 & 1 & 0 \\ \neg p & 1 & 1 & 0 \\ p & 1 & 1 & 0 \\ p & 1 & 1 & 0 \\ p & q & r & 0 \end{pmatrix}.$$

Another four multi-output operations on columns containing q and r are used to obtain

$$\overset{4}{\mapsto} \begin{pmatrix} \neg p & \neg q & r & 0 \\ \neg p & q & \neg r & 0 \\ p & \neg q & \neg r & 0 \\ p & q & r & 0 \\ p & q & r & 0 \end{pmatrix}.$$

Now, a multi-input implication operation on the first four rows followed by a multi-input implication operation on the rightmost column yield

$$\mapsto \begin{pmatrix} \neg p & \neg q & r & \mathrm{AND}(p, q, \neg r) \\ \neg p & q & \neg r & \mathrm{AND}(p, \neg q, r) \\ p & \neg q & \neg r & \mathrm{AND}(\neg p, q, r) \\ p & q & r & \mathrm{AND}(\neg p, \neg q, \neg r) \\ p & q & r & 0 \end{pmatrix}$$

$$\mapsto \begin{pmatrix} \neg p & \neg q & r & \mathrm{AND}(p, q, \neg r) \\ \neg p & q & \neg r & \mathrm{AND}(p, \neg q, r) \\ p & \neg q & \neg r & \mathrm{AND}(\neg p, q, r) \\ p & q & r & \mathrm{AND}(\neg p, \neg q, \neg r) \\ p & q & r & S_3(p, q, r) \end{pmatrix}.$$

On the whole, eight computational steps were required to synthesize S_3, and a constant number of auxiliary steps are required to initialize the crossbar in the form (8.1). In general, for an arbitrary n-input Boolean function $f : \{0,1\}^n \to \{0,1\}$, the synthesis of the contents of the AND-clauses takes $2n$ steps, while the rest of the computation is performed in a small constant number of steps. The number of memristors required in the crossbar is clearly at most of the order $n2^n$, depending on the synthesized function.

8.4 Chaotic circuits

Chaotic circuits taking advantage of the memristor dynamics have been presented in the literature for example in [8,24,31]. As noted in [76], memristive chaotic circuits have been proposed to be used in various applications, ranging from cryptography to medical purposes such as seizure detection. This section describes two chaotic memristor circuits, of which the latter is based on our publication [60]. In order to rigorously prove that these circuits truly are chaotic, very specific memristor models with no parameter variations are used. Certainly, one could obtain qualitatively similar dynamics with physical memristors, but so far no chaotic circuits using passive thin-film memristors have been demonstrated.

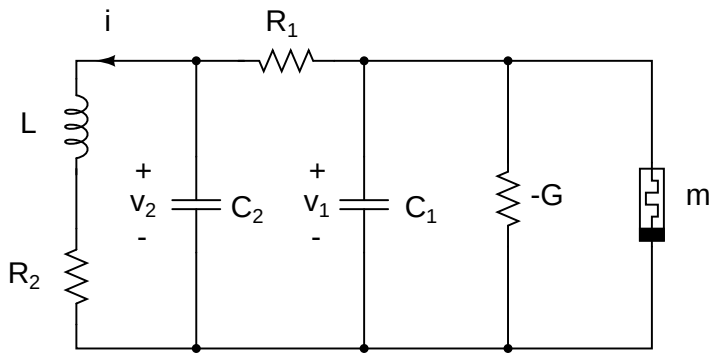

Figure 8.2: Memristive Chua's oscillator.

8.4.1 Memristive Chua's oscillator

The nonlinear *Chua's oscillator* [31] consisting of two resistors, two capacitors, an inductor, a negative resistor, and a flux-charge memristor is depicted in Figure 8.2. Its canonical configuration, without a memristor, is known to have a chaotic attractor [19], and using the following specific memristor model assures [31] that the memristive circuit has one also:

$$\begin{cases} i &= M(w)v = \begin{cases} av & \text{if } |w| < 1 \\ bv & \text{otherwise} \end{cases} \\ \dot{w} &= v \end{cases} \tag{8.3}$$

where a and b are positive constants, $M(w)$ is the memristance of the memristor, and the state variable w is allowed to take any real value. Since the conductance of this device is not a monotonic function of w when $a \neq b$, it is strictly speaking a flux-charge memristor or a memristive system, when the nomenclature of Chapter 2 is used, but not a memristor in terms of Definition 4.

The memristive Chua's oscillator satisfies the following set of differential equations:

$$\begin{cases} \dot{v_1} &= ((v_2 - v_1)/R_1 + Gv_1 - M(w)v_1)/C_1, \\ \dot{v_2} &= ((v_1 - v_2)/R_1 - i)/C_2, \\ \dot{i} &= (v_2 - R_2 i)/L, \\ \dot{w} &= v_1, \end{cases} \tag{8.4}$$

which, by changing and renaming of variables, can be simplified into

$$\begin{cases} \dot{x} &= \alpha(y - x + \xi x - M(w)x), \\ \dot{y} &= x - y + z, \\ \dot{z} &= -\beta y - \gamma z, \\ \dot{w} &= x. \end{cases} \tag{8.5}$$

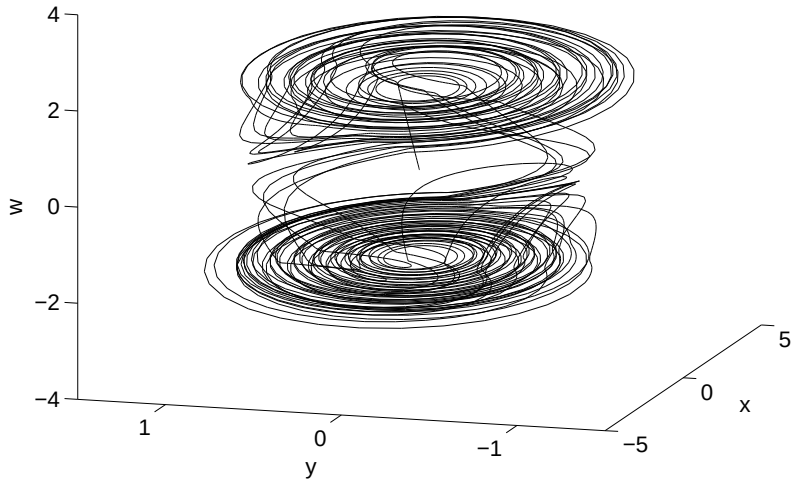

Figure 8.3: Chaotic attractor of the memristive Chua's oscillator.

By choosing the values of the parameters as suggested in [31], $\alpha = 10$, $\beta = 13$, $\gamma = 0.35$, $\xi = 1.5$, $a = 0.2$, and $b = 0.8$, one obtains a dynamical system with a chaotic attractor as shown in Figure 8.3.

The chaotic double scroll attractor resembles the one which is present in a Lorenz oscillator [73]. The Lorenz oscillator can be described by a set of three differential equations having two nonlinear terms. In comparison, Chua's oscillator is described by a set of four differential equations having a single nonlinear term.

8.4.2 Memristive logistic map

The logistic map $f : [0, 1] \to [0, 1]$, $L(x) = 4x(1 - x)$ is an archetypal example of a simple non-linear function having chaotic iterative behaviour. The iteration of L, $L^{(k)}(x) \equiv L(L(\ldots(L(x))))$, is unpredictable due to its sensitivity on initial conditions, it is indecomposable, and its periodic points form a dense subset of $[0, 1]$. Originally proposed in [75], the logistic map was designed to capture the dynamics of a animal population with reproduction and starvation. The following presents a memristive implementation of the logistic map, as originally described in our publication [60].

A memristive system suitable for computing the memristive logistic map

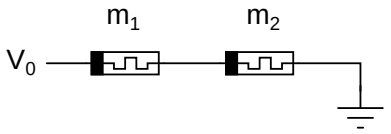

Figure 8.4: Memristors connected in series for self-terminating copying.

can be defined as follows:

$$\begin{cases} v &= \alpha r i \\ \dot{r} &= \begin{cases} \lambda v & \text{if } |v| > v_T \text{ and } 0 < r < 1 \\ 0 & \text{otherwise,} \end{cases} \end{cases} \tag{8.6}$$

where r is the state of the system, the parameters satisfy $\alpha > 0$ and $\lambda < 0$, and the constant $v_T > 0$ is the threshold voltage of the system. At a given state r the system acts as a linear resistor whose resistance is $R(m) = \alpha r$. Since $\lambda < 0$, the resistance of the system decreases (increases) if a sufficiently large positive (negative) voltage is set across it. Notice that in this specific model the smaller the state r, the more conductive the system is. This system is a memristor according to Definition 4, when $w = 1/r$ is chosen as its state variable.

Consider now two memristors, m_1 and m_2 connected in series as depicted in Figure 8.4, and assume that $r_1 > r_2$, or, written in another way, $R(m_1) > R(m_2)$. Setting the voltage $V_0 = 2v_T$ across this circuit now yields perfect self-terminating copying of the state r_2 to memristor m_1 as explained in Section 5.1.1.

Let now $V_0 = 2v_{T2}$ and let $v_{T1} = (k/(k+1))V_0$, where v_{Ti} denotes the threshold voltage of the ith memristor. Then the programming of m_1 stops when the voltage V across it equals

$$V = \frac{R(m_1)}{R(m_1) + R(m_2)} V_0 = \frac{k}{k+1} V_0. \tag{8.7}$$

This equality holds if and only if $R(m_1) = kR(m_2)$, or in other words, if and only if $r_1 = kr_2$. Thus this memristive circuit can be used to copy the state of m_2 to m_1 while simultaneously multiplying the result by k.

A memristive circuit which computes the logistic map is depicted in Figure 8.5. It consists of three memristors, three switches and an unity buffer. The voltages V_1 and V_2 are used to unconditionally program the memristors to state $r = 0$ or $r = 1$.

Let the threshold voltages satisfy

$$v_{T1} = (4/5)V_0, \quad v_{T2} = v_{T3} = (1/2)V_0, \tag{8.8}$$

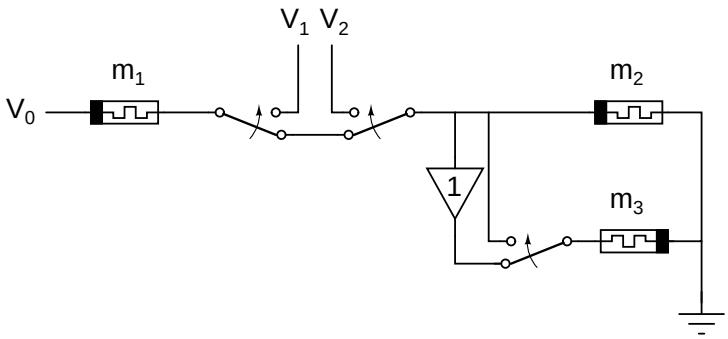

Figure 8.5: A memristive circuit for iterative computing of the logistic map.

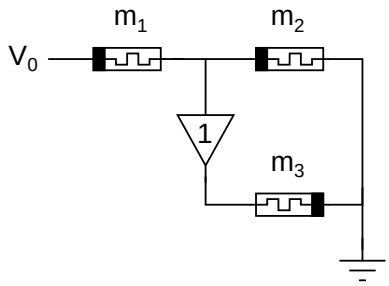

Figure 8.6: Simultaneously copying $r_1 = c$ as the state of m_2 and $1 - r_1 = 1 - c$ as the state of m_3.

and let the initial states of the memristor be $r_1 = c$, $r_2 = 1$, and $r_3 = 0$, where $c \in (0, 1)$ can be chosen arbitrarily. The circuit is operated by iterating the following four steps:

1. Let the switches be configured so that the circuit corresponds to Figure 8.6. Then, r_1 is copied as the state of m_2 while the state of m_3 is programmed into the reverse direction by an equal amount. As a result of this step, $r_1 = c$, $r_2 = c$, and $r_3 = 1 - c$.

2. Program m_1 into the high-resistance state $r_1 = 1$, while keeping the states of the memristors m_2 and m_3 unperturbed.

3. Configure the switches so that the circuit corresponds to Figure 8.7. Now the parallel resistance of the memristors m_2 and m_3 is copied as the resistance of m_1 and is simultanously multiplied by 4, since $v_{T1} = (4/5)V_0$. The parallel resistance value is

$$\begin{aligned} R(m_2|m_3) &= \frac{R(m_2)R(m_3)}{R(m_2) + R(m_3)} \\ &= \frac{\alpha^2 c(1-c)}{\alpha} = \alpha c(1-c). \end{aligned} \tag{8.9}$$

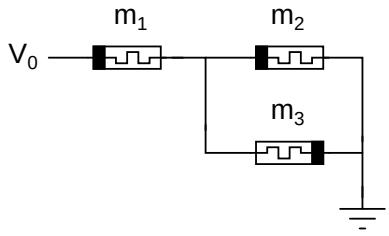

Figure 8.7: Self-terminating multiplication resulting in $r_1 = 4r_2r_3$ when $r_2 + r_3 = 1$.

Thus, after this step the state of m_1 equals $r_1 = 4c(1 - c)$.

4. Finally, program the memristors m_2 and m_3 back to their initial states $r_2 = 1$ and $r_3 = 0$.

In other words, the four operation steps above perform one iteration of the chaotic logistic map $L(x) = 4x(1-x)$, and store the value of the iteration as the resistance state of the memristor m_1. Now, repetitive iteration of these procedures results in a chaotic orbit of r_1.

8.5 Neuromorphic hardware

Neuromorphic architectures, which are inspired by the functionality and circuitry of the nervous system, are particularly well-suited to be implemented as CMOL circuits. In these architectures the *neuronal somas* are implemented by CMOS components, while the intercellular communication — the *axons* and the *dendrites* — and the *synapses* between neurons are realized using memristive nanowire crossbars [97]. Example 40 illustrates such an artificial neuron.

Example 40. *Consider the simple artificial neuron presented in Figure 8.8. An operational amplifier is used to create a virtual ground to one end of the memristors, while the other ends are driven with voltages V_{in1}, V_{in2}, and V_{in3}. Although this neuron has only three inputs, in general the number of inputs can be arbitrary. The currents flowing through the memristors are summed, and the total amount of current is represented by the voltage V_x which controls the voltage dependent voltage source. The output of this neuron is represented by the voltage V_{out}.*

If each of the memristors has an instantaneously linear I-V behavior

$$i_j = G(w_j) \cdot v, \tag{8.10}$$

where w_j denotes the state variable of the memristor m_j, and $G(w)$ is the instantaneous conductance of the device — also known as the memductance

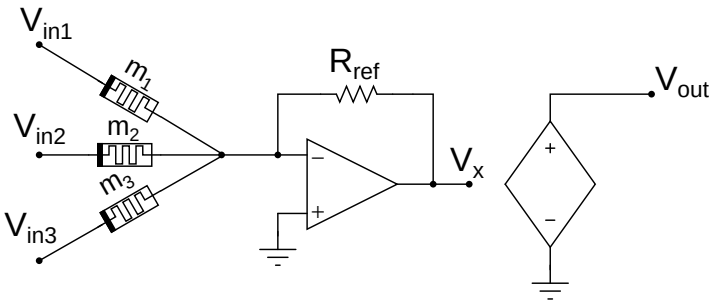

Figure 8.8: A simple CMOS artifical neuron with memristive synaptic inputs.

— *it follows that*

$$V_x \propto \sum_j G(w_j) \cdot V_{in,j}. \tag{8.11}$$

Thus the input voltages are multiplied by the memductances, and the sum of these products is represented by the voltage V_x*. In other words* V_x *represents the inner product of the vectors* $\mathbf{V}_{in} = (V_{in1}, V_{in2}, V_{in3})$ *and* $\mathbf{G} = (G(w_1), G(w_2), G(w_3))$*.*

The output V_{out} *of the neuron is typically a sigmoidal function of* V_x*. On the whole, the operation of this artificial neuron corresponds to the definition of the so-called perceptron model [74], except that only positive synaptical weights* $G(w)$ *are considered here, as conductance is always a positive quantity. In a neural network architecture, the input voltages would correspond to the outputs of other artificial neurons. A crucial point here is that the memristors realize a synaptic multiplication operation, which would take much more area if realized by CMOS circuitry. It depends on the application of the neural network whether or not continuous weights are needed, or if multi-level or binary synapses suffice.*

Remark 41. *An important special case of the synaptic multiplication presented in* (8.10) *is achieved, when the each of the voltages* V_{in} *across the memristors belong to the binary set* $\{0, V_R\}$*, where* V_R *is a read voltage. This corresponds to the case of binary neurons — for example spiking neurons — that have only two output values. Then* (8.10) *holds regardless of the* I*-*V *behavior of the memristor, since the read current is either zero when* $V_{in} = 0$*, or can be written in the form* $i_j = (i_j/V_R) \cdot V_R$*, where* $i_j/V_R \equiv G(w_j)$*.*

As a result, each binary neuron computes inner product $\mathbf{V}_{in} \circ \mathbf{G}$ *of the binary activation vector* $\mathbf{V}_{in}$ *and the generally continuously valued synaptical weight vector* $\mathbf{G}$*. This specific type of inner product will be crucial in the design of the memristive associative memories discussed in Chapter 9.*

In a CMOL architecture, the memristive synaptic connections can be

implemented physically above the CMOS neurons thus freeing up die area. In principle, synaptic densities comparable to biological nervous systems could be attained. For example, a memristor feature size of $F = 50$ nm yields a synaptic density of 10^{10} memristive synapses per square centimeter, which is comparable to that of the human cortex [96]. Moreover, each CMOS neuron can be connected to thousands of other neurons, thus enabling similar connectivity as that found in biological neural networks. Similar synaptic density could be attained by using other technologies such as floating gates, but there the synaptic connections would consume area from the neurons at the CMOS layer, if 3D integration were not used.

CMOL-type neuromorphic architectures such as the one described above are called *CrossNets*, and they were first proposed by Likharev et al. in [69]. Topologies of CrossNets are usually divided into two categories: the feed-forward *flossbar* topology and the recurrent *inbar* topology. Examples of the two topologies are depicted in Figure 8.9. An advantage of the inbar topology over the flossbar topology is that it allows greater flexibility to the size and form of CMOS somas due to their greater physical separation [108]. The flossbar topology can readily be used to implement a multilevel perceptron, while the inbar topology can be used to implement a Hopfield-type neural network, as is briefly discussed in Chapter 9.

Traditionally in CrossNets the synaptic two-terminal devices are assumed to be bistable. Therefore analog synaptic weights must be emulated by two crossbars of size $k \times k$ which yields $2k^2 + 1$ different conductance levels, as one of the crossbars is used for positive synaptic connections and the other for negative connections [109].

There are two different approaches to program the synaptic weights of a CrossNet. One is to learn the weights in a separate homomorphic network, which can be implemented for example in software. Once the learning phase is completed, the weights are copied into the CrossNet. This approach may be impractical for large networks due to the substantial amount of required computing time, and due to the fact that such learning does not easily provide fault-tolerance against device failures in the memristive crossbar. Another approach is to learn the weights online, for example through the *backpropagation algorithm* used in a multilevel perceptron, or the *outer product learning rule* used in the Hopfield network. These different learning approaches and their potential merits are thoroughly discussed in [108].

8.6 Cellular Neural Networks

Originally defined in 1988 by Chua and Yang [21], the Cellular Neural Networks constitute a class of information processing systems, which are made of massive aggregates of regularly spaced circuit clones, called *cells* that

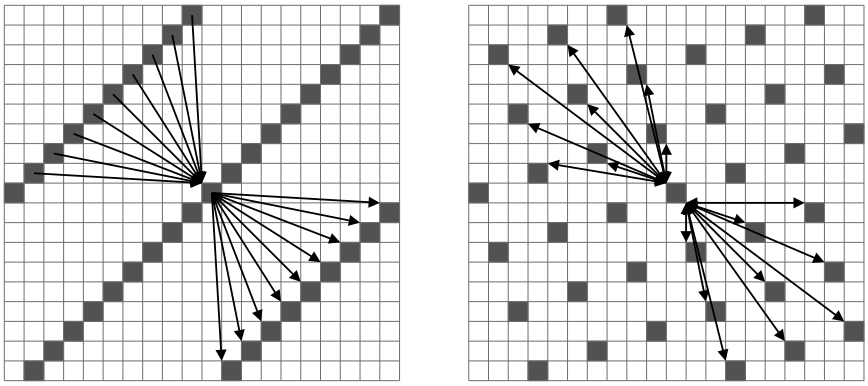

Figure 8.9: Examples of CrossNet topologies. Left: flossbar, right: inbar. Possible input and output connections of the central somas are indicated by arrows. In principle, the connections in either topology can be realized as uni- or bidirectional, but typically the flossbar topology is feedforward, and the inbar topology is recurrent, as illustrated. Note that the synaptic devices corresponding to the connections are not depicted.

communicate with each other only through their nearest neighbours. The CNN paradigm differs from conventional neural networks by allowing arbitrary nonlinearities for the cells and the intercellular communication. For example, a CNN cell may match the pattern of its neighbours' outputs with a predefined pattern and output the result as a logical value. On the other hand, the CNN paradigm also differs from the conventional notion of a cellular automaton, since a CNN in general operates in real time, with analog values. Moreover, in general the functionality — or *the rule* — of the cells is allowed to vary from cell to cell and change during computation. A conceptual picture of a CNN processor is depicted in Figure 8.10. Here each cell, depicted by a square, is connected to its nearest neighbours.

In a memristive implementation of a CNN processor, the intercellular connections are implemented by memristive crossbars. This allows for neighborhoods whose sizes are restricted only by the corresponding sizes of the connectivity domains of the nanowires, as explained in Section 7.2.2. The motivation for using memristive crossbars is the same than with the more conventional neural networks discussed in Section 8.5: the area consuming inter-cellular communication network can be lifted from the CMOS layer, thus allowing for a larger number of cells within the same die are. The memristive CNN cell designs presented in this section were originally proposed in our publications [54, 55, 59].

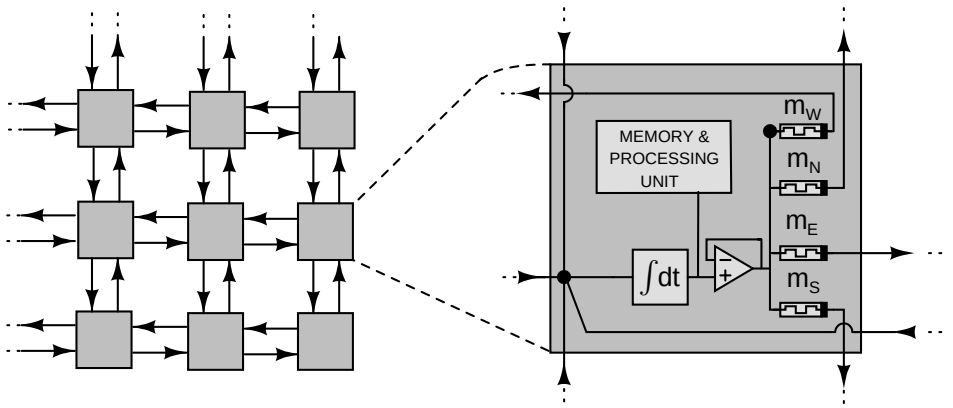

Figure 8.10: A conceptual CNN processor with memristors for neighbourhood connections.

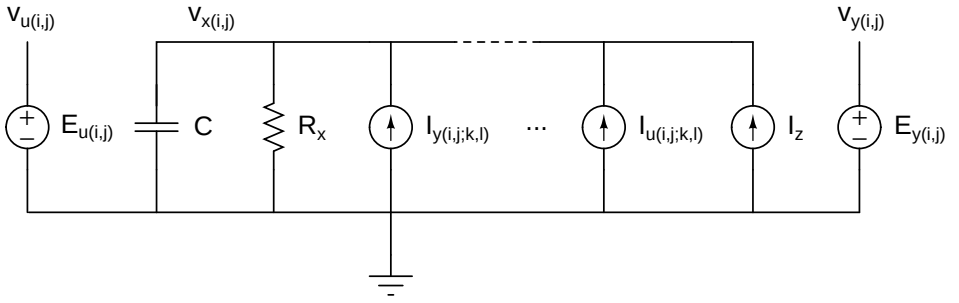

Figure 8.11: The standard CNN cell, modified from [20].

8.6.1 Standard memristive CNN

In the original paper defining cellular neural networks [21], a two-dimensional network of $M \times N$ cells arranged in M rows and N columns was investigated as a practically significant case. All cells in this CNN are identical to the one depicted in Figure 8.11. In fact, this network architecture has become canonical in the CNN literature, and is referred as the *standard CNN* [20].

Definition 42 (Modified from [20])**.** Mathematically, a standard CNN is a $M \times N$ rectangular array of cells $C(i,j)$ located at coordinates (i,j), $i = 1, 2, \ldots, M$, $j = 1, 2 \ldots, N$. Each cell $C(i,j)$ satisfies the

1. State equation

$$\dot{x}_{i,j} = -ax_{i,j} + \sum_{(k,l)\in N_{i,j}} A(i,j;k,l)y_{k,l} + \sum_{(k,l)\in N_{i,j}} B(i,j;k,l)u_{k,l} + z_{i,j}, \tag{8.12}$$

where $a \geq 0$ is a constant and $x_{i,j}, y_{i,j}, u_{i,j}, z_{i,j} \in \mathbb{R}$ are called the *state*, *output*, *input*, and *threshold* of the cell $C(i,j)$, respectively. The coefficient matrices $A(i,j;k,l)$ and $B(i,j;k,l)$ are called the A- and B-*templates*, respectively, and $N(i,j)$ is a collection of coordinates called the *neighborhood* of cell $C(i,j)$.

2. Output equation

$$y_{i,j} = g(x_{i,j}) = \frac{1}{2}|x_{i,j}+1| - \frac{1}{2}|x_{i,j}-1|, \tag{8.13}$$

which is essentially a truncated identity function with the minimum and maximum values of -1 and 1, respectively. It is called the *standard nonlinearity*.

3. Initial conditions $x_{i,j}(0)$ for all $i \in [1,M]$ and $j \in [1,N]$.

Figure 8.12 shows a standard CNN cell using memristors for the neighborhood connections. The cell's input is V_U. Although only five neighbourhood connections per template — the A template is drawn in black and the B template in gray — are drawn, in general the neighbourhoods could be much larger. When the cell is configured for computation, the switches $s8a$ and $s8b$ are opened and closed, respectively, and the switch $s7$ is closed to allow integration of the input currents. Notice that the cell's output voltage CNN_OUT is bounded by the supply voltages of the operational amplifier, and hence approximates the standard nonlinearity. There is no additional node for the cell state as it is also represented by CNN_OUT.

The cell's output is buffered by inverting and noninverting unity gain buffers to represent both positive and negative neighbourhood connections. The current $\pm I_{mem}(V_{\text{CNN\_OUT}})$ flowing through a neighbourhood connection in the case of the generic analog memristor model equals

$$\pm I_{mem} = \pm \alpha w \sinh(\beta V_{\text{CNN\_OUT}}), \tag{8.14}$$

where $w \in [0,1]$ is the state of the corresponding memristor. The linearity of the I-V relationship of the memristor depends on the parameters α and β; a perfectly linear relationship would correspond to the standard CNN of Definition 42.

Template programming

The templates are programmed with the continuous monitoring method described in Section 4.2.2. For example, the following procedure is used to program the A template's negative self-feedback memristor, which is denoted by m in Figure 8.12. The switches that are not mentioned are assumed to be open, and GND = 0.

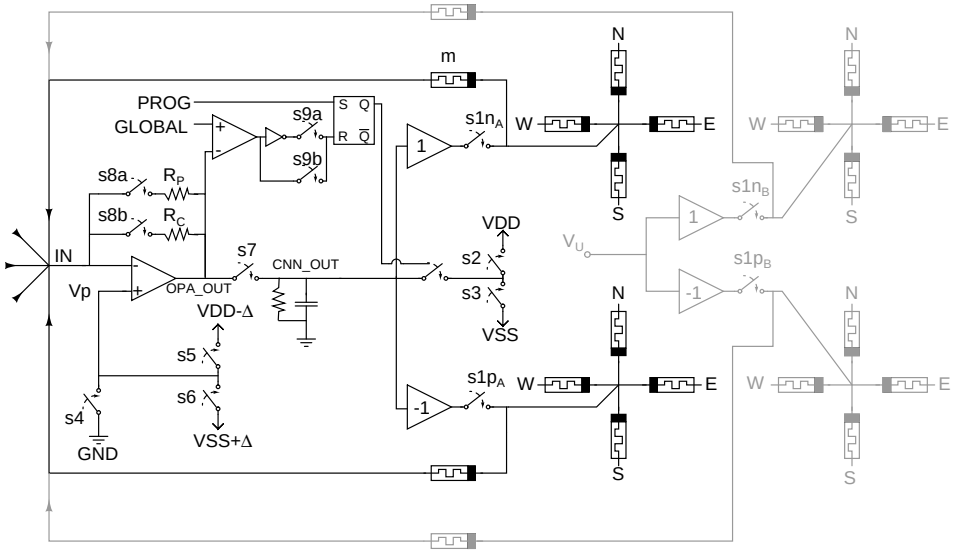

Figure 8.12: A standard memristive CNN cell, which is simplified so that the state x equals the output y, that uses the continuous monitoring method of Section 4.2.2 to program the memristors. The circuitry for the computing part and the A template connections are depicted in black, whereas the input node V_U and the B template connections are depicted in gray. Both the templates have self-feedback connections.

1. First m is programmed to a non-conductive state. For this, the control voltage PROG is set to logical 1 in order to close the switch connected to the SR-latch, and the switches $s1n_A$, $s3$, $s5$, $s8a$, and $s9b$ are closed. The control voltage GLOBAL is set to $-(V_{DD} - V_{SS} - \Delta)$. This configuration is maintained for a sufficiently long time to allow the state of the memristor to reach $w \approx 0$ regardless of its initial value.

2. To program the memristor into a predefined state, the following configuration is used. The control voltage GLOBAL is set to a value $-V_{ref}$, where $I = V_{ref}/R_p$ is the current through the memristor at its desired state, when the voltage across it is $V_{DD} - V_{SS} - \Delta$. The switch s_3 is opened, and the switches s_2 and s_6 are closed. Now the output Q of the SR-latch flips from the logical 0 to 1 as the memristor's state reaches the desired value.

Remark 43. *Memristors are programmed at a constant voltage* $\pm(V_{DD} - V_{SS} - \Delta)$. *Thus the magnitude of the auxiliary voltage* $\Delta > 0$, *and the magnitudes of the supply voltages determine the programming rate of the memristors.*

Remark 44. *A lookup table may be required to map a desired template value to the control voltage PROG. This is due to the* $I - V$ *nonlinearity of memristors, as the voltages used in computing are lower than the ones used for programming.*

A sketch of the memristive template layer is depicted in Figure 8.13. Local memristive memory, not shown in the cell circuit of Figure 8.12, is added to enable local memory and possible stateful logic computations. The advantage of using such a local memory and logic unit is further discussed in Section 8.6.3.

Unlike in transistor-based CNN realizations that convey template patterns in parallel as global voltages, different template matrix entries need to be programmed sequentially. Furthermore, even the programming of a specific template entry requires a sequence of operations as is described next.

Suppose that the neighbourhoods of the cells consist only of nearest neighbours, and consider the programming of the memristors corresponding to the positive northern neighborhood connections of the A template. All of these connections cannot be programmed simultaneously, since some of the cells must drive the programming voltage V_{DD} while some other cells need to compare the memristor's current with the GLOBAL signal as discussed above. It is not enough to perform the programming in only two steps, since this would lead to unwanted programming of the southern connections. Moreover, there needs to be also horizontal separation of the cells participating in the programming to avoid unwanted programming of the

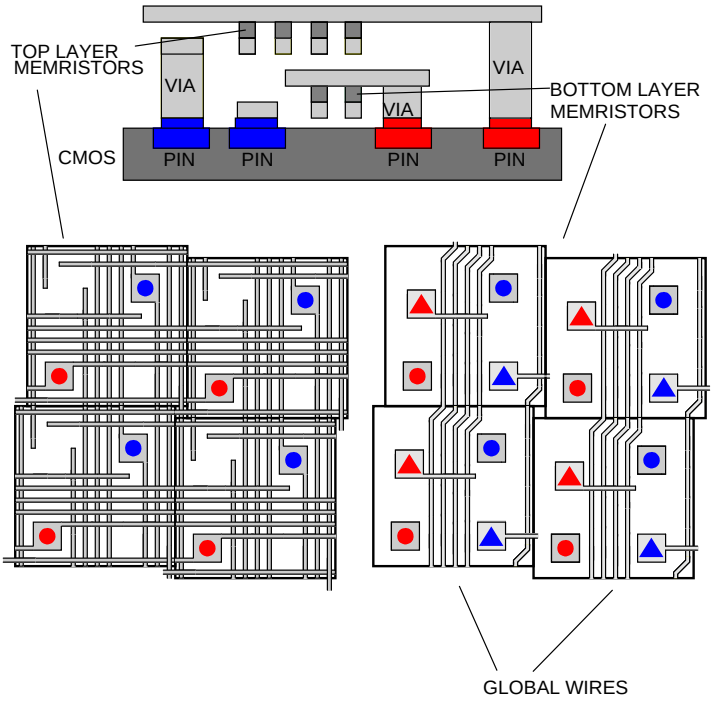

Figure 8.13: Memristive crossbar layers for one template layer (left) and local memory and logic (right). Each CNN cell has pins corresponding to blue and red circles and triangles. The global wires are driven by signals that allow programming and logic of the memristors forming the local memory.

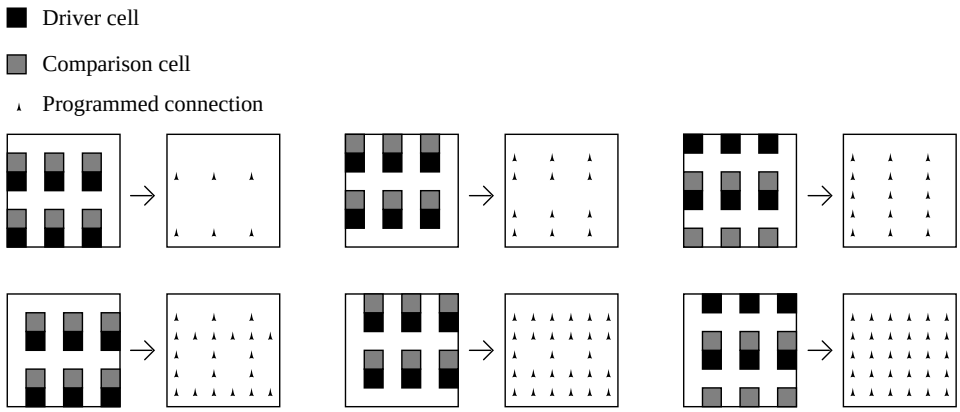

Figure 8.14: Sequential programming of the northern connections of the A template.

horizontal connections. As a result, six programming steps, as depicted in Figure 8.14, are required in order to program a single entry in all of the template matrices. Even more steps are required if the neighbourhoods of the cells are larger — for example, if they contain second nearest neighbours.

To cope with this inherent slowness of the template programming, the continuous monitoring method of Section 4.2.2 is used. Cyclical programming would yield a simpler cell structure with the expense of a longer programming time. A possible remedy to the long programming times of the templates is the vertical stacking of memristive crossbars described in Section 7.2.5, as it would allow for a large number of preprogrammed templates to be used in computation.

8.6.2 Memristive binary CNN

Restricting the computation of a CNN to only binary values obviously simplifies the cell circuitry and allows signal restoration. In the following, a binary — or black-and-white — CNN with bistable memristors for neighbourhood connections and local memory and logic [54] is presented. The state equation of a binary CNN cell is the OR function of its neighbours' outputs, which can be seen as a restriction of the linear combination (8.12) to binary values.

The binary CNN cell shown in Figure 8.15 consists of the memristive neighbourhood connections, the work memristors for local memory and stateful logic, a keeper subcircuit previously discussed in Section 6.1 to facilitate the stateful logic operations, two sets of access circuits controlled by row and column decoding signals, and a driver subcircuit which drives the cell output Y to its neighbours, when activated by the RUN signal. The

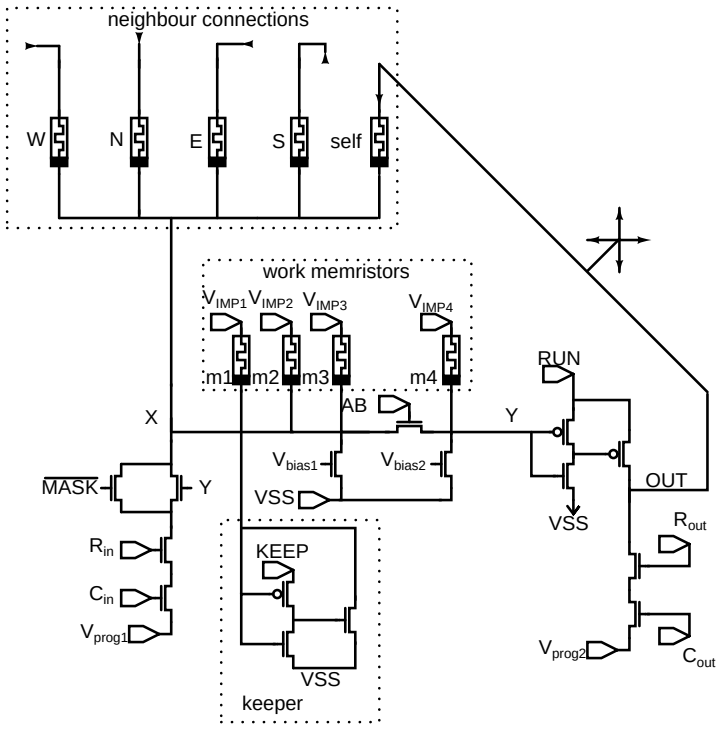

Figure 8.15: Structure of a memristive binary CNN.

control signal AB can be used to separate the state node X from the output node Y. For this implementation, rectifying memristors should be used in order to prevent stray-currents from arising in the nanowire crossbars.

This cell architecture is suitable for processing binary templates which do not require threshold logic. It allows for asynchronous computation, and a very dense implementation of a CNN processor as a cell consists only of 15 CMOS transistors. Also, as was the case with the memristive standard CNN, space-dependent templates are readily available.

In the following, the programming and the local memory of this CNN cell are described. A binary CNN application is discussed in Section 8.6.3.

Programming the neighbourhood connections

The memristors used for neighbour connections (N, S, W, E and self-feedback connections are drawn) connect the output nodes OUT to the state nodes X.

Each cell has two sets of row and column decoding signals, which are denoted by R_{in}, C_{in}, R_{out}, and C_{out}. The control signal $\overline{\text{MASK}}$ can be used to make the programming conditional on the output Y of the CNN cell. The control voltages V_{prog1} and V_{prog2} decide the programming voltage and

polarity. For example, to program the self-feedback memristor to the on-state, all of the row and column decoding signals of the cell and $\overline{\text{MASK}}$ are set high, and the programming voltages are chosen such that the voltage across the memristor $V_{prog2} - V_{prog1}$ is sufficiently large. As discussed in Section 8.6.1, the template programming must be performed sequentially.

If the control signal $\overline{\text{MASK}}$ is low, the voltage at the output node Y affects the writing of template values: if Y is low, the programming voltage V_{prog1} is not conveyed to the state node X, preventing the programming. On the other hand, if voltage at Y is high, template programming is possible. This feature can be used to create the so-called *transient mask*: no incoming connections are programmed into the ON state if Y is low. The transient mask is a special case of space dependent template programming whose operation depends on the intracellular memory.

Intracellular memory

Figure 8.15 also shows four local memristors that are used for local memory and stateful logic. This local memory unit can be used to combine results from previous computations, and to affect the selection of the subsequent templates. A large local memory can be afforded due to memristors' small footprint. It should be noted that the local memristors do not get programmed unintentionally when the template memristors are programmed, because the control voltages $V_{IMP1}, \ldots, V_{IMP4}$ are kept in the middle of the voltage range in order to keep the voltage across the local memory memristors below a programming threshold.

8.6.3 CNN Universal Machine: computing with waves

The memristive implementations of the standard and binary CNNs in the previous subsections are instances of the *CNN Universal Machine* [20, 91], which is a general analog-and-logic array computer. The most important features of the universal machine are the ability to apply several space-dependent templates simultaneously and sequentially, and the availability of intracellular memory and logic. The instructions of the universal machine correspond to spatial-temporal waves traversing the CNN array [91, 92]. These waves can interfere, diffract and annihilate with other waves and the input *flows* of the CNN.

Wave-type logic computing

In the following, a CNN wave-type algorithm for evaluating an arbitrary Boolean function on a specific input is presented. Here only the general ideas are described; a more formal treatment can be found in our publications [61,

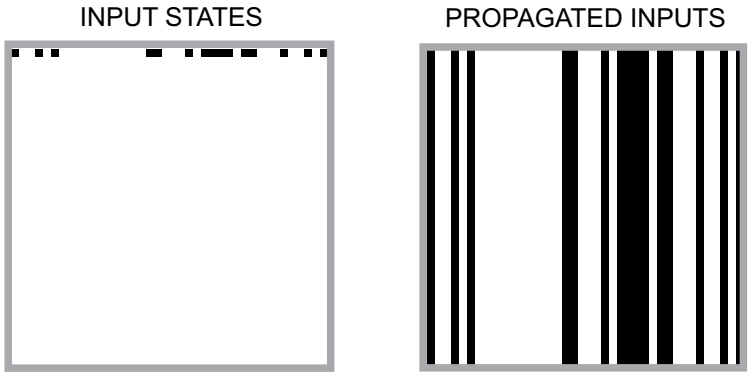

Figure 8.16: Initializing the evaluation of a Boolean function $f : B^n \to B$, where $n = 40$ and the number of disjunctive clauses $t = 40$ in the conjunctive normal form. Left inset: Initially, the values of the variables of f are represented by the states on the northernmost row of the CNN. Right inset: The values are propagated over the whole processor.

67]. The purpose of this subsection is to provide a simple example of wave-type computing — the reader may notice some resemblance with this method to the one discussed in Section 8.3 on parallel implication logic.

A standard CNN model is assumed, where each of the cells has at least one bit of local memory. The self-feedback constant a is fixed to $a = 0$ in all of the processing steps described in the following. The input values of the Boolean function are represented by the binary states of the northernmost row of the CNN — a black pixel stands for 1 and a white pixel for 0. The first rule assigned to the CNN corresponds to propagating these values across the whole processor, and it can be written in the binary case as an A template as

$$A_1 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad z = -0.5.$$

Here the centermost entry of A_1 corresponds to the cell itself, and all the rest entries correspond to the cells at that position relative to the cell. This first step of the computation is illustrated in Figure 8.16.

Let the Boolean function f be given in the conjunctive normal form. In the CNN representation of f, each of the disjunctive clauses corresponds to a row of the CNN, and each cell on a row corresponds to a specific input variable of the Boolean function, as depicted in the left inset of Figure 8.17. Say, for example, that the second disjunctive clause of f contains the non-inverted variable p_1, and that the fourth disjunctive clause of f contains the inverted variable p_3, that is,

$$f \equiv (\ldots) \wedge (p_1 \vee \ldots) \wedge (\ldots) \wedge (\ldots \vee \neg p_3 \vee \ldots) \wedge \ldots .$$

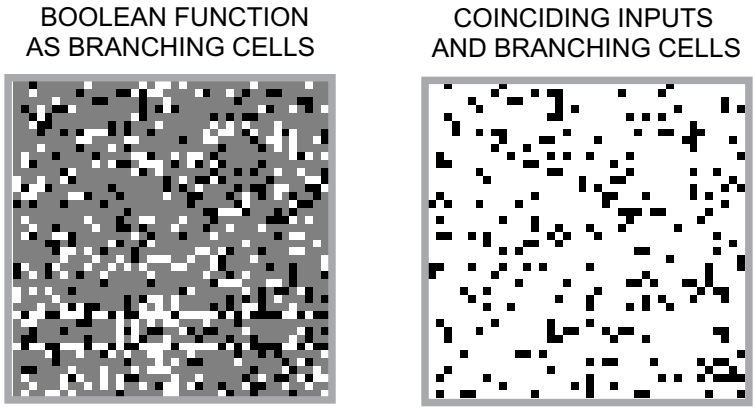

Figure 8.17: Representing a Boolean function by branching cells. Left inset: The noninverted branching cells are drawn in black while the inverted branching cells are drawn in white. Nonbranching cells are drawn in gray. Right inset: Each cell whose current state corresponds to the type of the branching cell is set to 1, while the states of all the other cells are set to 0.

Then, using the intracellular memory, the first cell on the second row is marked as a *noninverted branching cell*, and the third cell on the fourth row is marked as an *inverted branching cell*. Next the previously propagated input wave is compared with the branching cells. Each cell whose current state corresponds to the type of the branching cell is set to 1, while the states of all the other cells are set to 0, as illustrated in the right inset of Figure 8.17.

Next, a wave is propagated eastwards by the rule

$$A_2 = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad z = -0.5.$$

An illustration of this rule on a specific Boolean function and its input is depicted in the left inset of Figure 8.18. Ultimately, only the rows whose corresponding disjunctive clauses evaluate to zero are left white, while all the rest of the rows have a black cell at the eastern end of the row. Finally, the results of the evaluations of the disjunctive clauses are propagated south by the rule

$$A_3 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad z = -1.5,$$

as illustrated in the right inset of Figure 8.18. The result of the evaluation of the Boolean function can be read from the south-eastern corner of the

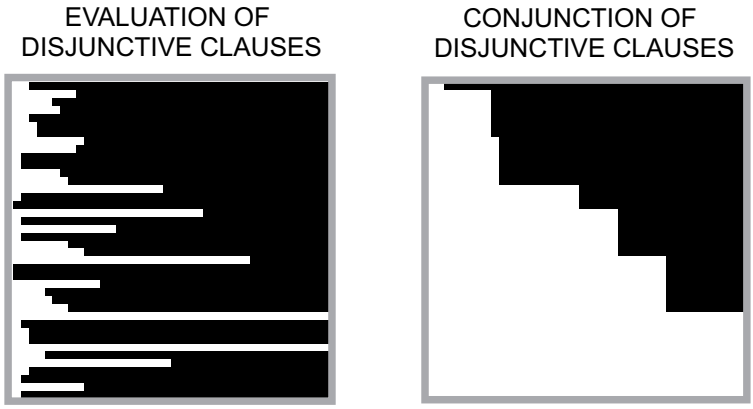

Figure 8.18: Evaluating the disjunctive clauses in parallel, and finally taking their conjunction for the result. Left inset: Eastward propagation for evaluating the disjunctive clauses. Right inset: Conjunction of the disjunctive clauses by the transient mask. The result of the evaluation is read from the south-eastern corner of the CNN, where white corresponds to 0 and black to 1. In this example, the result is 0.

CNN, in the example the result is 0. Indeed, a Boolean function f evaluates to zero if and only if at least one of its disjunctive clauses has value zero.

Remark 45.

- *The CNN can be divided into non-interfering subsections with the transient mask. This allows the computation of multiple Boolean functions simultaneously as is discussed in [61]. Such parallel computing is depicted in Figure 8.19, which shows measurements from the MIPA4k CNN processor.*
- *The utilization of gray-scale cells and local memory allows to compress the representation of the Boolean function as is described in [67]. Measurements from the CNN processor MIPA4k [88] configured for gray scale processing are shown in 8.20. For technical reasons, the binary values of the four three steps are inverted in this experiment.*
- *All the propagation rules used for the evaluation can be performed asynchronously.*

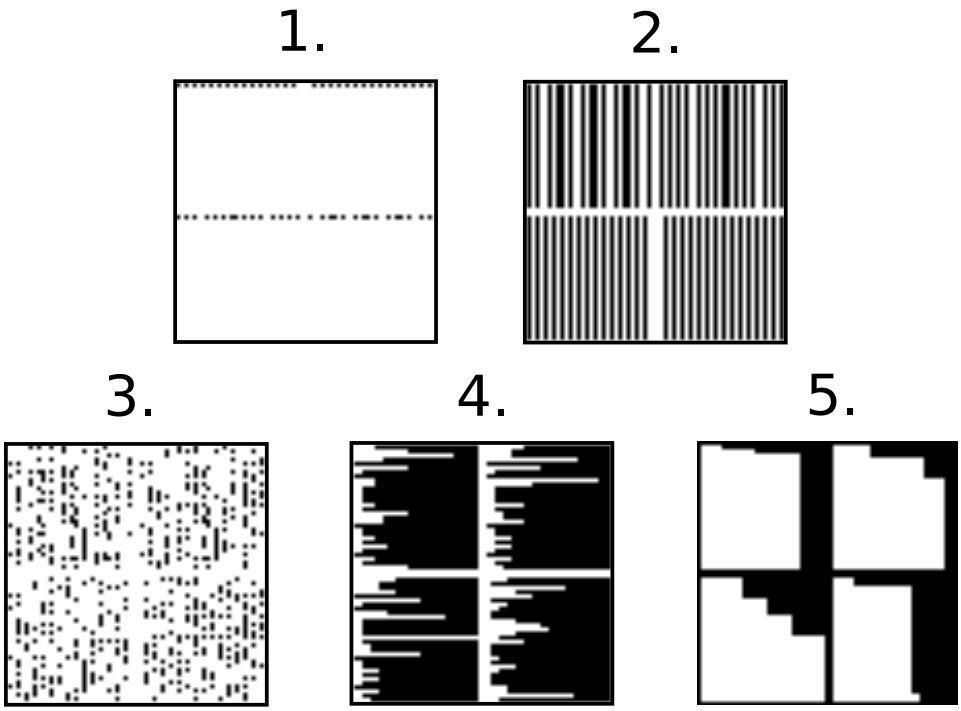

Figure 8.19: Parallel computing of Boolean functions using the transient mask. Measurements from the MIPA4k CNN processor.

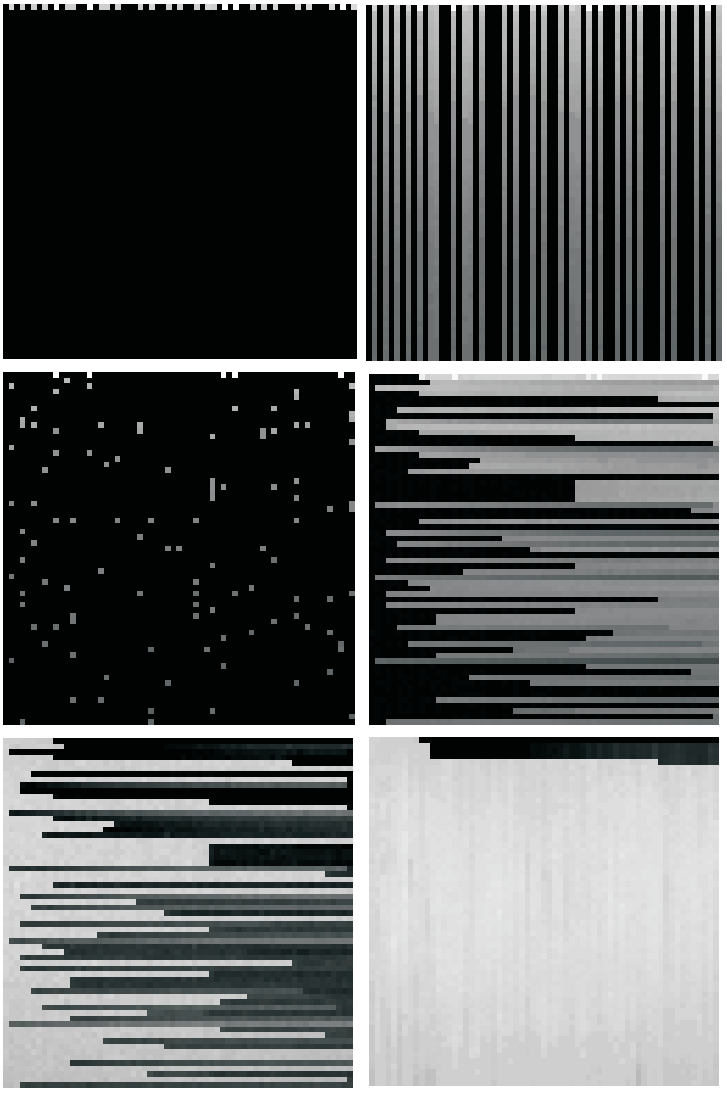

Figure 8.20: Measurements from the MIPA4k CNN processor configured in gray scale mode of operation, corresponding to Figures 8.16, 8.17, and 8.18. For technical reasons, the first four computing steps are performed on inverted values of Boolean variables, and the cell states are then inverted in the fifth step.

Chapter 9

Memristive Associative Memories

In this chapter various memristive implementations of a neuromorphic memory structure called the *associative memory* are considered. In contrast to random access memories, where the information is stored to and retrieved from explicitly given locations, 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. Such a search typically requires computations of many thresholded sums of bitwise correlations between the input vector and the contents of the memory. In a parallel hardware implementation of a large-scale associative memory one thus needs many such sum-and-threshold units, which in neuromorphic terms can be regarded as artificial neurons.

In pure CMOS implementations of associative memories the artificial neurons must share die area with the memory elements, which can be for example SRAM or flash memory cells. In contrast to this, in a CMOL implementation the memory elements, memristors, are located above the CMOS layer, which frees up CMOS die area and enables more artificial neurons to be fabricated in CMOS. As noted in Subsection 8.5, a memristor feature size of $F = 50$ nm yields a synaptic density of 10^{10} memristive synapses per square centimeter, which is comparable to that of the human cortex [96].

The prospect of scaling up the capacities of associative memories is the prime motivation of this chapter, as it would allow for architectures which due to their complexity cannot be simulated in real-time in software using conventional CMOS memory architectures. In particular, throughout this chapter it is assumed that the word length L of the memory input — and output — is very large, of the order of thousands of bits. Such a memory architecture would be well suited for example for real-time pattern recognition

in natural images, which would be useful in autonomous robotics, to name one particular field of application. Moreover, there is a reason to believe that the associative memory in the brain uses high-dimensional input and output data [39], and thus such a memory may be needed for implementing a whole-brain model such as the one discussed in [96]. The assumption of very wide wordlengths distinguishes the considerations of this chapter from some of the proposals of memristive associative memories in the literature, where associations were modelled using only a few memristors [85, 94].

This chapter is organized as follows. Section 9.1 discusses the basic principles of associative memories including different data representations and memory capacities. In Section 9.2 the autoassociative content-addressable memory (ACAM) architecture is considered, and its memristive implementation is presented. In Section 9.3 memristive implementations of a certain class of associative memory structures called sparse distributed memories are considered. Finally, a summary and discussion of the memristive associative memory architectures disussed in this chapter is presented in Section 9.4. The main reference for this chapter is our publication [65].

9.1 Definitions and architectures

9.1.1 Associative memory

Let $\mathbf{u}_i$ and $\mathbf{v}_i$, where $i = 1, \ldots, M$ be binary vectors of length L. An associative memory can store a set of associations $\mathbf{u}_i \to \mathbf{v}_i$ between these vectors. Formally this means that when the memory is searched by a vector $\mathbf{z}$, it returns the vector $\mathbf{v}_i$ (or just the index i) whose index i minimizes the Hamming distance

$$d_H(\mathbf{z}, \mathbf{u}_i) = \sum_j^L (\mathbf{z}(j) - \mathbf{u}_i(j))^2. \tag{9.1}$$

Here $\mathbf{x}(j)$ denotes the jth element of $\mathbf{x}$.

A memory satisfying the definition above is called *heteroassociative*, and besides simply storing key-value pairs, it can also be used to store sequences of vectors provided that the key and value vectors have the same length. A sequence is formed by letting the value of a previous pair to be the key of the next pair. When $\mathbf{u}_i = \mathbf{v}_i$ for all i, the memory is called *autoassociative*, and it allows for pattern completion or error correction.

An early review and system-theoretical formulation of associative information structures was given in [47]; notably also physical realizations of content-addressable and distributed memory structures were presented in this monograph. Furthermore, a first broad review and analysis of content-addressable memories and their hardware implementations was presented

in [48]. Further discussions on the theoretical basis of the memory architectures considered in this work can be found for example in [30,37,41,49,113].

9.1.2 Data representation

Throughout this chapter the vector length L is assumed to be very large, in the order of thousands of bits. Furthermore, all stored vectors are assumed to be either *sparse* or *dense*. A sparse vector contains only a small fraction of ones, for example 50 ones out of a total of $L = 10000$ bits, while in dense binary vectors the numbers of zeros and ones are close to $L/2$. Raw data is often inherently dense, and compressed data *is* dense, since it contains a maximal amount of information. These examples easily point out the need for associative memory architectures storing dense input patterns.

The usefulness of sparse representation may be understood by considering a data processing method called *dimensionality reduction*. In this scheme a dense vector, for example a gray-scale bitmap image, is mapped into a sparse vector whose non-zero coordinates contain most of the variance in the original data. Sparse representation is suitable for pattern recognition and completion, since the non-zero entries correspond to specific features in the data. For example in sparse coding of natural images [81], the basis functions correspond to oriented local structures.

It is known that the neural activity in parts of the brain is sparse [6]. In the visual and auditory cortex this may serve pattern recognition, but sparse activity of neurons is also beneficial in terms of metabolic efficiency [6]. This suggests that sparse representation may be useful in reducing the power consumption of hardware implementations of associative memories.

9.1.3 Unary and distributed architectures

Neural associative memory architectures can be divided into two categories: *unary* – also known as *grandmother cell* – architectures and *distributed* architectures [9]. In the former, each stored vector is allocated to a specific neuron which activates only when a vector close to it is given as input to the network. It follows from this that the number of neurons N is the upper limit to the number of binary vectors M the network can store, that is, $M \leq N$. In practice a unary associative memory can be implemented as a lookup table or as a content-addressable memory as discussed in Section 9.2.

In distributed memory architectures, multiple neurons are activated for each input. In principle this enables storing more input vectors than there are neurons in the network, that is, it is possible that $M > N$. The upper limit for the number of vectors that can be retrieved without errors depends not only on the memory architecture but also on the data distribution of the input vectors. When operating with sparse vectors the storage capacity

Table 9.1: Network capacities of associative memory architectures.

Architecture	C_D	C_S
Autoassociative CAM	1	N/A
Willshaw memory	N/A	0.69
Sparse Distributed Memory	0.15	≥ 0.69

of the network may be much greater than N when N is large. Memristive implementations of two distributed memory architectures are proposed in Section 9.3.

9.1.4 Capacities of associative memories

The *network capacity* C of an associative memory is defined as the maximum quantity of stored information per synapse [46]. By definition C is always non-negative, and for binary synapses it satisfies $C \leq 1$. To calculate the *vector capacity* M which is the number of vectors that can be stored into the memory, one needs to know C, the total number of synapses S, and the information content in a single vector I, assuming that the input vectors are independent and contain an equal quantity of information. Then

$$M = CS/I. \tag{9.2}$$

For example, in the case of a square associative memory with N neurons, the number of synapses equals $S = N^2$. If logarithmically sparse vectors of length $L = N$ are used, each vector contains $I \approx \log(N)^2$ bits of information. Then the vector capacity of the memory equals

$$M \approx CN^2/\log^2(N). \tag{9.3}$$

On the other hand, if dense vectors are stored, then $I = N$ and

$$M = CN. \tag{9.4}$$

For example, with $N = 10^6$, one obtains $N^2/\log^2(N) \approx 5 \times 10^9$, which shows that the vector capacity M depends strongly on the distribution of the input data.

In Table 9.1 the network capacity values of the associative memories discussed in this chapter are presented. These results were adopted from references [38, 41, 46]. Here C_D and C_S denote the capacity for dense and sparse data vectors, respectively. Notice that sparse vectors are not considered for the autoassociative content-addressable memory (ACAM), while Willshaw memory is assumed to operate only on sparse vectors.

9.1.5 Literature review of memristive associative memories

Memristive associative memories have been presented in literature in [5, 25, 85, 94, 110]. Of these the most relevant to this chapter is [110], which discusses the implementation of a classic associative memory architecture called the *Hopfield network* [30] as a CrossNet circuit. This implementation requires analog synaptic weights, which are proposed to be realized as small crossbars of binary devices, and it yields a vector capacity of $M = 0.118L$ [108]. Being a CMOL-based neuromorphic network, this architecture closely resembles the ones described in Sections 9.2 and 9.3. The Hopfield network has several disadvantages when compared to the memory designs described in this chapter. As noted above, the capacity of the memory is directly proportional to the length of the input vector. With the memristive ACAM described in Section 9.2 or the Sparse Distributed Memory discussed in Section 9.3 this is not the case, as their capacities depend on the number of rows in the memory matrices, and can therefore be chosen independently from the input vector length. Moreover, the Hopfield network is a dynamical system, whose artificial neurons need to communicate with each other multiple times during an associative search in order to find the equilibrium state that corresponds to the output of the search. In contrast to this, the memory structures described in this chapter yield the output of a search in a single step. The capacity of a Hopfield network using dense input vectors equals that of a similarly sized SDM, while on logarithmically sparse vectors the capacities of a Hopfield network and the Willshaw memory described in Section 9.3 are roughly the same [46]. Therefore it can be concluded that the associative memory architectures presented in this chapter are better suited for a memristive implementation than the Hopfield network, but yield the same or higher capacities.

Memristive implementations of a *content-addressable memory* (CAM) have been proposed in [5] and [25]. The CAM is a computer memory architecture which compares input search data against a table of stored data, and returns the address of the matching data [82]. It is not an associative memory as defined in the context of this chapter, as it recognizes an input only if it exactly matches a stored vector (or an explicitly defined part of that vector). Therefore despite its name, the design of the autoassociative CAM in Section 9.2 differs significantly from the ones described in [5, 25].

In [85] and [94] the dynamics of analog memristors are used to create associative responses in small scale memristive circuits. For example [85] presents a three memristor circuit, whose dynamics are reminiscent of the famous Pavlov's experiment. These considerations differ considerably from the approach taken in this chapter, as the main interest in the following is to investigate memristive associative memories with very large capacities.

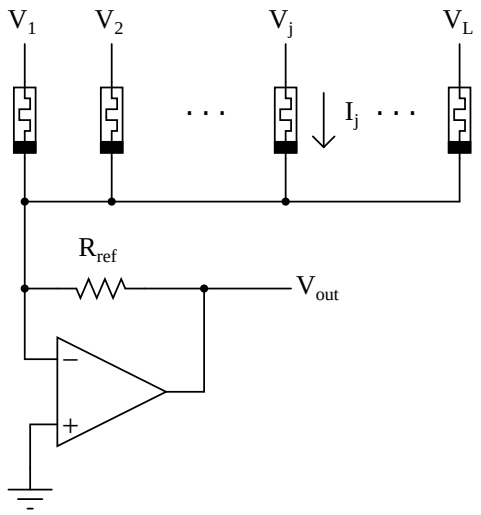

Figure 9.1: A memristive circuit for computing the inner product $\mathbf{b} \circ \mathbf{a}$. The elements of the binary vector $\mathbf{b}$ are represented by the voltages $V_j \in \{0, V_R\}$, $j = 1, \ldots, L$, whereas the elements of the real-valued vector $\mathbf{a}$ are represented by the currents I_j. The voltage V_{out} is directly proportional to the value of the inner product.

9.2 Memristive autoassociative CAM

Recall from Remark 41 that the inner product between a binary vector and a continously valued vector can be easily computed as a current sum using memristors, binary voltage inputs, and a virtual ground. In Figure 9.1, such an inner product circuit is depicted. This subcircuit is a crucial part in the associative memory architectures described in this chapter.

9.2.1 Autoassociative CAM

An *autoassociative CAM* (ACAM) [9] compares input search data against a table of stored data, and returns the addresses of the stored data which are nearest to the input data in Hamming distance. Since ACAM is a unary architecture, its vector capacity equals $M = N$, where N is the number of memory rows, independent of the distribution of the input data. In this section dense representation of data is assumed; associative memories operating on sparse data are considered in Section 9.3.

Let the contents of the memory be represented by a binary matrix U, whose rows $\mathbf{u}_i$ correspond to the stored vectors, and let $\mathbf{z}$ be the input vector for the associative search. Therefore the search should yield the indices i for

which the Hamming distance $H(\mathbf{z}, \mathbf{u}_i)$ is minimized. Since

$$H(\mathbf{z}, \mathbf{u}_i) = ||\mathbf{z}||^2 - 2\mathbf{z} \circ \mathbf{u}_i + ||\mathbf{u}_i||^2, \tag{9.5}$$

it follows that the vector $\mathbf{u}_i$ that minimizes the Hamming distance is the one which maximizes the inner product $\mathbf{z} \circ \mathbf{u}_i$. Indeed, since the vector length L is large and the vectors were assumed to be dense, it follows that

$$||\mathbf{z}||^2 \approx ||\mathbf{u}_i||^2 \approx L/2. \tag{9.6}$$

For the hardware implementation it is very convenient that the inner product is sufficient for measuring the distance between two vectors: due to this only ones, and not zeros, need to matched in the input vector and the stored vectors.

9.2.2 Implementation of a memristive ACAM

Consider a CMOL-type implementation of an ACAM whose memory elements are binary memristors, as depicted in Figure 9.2. The contents matrix U is represented by the conductances of the memristors in the memristive crossbar. The vertical nanowires are used for communicating the input and output vectors, and comparison of input and stored vectors is performed over the horizontal nanowires. All of the nanowires are interfaced with CMOS blocks as depicted in Figure 9.2(b) — the vertical nanowires are interfaced with *driver blocks* (DB), while the horizontal nanowires are interfaced with *comparison blocks* (CB).

In general the length L of an input vector and the number N of the stored vectors need not be equal. Consider for example the case $N > L$. This corresponds to N horizontal and L vertical nanowires, and thus $N-L$ of the CMOS cells consist only of a comparison block depicted in Figure 9.2(b). Notice here that the word *cell* is used instead of a neuron, since each of the CMOS cells has also local memory latch and inputs of multiple global signals, and therefore contains more functionality than a generic artificial neuron. This choice of terminology is maintained in Section 9.3 where the CMOS cell is extended to function as part of the Sparse Distributed Memory. In the following the operation of the proposed memristive ACAM is presented in detail.

Storing a vector

When a binary vector $\mathbf{u}$ is stored into the memory, an available row of the ACAM is chosen. The corresponding horizontal nanowire is driven to a negative voltage $-V_{\text{prog}}$, and the vertical nanowires are driven sequentially to voltages corresponding to the bits of $\mathbf{u}$: the jth vertical nanowire is driven

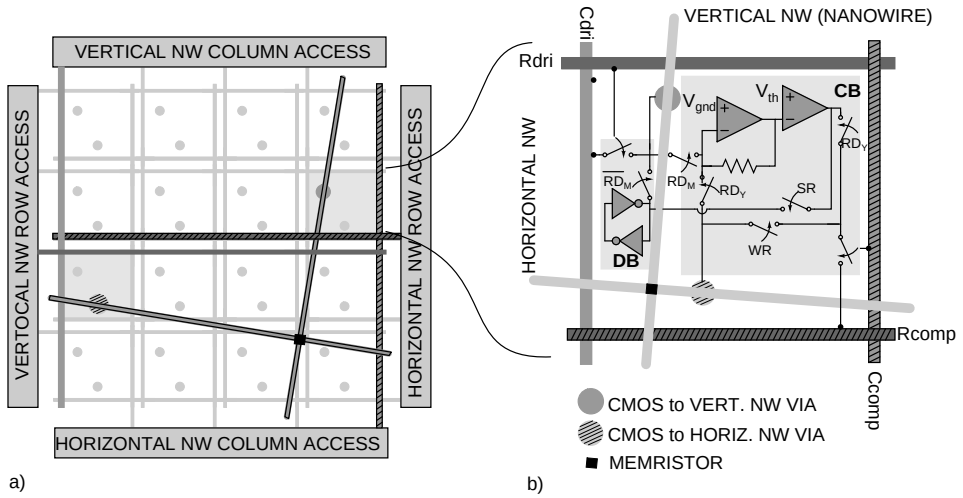

Figure 9.2: Schematic of the memristive ACAM. (a) High-level view of the memory architecture. (b) A single CMOS cell.

to a positive voltage V_{prog} if the corresponding bit $\mathbf{u}_j = 1$, and to ground otherwise. The voltage V_{prog} is chosen to satisfy

$$V_{\text{prog}} < V^T < 2V_{\text{prog}}, \tag{9.7}$$

where V^T is the programming threshold of the memristor. This assignment of voltages programs the bits of the input vector as the resistances of the memristors on this row of the nanowire crossbar: value zero is represented as a memristor in the OFF-state, and value one is represented by a memristor in the ON-state. If the row has been used previously, one may first initialize it by driving a large negative voltage across the corresponding memristors. The selection of the nanowires is accomplished with the CMOS microwires denoted by Cdri, Rdri, Ccomp and Rcomp, as is depicted in Figure 9.3a). In particular, Rdri controls the switch connecting Cdri to the latch in the driver block, while Ccomp controls the switch connecting Rcomp either to the CMOS circuitry in the comparison block or directly to the horizontal nanowire, depending on the selected global configuration of the memory circuit.

Search operation

When the memristive ACAM is searched by a binary vector $\mathbf{z}$, the driver blocks are used to drive the vertical nanowires to voltages corresponding to $\mathbf{z}$: the jth vertical nanowire is driven either to a positive voltage V_{read} if $\mathbf{z}_j = 1$, or to ground if $\mathbf{z}_j = 0$. For this, the search vector must be sequentially stored into the driver block latches before the search as depicted in Figure 9.3(b),

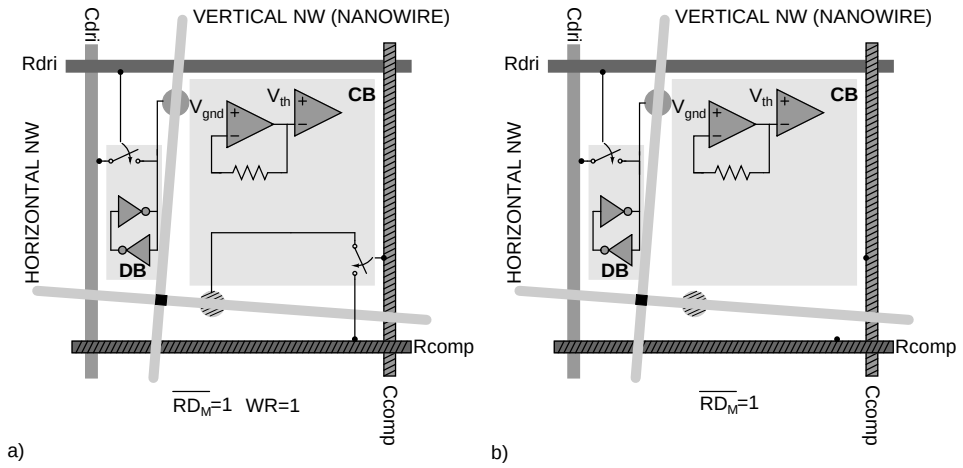

Figure 9.3: Memristive ACAM CMOS cell configured for the write-in operations. (a) Storing the input vector as the states of the memristors on a horizontal nanowire. The programming of the memristors is performed by driving the vertical nanowires sequentially by voltages $\{V_{\text{prog}}, \text{GND}\}$ corresponding to the input bits, and by driving the selected horizontal nanowire at a constant negative voltage $-V_{\text{prog}}$. The horizontal nanowires not attending to this operation can be connected to ground, or they can be left floating. On these nanowires the memristors are not programmed, as the voltage across them does not exceed their programming threshold. (b) For the search operation, the input vector bits are stored into the latches at the driver blocks.

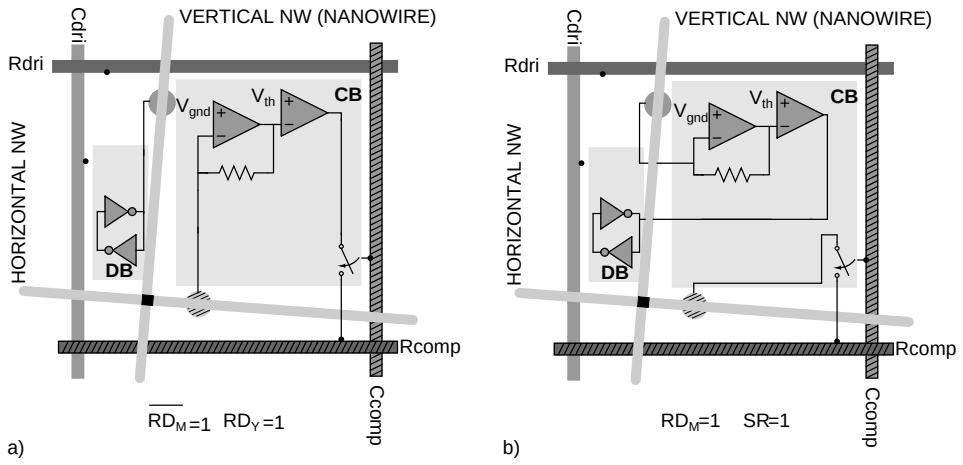

Figure 9.4: Memristive ACAM CMOS cell configured for the read-out operations. (a) Search operation. The input vector bits are driven to the vertical nanowires by the driver block latches. The horizontal nanowires are connected to virtual ground, and the incoming current is measured and thresholded. The result of the threshold comparison can be read from the Rcomp microwire. (b) Read-out of a stored vector. A selected horizontal nanowire is driven with a read voltage, and the vertical nanowires are connected to the virtual grounds. The incoming currents from the vertical nanowires are thresholded, and the results are stored into the latches at the driver blocks, from where they can be read out.

since the addressing scheme does not allow providing data to all vertical nanowires simultaneously. Indeed, there are $\sqrt{N}$ address lines for the N driver blocks. The currents coming in from the horizontal nanowires to the virtual ground of the comparison cells are then measured and thresholded with a negative threshold voltage V_{th}, which should not be confused with the memristor programming threshold V^T. As described in the beginning of this section, the voltage at the input of the comparator represents the value of an inner product $\mathbf{z} \circ \mathbf{u}_i$. Thus the horizontal nanowires whose currents exceed the threshold value correspond to the rows of the ACAM which are within a desired distance of the search vector. The result can be read from the output of the comparator, as depicted in Figure 9.4(a).

In general, multiple stored vectors $\mathbf{u}$ may be close enough to the search vector $\mathbf{z}$ to be selected during the search. The number of selected vectors $\mathbf{u}$ depends on the threshold voltage V_{th} which can be correspondingly adjusted, for example in a logarithmic search, to yield k vectors $\mathbf{u}$ closest to $\mathbf{z}$.

Reading out

The read-out of a vector $\mathbf{u}_i$ is achieved by driving the corresponding horizontal nanowire while simultanously measuring the currents at the vertical nanowires. In Figure 9.4(b) the read-out operation is depicted. The value corresponding to a bit in the stored vector is written into the driver block's latch, from which it can be driven onto the Cdri microwire.

9.2.3 Simulation of the ACAM cell

To demonstrate the operation of the proposed memristive memory design, the ACAM cell was simulated with SPICE. In the simulation the binary memristor model described in Section 3.4 was applied with parameters $R_{\mathrm{ON}} = 10$ MΩ, $R_{\mathrm{OFF}} = 100$ MΩ, $V^T = 1.5$ V, and a switching time of approximately 50 ns. For simplicity only one memristor was included in this simulation — that located at the crosspoint of the horizontal and vertical nanowires connected to the simulated cell — and therefore the search operation corresponds to finding whether or not the input bit equals the stored bit. The memristor is assumed to be initially in the OFF-state.

The values of the control voltages are depicted in the top subfigure of Figure 9.5. The four subfigures below it show the voltages of the DB latch and Rcomp. The simulation consists of three parts.

From 0 s to 0.5 μs the cell is configured for storing a bit to the memristor, as depicted in Figure 9.3a). The value of the stored bit is driven to DB at time 0.1 μs, while Rcomp is driven to -1 V at time 0.3 μs. If the bit to be stored is one, the voltage across the memristor exceeds its programming threshold voltage, which programs the memristor to the ON-state.

The search operation, corresponding to Figs. 9.3b) and 9.4a), is performed from 0.5 μs to 1.0 μs. The input bit is driven to DB at time 0.6 μs, and the result of the search can be read from Rcomp at time 0.8 μs.

The cell is configured for read-out of the stored bit, as depicted in Figure 9.4b), from 1.0 μs to 1.5 μs. Rcomp is used to drive a read voltage of 1 V across the memristor, and the result is stored to DB. The read-out can take place from 1.3 μs on.

9.3 Sparse distributed memory architectures

As demonstrated by the network capacities given in Table 9.1, the associative memory architecture used in any given application should depend on the distribution of the stored data. If dense data is used, autoassociative content-addressable memory as described in the previous section is recommended. For operation with sparse data, *sparse distributed memories* are preferred.

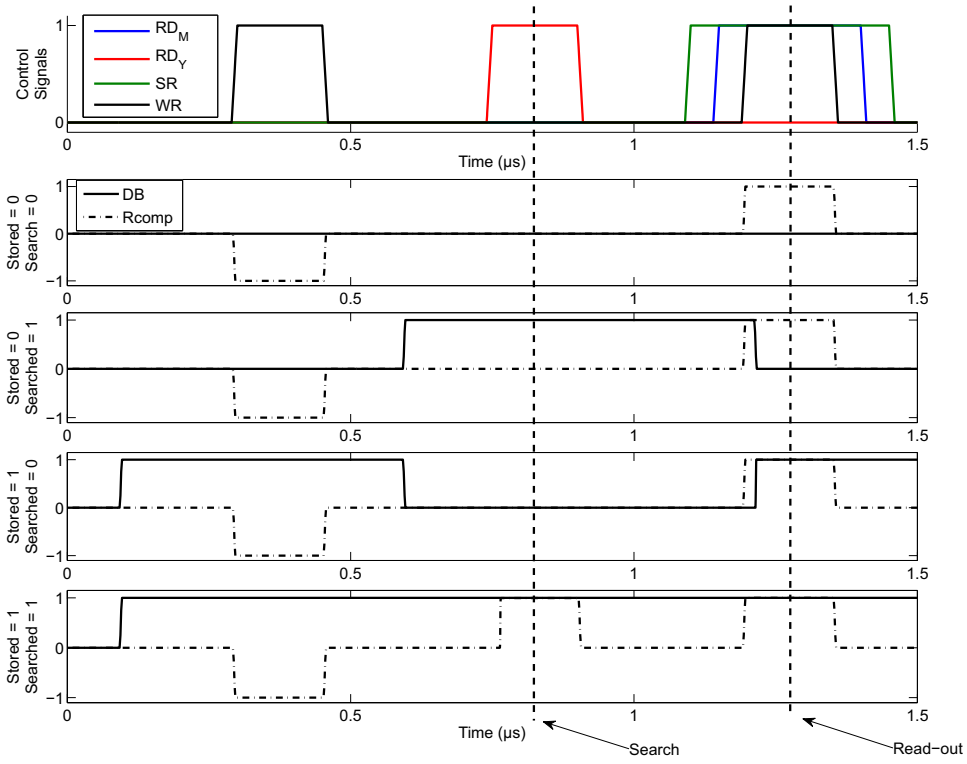

Figure 9.5: SPICE Simulation of a memristive ACAM cell using the binary memristor model. From 0 s to 0.5 μs the cell is configured for storing a binary value to this memristor. From 0.5 μs to 1 μs it is configured for the search operation, and from 1 μs to 1.5 μs it is configured for the read-out of the value of the memristor. Here the search operation equals the correlation of the stored bit and a searched bit, as only one memristor is considered for simplicity. Top inset: Control signals $\mathrm{RD_M}$, $\mathrm{RD_Y}$, SR and WR as a function of time. Other insets: Voltages at the DB and Rcomp nodes. The result of the search operation can be read from Rcomp at time $T = 0.8$ μs, while the result of the read-out can be read from DB at time $T = 1.3$ μs.

These are artificial neural-net associative memories that can be seen as associative generalizations of the conventional random-access memory (RAM). In contrast to RAM, where each address refers to its own memory location, addresses to sparse distributed memories activate multiple memory locations. This yields a distributed representation of the input address, which is used in the subsequent read and write operations in the memory: the data vector that is associated with an address is stored in multiple locations. In the following the memristive implementations of two distributed memory architectures: the Willshaw memory and the Sparse Distributed Memory (SDM) are presented. SDM can be seen as a generalization of the Willshaw memory with analog weights and an auxiliary memristive ACAM-type read-only memory which is used to make the size of the memory – and therefore also its capacity – independent of the input vector length.

9.3.1 Memristive Willshaw memory

The Willshaw memory [113] is a neuromorphic heteroassociative memory, which uses binary synapses and stores sparse data vectors. As the ACAM, its contents can be represented by a binary matrix, which is denoted by W. A key-value pair $\mathbf{u} \to \mathbf{v}$ of two sparse column vectors of lengths L and N, respectively, is stored into the memory by updating the matrix W according to the outer-product rule

$$W := \mathrm{OR}(W, \mathbf{v}\mathbf{u}^T). \tag{9.8}$$

As noted in Section 9.1.1, autoassociation is achieved by setting $\mathbf{u} = \mathbf{v}$. If the stored vectors are sparse and have K ones, the vector capacity of the Willshaw memory equals

$$M \approx 0.69(N/K)^2. \tag{9.9}$$

Since the number of ones in the matrix W never decreases when storing a new vector into the memory, one should use sparse representation of data in order to limit the amount of overwriting of previously stored data. In the information theoretical sense the maximum capacity of this memory is obtained when $K = \log_2(N)$, as is shown in [46].

The search of a Willshaw memory is performed as L thresholded inner products between the rows of W and the search vector $\mathbf{z}$:

$$\mathbf{v}(j) = H(\mathbf{w}_j \circ \mathbf{z} - \Theta), \tag{9.10}$$

where Θ is a threshold value and H is the Heaviside function

$$H(a) = \begin{cases} 1 & a \geq 0 \\ 0 & a < 0. \end{cases} \tag{9.11}$$

The optimal choice of the threshold depends on the types of bit errors present in the address $\mathbf{z}$. In the original Willshaw model [113] the threshold $\Theta = \sum \mathbf{z}_i$ was specified; this threshold is optimal in the case where $\mathbf{z}$ contains only miss-type errors, that is, only values $z_i = 0$ are potentially erroneous.

The only difference between the search operation of the Willshaw memory and that of the ACAM is the distribution of the input data. Indeed, with the Willshaw memory the search vector is sparse, and thus the threshold should be smaller than with the ACAM. Storing vectors is also performed very similarly to the ACAM, the only difference being that in the Willshaw memory the input is stored on multiple rows simultaneously. Therefore the memristive ACAM architecture described in Section 9.2 and depicted in Figure 9.2 implements also the Willshaw memory. As a conclusion, this memory architecture should be configured as an ACAM when dense vectors are used, and as a Willshaw memory when sparse vectors are used.

9.3.2 Structure and operation of a Sparse Distributed Memory

The *Sparse Distributed Memory* (SDM) is an artificial-neural-net associative memory whose circuit resembles that of the cerebellar cortex [37]. It also resembles the conventional RAM architecture more than the Willshaw memory does, as it uses an explicit address vector along with a word-in vector. A high-level view of the SDM architecture is depicted in Figure 9.6. It consists of two parts: a memristive ACAM-type read-only *address matrix* $\mathbf{A}$, and a Willshaw-type *content matrix* $\mathbf{C}$, whose elements are integer counters of small absolute value. The binary address matrix is used to produce a sparse *activation vector* $\mathbf{y}$, which indicates rows of the content matrix used in the store and retrieve operations.

A vector $\mathbf{z}$ is stored into SDM by incrementing the jth counters of all activated rows if $\mathbf{z}_j = 1$, and by decrementing them otherwise. The retrieve operation is performed by columnwise summing of contents of activated rows of the content matrix and by thresholding the result, thus yielding $\mathbf{w} = H(\mathbf{Cy})$, where H is the Heaviside step function (9.11). In other words the read operation works as with the Willshaw memory, whereas the write operation differs in that the value of the counter is not binary and it is allowed also to decrease. It has been shown that five-bit counters with values in $[-16, 15]$ are sufficient for practical operation of the SDM [37]. Reducing the range of the counter reduces the capacity of the memory – SDM works even with one-bit counters but then only the most recently stored data can be retrieved reliably. As the counters are implemented by analog memristors it follows that the capacity of the memristive SDM depends on the multilevel programming capability of the memristors. In particular, the programming rate of the state variable should be approximately constant at

a fixed programming voltage pulse, and the number of allowed state variable values should be large. The generic analog memristor model described in Section 3.4 satisfies these requirements, as does for example the physical memristor reported in [32].

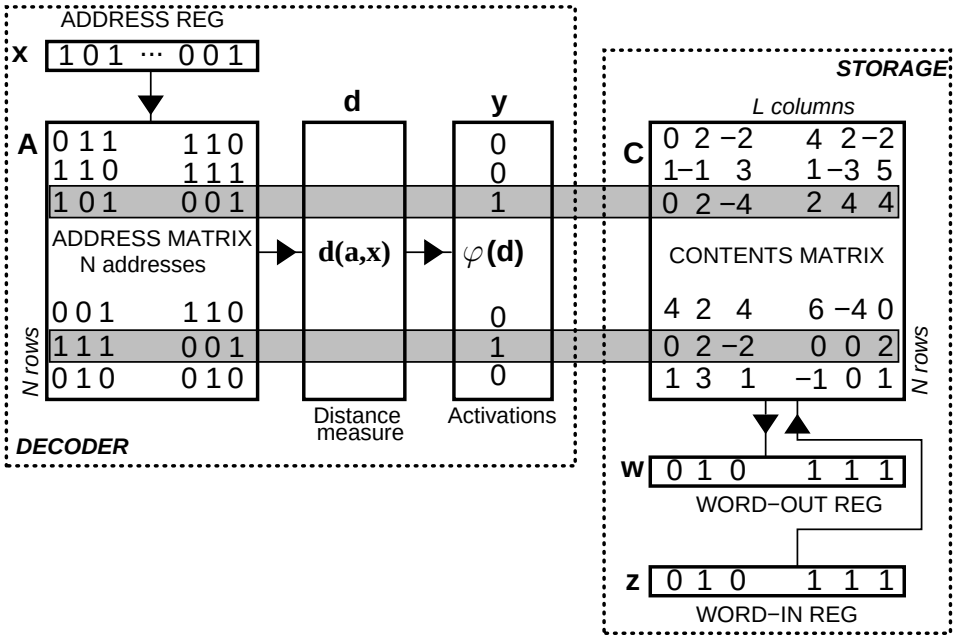

Figure 9.6: A high-level view of the SDM. The address vector $\mathbf{x}$ is compared with the rows of the address matrix A for example with respect to their Hamming distance, resulting in the distance measure vector $\mathbf{d}$. Thresholding $\mathbf{d}$ gives the activation vector $\mathbf{y}$, which is used to activate the corresponding rows in the content matrix C. In the search operation, these rows are summed element-wise, and the result is thresholded yielding the output vector $\mathbf{w}$. When the input vector $\mathbf{z}$ is stored into the memory, the corresponding activated counters on the content matrix rows are incremented or decremented as explained in the text above.

If the address register and the word-in register coincide then the SDM functions as an autoassociative memory. This gives rise to improved error-correction, as the word-out vector can be seen as a corrected version of the original input, and can be fed back as the input vector for an iterated search [37]. When the address register and the word-in register do not coincide but have the same length L, the feedback loop from the word-out register to the address register establishes a heteroassociative memory structure capable of storing sequences, as was described in Subsection 9.1.1. Also in this case there is iterative error-correction: an erroneously begun sequence converges to the stored one [37]. A mix of autoassociative and

heteroassociative functionality is obtained by using a word-in register of length $L + K$, and by copying the contents of the address register as the first L bits of the word-in register. When storing a vector, the last K bits of the word-in register can be arbitrary. Now a search can be iterated as in the autoassociative case, but in addition to the corrected address vector, one obtains another binary vector of length K, which gives a heteroassociation between the address vector and the word-in vector. This mode of operation of an SDM is thoroughly analyzed in [95].

In the following, a memristive implementation of an autoassociative SDM, for which the lengths of the address and word-in register both equal L, is presented. A major difference between the SDM and Willshaw memory architectures is that the capacity of the SDM is not limited by the input vector length L. This is because the address matrix yields a sparse activation vector whose length N is independent of L, and this activation vector is used as the locations to store the word-in vector in the Willshaw-type content matrix. Thus one can design an SDM for which $N \gg L$, and since the vector capacity M of the SDM is a function of N, it is possible that $M \gg L$. However, M depends heavily on the distribution of the word-in data. When it is dense, the capacity of the SDM is of the order $M = 0.15N$ [38]. On the other hand when the word-in data and the activation pattern are logarithmically sparse, the capacity is much higher, at least of the order of a Willshaw memory of that size,

$$M \approx \frac{0.69LN}{\log(L)\log(N)}, \tag{9.12}$$

the exact number depending among other things on the range of the counters in the content matrix and the required fidelity of the retrieval operation.

9.3.3 Implementation of a memristive SDM

Figure 9.7 shows the schematic of a memristive SDM cell. CMOS circuitry is used to implement the address register, the distance vector $\mathbf{d}$, the activation vector $\mathbf{y}$, the word-out register, and the word-in register. The address and content matrices are mapped to two vertically stacked memristive crossbars, as this layout allows for efficient utilization of silicon area. The address matrix is mapped to a crossbar array of binary memristors as in the ACAM, and the content matrix is mapped to a crossbar array of analog memristors. Compared to the ACAM implementation shown in Figure 9.2, the CMOS cell has been appended with a latch for bitwise storing of the activation pattern $\mathbf{y}$, and with multiple switches for selecting between the use of the address matrix and the content matrix, and furthermore between the write and read phases of the content matrix.

The search of the address matrix is depicted in Figure 9.8. During this phase the SDM cell is configured as an ACAM cell, and the the search

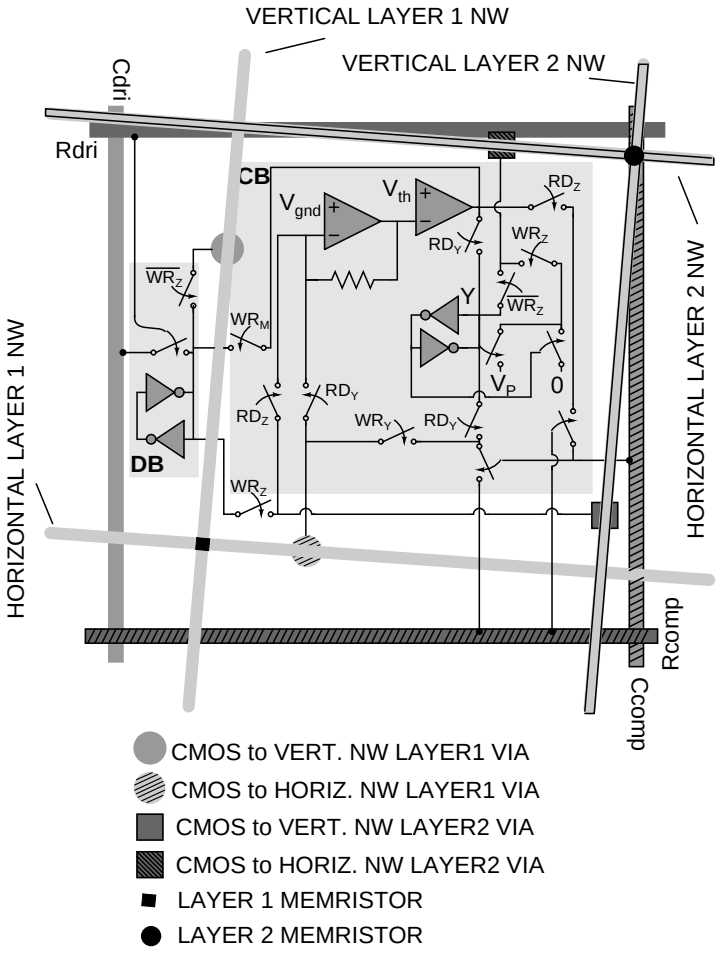

Figure 9.7: CMOS cell of the memristive SDM. This layout extends the CMOS cell of Figure 9.2(b) by an interface to a second nanowire layer and some required CMOS logic for the operation of the memory architecture. The iterative search of SDM is obtained by closing the switch WR_M and copying the result of the search from the latch Y to the input latch in the driver block DB. Then a new search can be conducted with the updated search vector.

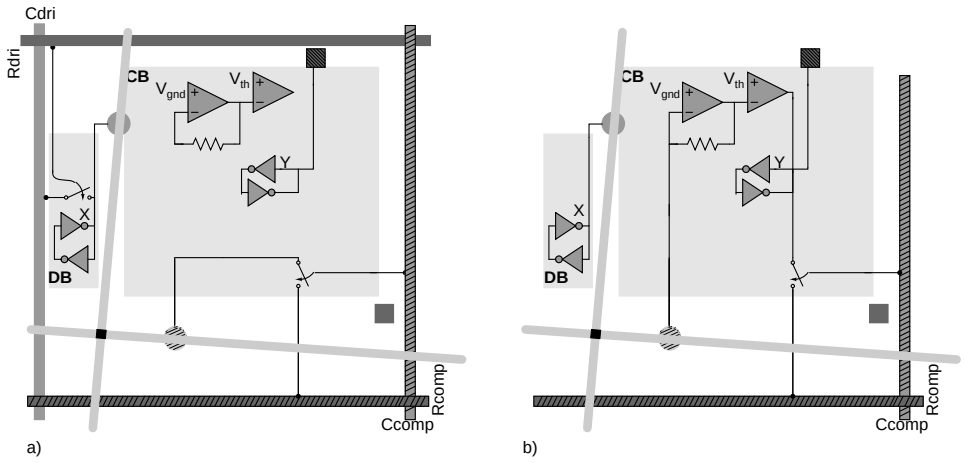

Figure 9.8: SDM cell configured as an ACAM for the address matrix search. (a) Storing the input vector as the states of the memristors on a horizontal nanowire. Using this configuration, the input vector bits can also be stored into the latch at the driver block. (b) The search operation. In contrast to the configuration depicted in Figure 9.3(b), the result of the search is stored to the latch Y.

operation is conducted as explained in Section 9.2. In contrast to Figure 9.2, the thresholded result which identifies the active rows in the SDM Decoder is stored into the latch Y instead of a direct read-out. This result is used to read and write the content matrix. The retrieval of data from the content matrix is depicted in Figure 9.9(a). The latched result Y is used to drive the second-layer horizontal nanowire, while the second-layer vertical nanowire is connected to the virtual ground at the input of the operational amplifier. As noted in the previous subsection, the retrieval is achieved by thresholded matrix product $\mathbf{w} = H(\mathbf{Cy})$, where the matrix $\mathbf{C}$ consists of non-negative analog values, and the vector $\mathbf{y}$ is binary. Thus the inner product method described in Section 8.5 can be applied here. Notice that the threshold voltage V_{th} should be proportional to the number of rows selected by address matrix search, and to count the number of selected rows, additional analog CMOS circuitry needs to be used. For example, each activated CMOS cell can drive a constant current on a global microwire, and by measuring the sum current one obtains the number of activated rows. However, for simplicity this part of the CMOS cell circuitry has been omitted from the schematic of Figure 9.7.

Writing to the content matrix is performed by applying a voltage pulse on those second-layer horizontal nanowires which are selected by the address matrix search explained above. The direction of the programming is determined by the word-in vector which must be written bitwise to the latches

at the driver blocks in advance. The configuration of the memristive SDM cell during the writing of the content matrix is depicted in Figure 9.9(b).

9.3.4 Simulations and error analysis

The main difference in the operation of the proposed implementations of the SDM and the ACAM is the programming of the analog weights representing the SDM counters. The fidelity of the retrieve operation depends on the accuracy of the programming of the analog memristors — the retrieve operation itself is simply implemented by the inner product method described in the beginning of Section 9.2. Simulations of the ACAM cell presented in Subsection 9.2.3 apply here for the search and retrieve operations; in the following a simulation of programming the analog memristor located at the crosspoint of the second level horizontal and vertical nanowires contacted to the simulated cell is presented. Furthermore, the effect of mismatched programmming rates of analog memristors on the capacity of the SDM is shown.

In Figure 9.10 voltage waveforms during the write operation are shown. Here Z denotes the voltage at the driver block latch and corresponds to the value of a single bit of the word-in vector, Y is the voltage at the comparison block latch corresponding to a single bit of the activation vector, and V_P is a square wave voltage signal used to program the analog memristors. The voltage signal V_P is propagated onto the second-layer horizontal nanowire only if $Y = 1$, that is, if the corresponding SDM row is selected at the SDM Decoder. If $Y = 0$, the second-layer horizontal nanowire is tied to ground. Moreover, due to an appropriate choice of the memristor's programming threshold, it is programmed only when the polarity of V_P is different from the polarity of Z. Thus the direction of the programming depends on the value of Z, that is, on the value of the corresponding bit of the word-in vector. The amount of change in the state of a memristor when programmed is determined by the amplitude and wavelength of V_P.

The accuracy of the counters depends on the characteristics of the analog memristors. The proposed write operation does not guarantee an integer change in the counter value, as the amount of programming step is affected by several nonidealities, including device-to-device mismatch in the programming thresholds among the analog memristors. On the other hand, as the write operation is distributed over multiple second-layer horizontal nanowires corresponding to rows in the content matrix, the individual variations may average out. The advantage of the proposed design is that programming can be performed in parallel for each memristor on a given row, and simultaneously for all rows. If in practice the accuracy of this pulse-based programming is not enough, the cyclical programming described in Subsection 4.2.2 can be used. However, cyclical programming requires mul-

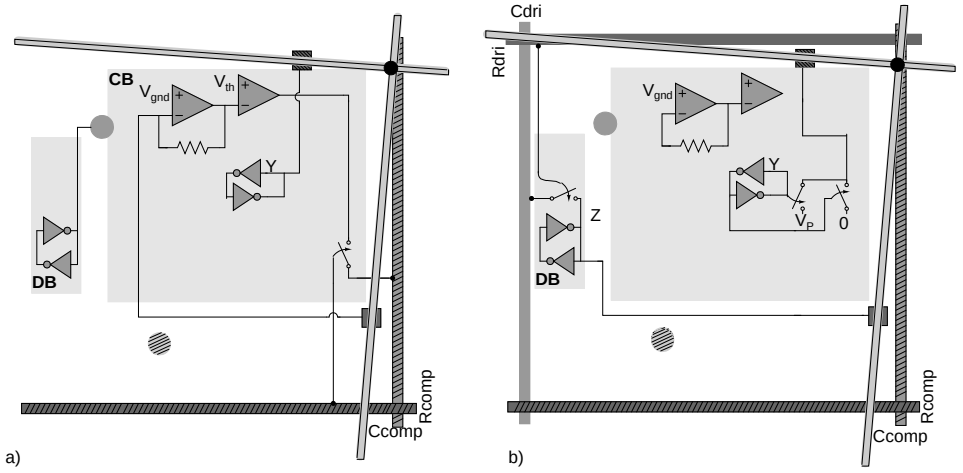

Figure 9.9: SDM CMOS cell configured for reading and programming the content matrix. (a) retrieval operation. The result of the address matrix search is driven from the Y latch to the second-layer horizontal nanowire, while the second-layer vertical nanowire is connected to virtual ground. The currents coming in from the second-layer vertical nanowires are measured and thresholded, and the result of the retrieval operation can be read from the Rcomp microwire. (b) Incrementing and decrementing the counters in the content matrix. The rows selected for this operation are determined by the result of the address matrix search stored into latch Y. The direction of programming is determined by the bit values Z stored into the latch at the driver block. Global square wave voltage signal V_P is used for the programming.

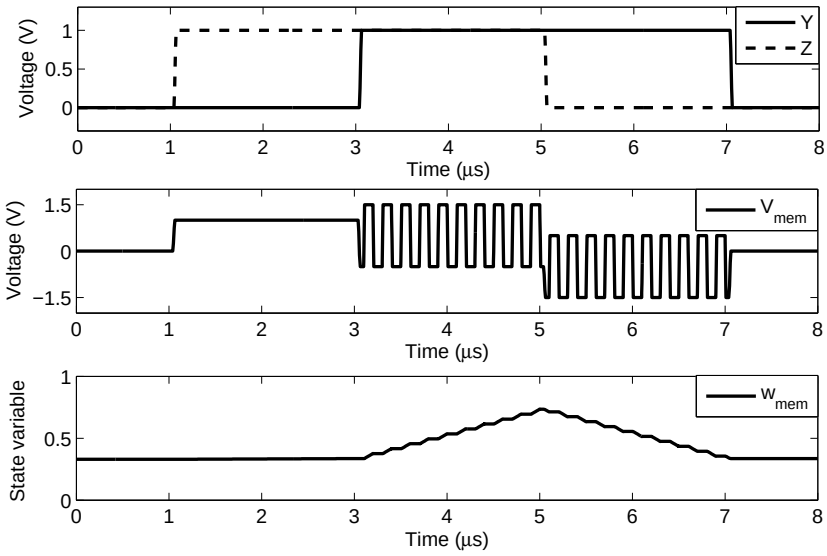

Figure 9.10: SPICE Simulation of the programming of an analog memristor in the content matrix. Top inset: different combinations of the latch voltages Y and Z. Middle inset: voltage across a simulated analog memristor within the content matrix. Bottom inset: state variable $w \in [0, 1]$ of the memristor during the programming. Programming takes place when the memristor is selected by the activation signal Y, while the direction of the programming depends on the value of the word-in bit Z. Square wave voltage signal V_P used in this simulation had DC value of 0.5 V, amplitude of 1.0 V and frequency of 5 MHz. The generic analog memristor model described in Section 3.4 was used with parameters $\alpha = 1 \times 10^{-3}$, $\beta = 1 \times 10^{-2}$, $\eta = 0.27$, and $\lambda = 10$, corresponding to a programming threshold of approximately $V^T = 1.1$ V at the timescale of 1 μs.

tiple programming cycles per device, and cannot be easily applied to all memristors in the content matrix simultaneously.

Figure 9.11 shows simulation results for the effect of inaccurate programming on the capacity of an SDM. In the simulations, a varying number of dense binary input vectors were stored in the content matrix with $L = N = 2^{11}$ using logarithmically sparse activation patterns, corresponding to the theoretical vector storage capacity $M = 0.15N$. This capacity is defined as the number of vectors storable in the memory with a bit error probability $P_E = 0.005$. Pulse-based programming of the memristors was assumed with 32 nominal states. Errors in the programming phase were approximated by assuming that the magnitude of state change in programming a memristor is drawn from the normal distribution $\mathcal{N}(1, \sigma_P^2)$. These magnitudes were assumed to be different for each memristor in the content matrix, but fixed for a given memristor. The state value of each memristor was limited to $[-16, 15]$; note that due to the random variation in the programming, not all devices have exactly 32 distinct states within this programming interval. This simulation indicates that the SDM is not very sensitive to error in programming the counter values: a 10% standard deviation of the programming magnitude has negligible effect on the bit error probability in the content matrix, and a standard deviation of over 80% is required to reduce the vector capacity by a factor of 0.5.

9.4 Discussion on hardware requirements

Recall from Section 9.2 that an important basic operation in the considered memory implementations is the computation of inner products between vectors as current sums over nanowires. It is thus relevant to consider the resolution with which it is necessary to distinguish between the magnitudes of these current sums in practical circuits. Suppose the input vector length $L = 10000$. The inner product of two independent dense vectors $\mathbf{u}$ and $\mathbf{v}$ is modeled by the binomial distribution $B(n, p)$ with $n = 10000$ and $p = 0.5$. The standard deviation of this distribution is $\sqrt{np(1-p)} = 50$, and for example, less than a thousand-millionth of vectors $\mathbf{v}$ in $\{0, 1\}^L$ have an inner product with $\mathbf{u}$ smaller than 4700. In other words, if two vectors have an inner product smaller than 4700 or greater than 5300, it is highly probable that they are not sampled independently from the uniform distribution. To discriminate between inner product values of 5000 and 4700, one needs to be able to distinguish the corresponding current sums with the analog circuitry. Thus the analog circuitry must have at least a five-bit resolution, as $L/2^5 \approx 300$. On the other hand, if sparse data vectors are used, some tens of input currents are summed together, and even small variations in the input currents should be distinguished. Again, a resolution of five bits, or

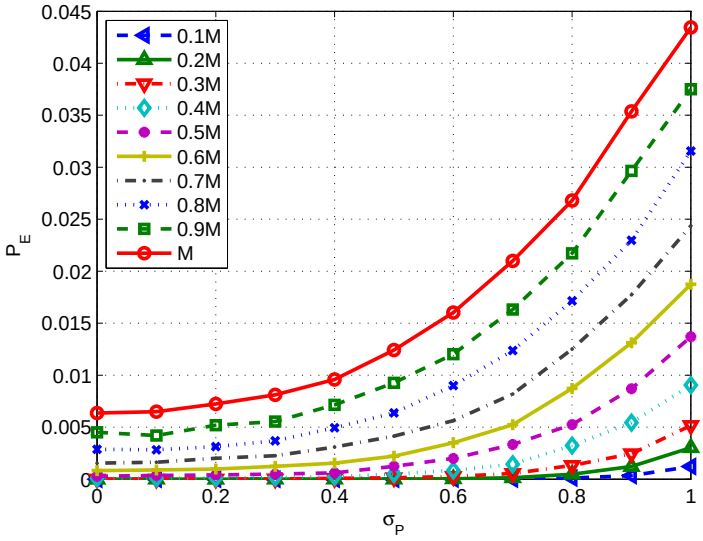

Figure 9.11: Probability of bit error P_E vs. standard deviation σ_P of the programming error in the SDM content matrix, simulated for $L = N = 2^{11}$, nominal memristances integer-valued in $[-16, 15]$, and number of stored vectors from $0.1M$ to M, where $M = 0.15N \approx 307$.

32 different current levels, seems like a reasonable choice when $L = 10000$.

Consider then the area required by the single CMOS SDM cell of Figure 9.7. Each cell consists of multiple switch transistors, four inverters, and two operational amplifier subcircuits. When a modern CMOS process is applied, the analog circuits in the cell require most of the area. The area of the analog CMOS circuits is governed by their accuracy requirements, as random variation reduces with increasing transistor area, and can be accurately predicted for a given circuit [45]. However, it must be noted that the random variation in memristors is not yet well understood. As the targeted computing accuracy is affected by the combined variation in CMOS and memristors, only rough area estimations are possible at this point. For a density estimation of the proposed memristive associative memory architecture, assume that a CMOS cell area of 20×20 μm^2 and a memristor footprint of 50×50 nm^2 yield sufficient accuracy for the analog current summing and thresholding. These estimates yield a CMOS cell density of 2.5×10^5 cells per square centimeter, and a synaptic density of 6.25×10^{10} memristors per square centimeter. The chosen memristor footprint is the size of the digital memristor reported in [36], whereas the analog memristor reported in [32] is approximately four times larger. The smallest analog memristor reported so far is [2], with an active switching area less than 20 nm^2, but this work

is very recent, and no thorough description of this device has yet been published. In any case, since the implementations of the ACAM and Willshaw memories require only binary memristors, they could be fabricated at the estimated density using current technology.

From the fabrication point-of-view, there are some challenges in the proposed memristive associative memory architecture. One point of concern is the requirement for unbroken, relatively long nanowires. This requirement comes up also with conventional memristive RRAM-architectures, and it must be solved if such large-scale architectures are to be realized. Another problem may lie in the mismatch and non-linear programming among the analog memristors needed for the implementation of the Hopfield network and the SDM. The analog memristor in [32] seems to be suitable for these memories, but so far no large-scale crossbar of such memristors has been reported. Another possible solution is to use multiple binary memristors to emulate an analog weight, as was originally proposed for the implementation of CrossNets [70].

As mentioned in Section 9.2, in general an ACAM, a Willshaw memory, or an SDM need not have an equal number L of columns, or bits of the search vector, and rows N, or memory locations. The considered memristive implementation yields in this case a rectangular $N \times L$ nanowire crossbar. As the aspect ratio of a chip is limited in practice, this proposes a layout problem in order not to waste silicon area. Vertical stacking of nanowire crossbars and via translation [107] might present a solution to this; further consideration is left for future research, however. Similarly the capacities of the considered associative memory architectures may benefit from additional, vertically stacked memristive crossbars.

Chapter 10

Conclusion

This thesis considers the principles of memristive computing. This is a timely topic as the first memristive memory chips are predicted to become commercially available within the next few years. The results presented in this work indicate that memristors are well-suited for complementing CMOS circuitry also in computational tasks.

The level of abstraction in the treatment of memristive computing increased throughout the thesis, so that the first chapters were focused on the theoretical and physical foundations of a single memristor, while Chapters 8 and 9 proposed system-level architectures that significantly benefit from the characteristics of memristive devices. The middle part of this thesis, Chapters 4 through 7, described elementary programming and read operations of digital and analog memristors. All the considerations based on the contributions of our research group are theoretical, as we currently do not have access to fabricated memristive circuitry. However, we plan to begin the experimental validation of our work by the end of year 2012. This will be crucial for refining our ideas on memristive computing.

In the following I identify some common features of those computing architectures which by the results of this thesis seem to profit from a memristive implementation.

In large-scale integrated systems memristors are most likely located within nanowire crossbars, as such array structures mitigate the nanoscale aligment problem in fabrication. For interfacing the crossbars with active CMOS circuitry, the CMOL architecture is the most prominent candidate. In CMOL-type architectures, CMOS cells are used to drive the nanowires of the crossbar, and hence to access the memristors at the crosspoints of the wires. To take full computational advantage of this design, the CMOS cells should form a parallel processing architecture, in which memristors would be used both as memory elements and as programmable connections between the cells. As examples of such processing architectures, various implementa-

tions of artificial neural networks were considered in this thesis in Chapters 8 and 9. Certainly, other parallel processing architectures such as Networks-on-Chips could be implemented also as CMOL-type architectures. The first memristive computing architectures to be commercially available are likely to be field-programmable gate arrays, which were discussed in Section 8.2, since they benefit significantly from lifting the configuration bits from the CMOS circuitry to the memristive layer. In a memristive FPGA, CMOS cells contain the logic gates, while wiring is implemented by the memristive crossbar.

It is also possible for the memristors themselves to perform logic operations as was investigated in detail in Chapter 6 and Section 8.3. Memristive stateful logic can help to reduce the amount of logic circuitry needed at the CMOS cells, thus allowing for higher cellular densities at the CMOS layer. It is yet unclear to what extent stateful logic can replace more conventional logic in this type of computing within memory, and there are some fundamental open problems in its implementation. For example, it would be very useful if stateful logic could be completely parallelized so that multiple independent operations could be performed at once within a CMOL architecture. Another important question is how the control logic required for performing the stateful operations could also be implemented by memristors. In addition to parallel stateful logic within a CMOL architecture, another possible application area of stateful logic is in printed and organic electronics, where transistors consume significantly more area than memristors.

Analog memristors are considered in several parts of this thesis, as they allow for simple circuit realizations of continuous-valued processing. Physically however, analog memristors are typically larger than digital memristors, and correspondingly yield memristive crossbars with fewer devices. The computational effect of this tradeoff between the number of states in a single device and the total number of devices is hard to estimate before large-scale analog memristor crossbars are fabricated. Analog memristors seem to be very useful in certain applications, for example some mixed-mode architectures such as the sparse distributed memory of Chapter 9 benefit considerably from their use. At the moment, however, it seems likely that the first large-scale memristive architectures will use digital memristors with a few different conductance states, and thus analog devices must be emulated with digital memristors.

In a large-scale CMOL-type architecture the data input and output may become a bottleneck, as the number of memristors is typically very much larger — ideally, the fourth power of — the number of CMOS addressing wires. It is beneficial then that processing is performed as close to the memristive devices as possible. A future remedy for the data transfer problem may be 3D integration, which would allow bringing the data to the CMOS cells in parallel. For example, a neuromorphic hardware mimicking

the functionality of the *ventral stream* [10] in the brain could consist of a 3D integrated stack of a pixel-parallel camera processor [88], an intermediate feature processing circuit and an associative memory used for object recognition.

The next decades will show the impact of memristive computing. Similar computational principles to the ones described in this thesis can certainly be used for other upcoming resistive memory technologies such as the phase-change memory. There are indisputable advantages in performing computation as close to memory as possible. A new level of parallelism will become available once intra-chip communication and computational tasks can be realized with programmable nanoscale components which do not consume area at the CMOS layer.

Bibliography

[1] International technology roadmap for semiconductors 2010. Available online at http://www.itrs.net.

[2] Adam, G., Alibart, F., Gao, L., Hoskins, B., and Strukov, D. B. Fighting variations in Pt/TiO_{2-x}/Pt and Ag/A-Si/Pt memristive devices. *Extended abstract in Nature Conference on Frontiers in Electronic Materials: Correlation Effects and Memristive Phenomena*, 2012.

[3] Afifi, A., Ayatollahi, A., and Raissi, F. STDP implementation using memristive nanodevice in CMOS-nano neuromorphic networks. *IEICE Electronics Express 6*, 3 (2009), 148–153.

[4] Alibart, F., Gao, L., Hoskins, B. D., and Strukov, D. B. High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm. *Nanotechnology 23* (2012).

[5] Alibart, F., Sherwood, T., and Strukov, D. B. Hybrid CMOS/nanodevice circuits for high throughput pattern matching applications. In *Proceedings of 2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)* (2011), pp. 279–286.

[6] Baddeley, R. An efficient code in V1? *Nature 381* (1996), 560–561.

[7] Bandyopadhyay, A., Serrano, G. J., and Hasler, P. Adaptive algorithm using hot-electron injection for programming analog computational memory elements within 0.2% of accuracy over 3.5 decades. *IEEE Journal of Solid-State Circuits 41* (2006), 2107–2114.

[8] Bao, B. C., Liu, Z., and Xu, J. P. Steady periodic memristor oscillator with transient chaotic behaviours. *Electronics letters 46*, 3 (2010), 237–238.

[9] Baum, E. B., Moody, J., and Wilczek, F. Internal representations for associative memory. *Biological Cybernetics 59*, 4-5 (1988), 217–228.

[10] Bear, M. F., Connors, B. W., and Paradiso, M. A. *Neuroscience: Exploring the Brain.* Lippincott Williams & Wilkins; Third edition, 2006.

[11] Bi, G., and Poo, M. Synaptic modification by correlated activity: Hebb's postulate revisited. *Annu. Rev. Neurosci. 24* (2001), 139–166.

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

[13] Chang, T., Jo, S.-H., Kim, K.-H., Sheridan, P., Gaba, S., and Lu, W. Synaptic behaviors and modeling of a metal oxide memristive device. *Applied Physics A 102* (2011), 857–863.

[14] Chen, A., Haddad, S., Wu, Y.-C., Fang, T.-N., Lan, Z., Avanzino, S., Pangrle, S., Buynoski, M., Rathor, M., Cai, W., Tripsas, N., Bill, C., VanBuskirk, M., and Taguchi, M. Non-volatile resistive switching for advanced memory applications. In *Proceedings of the IEEE International Electron Devices Meeting, 2005. IEDM Technical Digest.* (2005), pp. 746–749.

[15] Chen, Y., Jung, G., Ohlberg, D., Li, X., Stewart, D., Jeppesen, J., Nielsen, K., Stoddart, J., and Williams, R. Nanoscale molecular-switch crossbar circuits. *Nanotechnology 14* (2003), 462–468.

[16] Chevallier, C. J., Siau, C. H., Lim, S. F., Namala, S. R., Matsuoka, M., Bateman, B. L., and Rinerson, D. A 0.13μm 64Mb multi-layered conductive metal-oxide memory. In *Proceedings of the 2010 IEEE International Solid-State Circuits Conference* (2010), pp. 260–261.

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

[18] Chua, L. O., and Kang, S. M. Memristive devices and systems. *Proceedings of the IEEE, 64*, 2 (1976), 209–223.

[19] Chua, L. O., and Lin, G.-N. Canonical realization of Chua's circuit family. *IEEE Transactions on Circuits and Systems 37*, 7 (1990), 885–902.

[20] Chua, L. O., and Roska, T. *Cellular Neural Networks and Visual Computing: Foundations and Applications.* Cambridge University Press, 2005.

[21] Chua, L. O., and Yang, L. Cellular neural networks: Theory. *IEEE Transactions on Circuits and Systems Vol. 35*, 10 (1988), 1257–1272.

[22] DeHon, A. Array-based architectures for FET-based, nanoscale electronics. *IEEE Transactions on Nanotechnology 2*, 1 (2003), 23–32.

[23] DeHon, A., Goldstein, S. C., Kuekes, P. J., and Lincoln, P. Nonphotolithographic nanoscale memory density prospects. *IEEE Transactions on Nanotechnology 4*, 2 (2005), 215–228.

[24] Driscoll, T., Pershin, Y., Basov, D., and Ventra, M. D. Chaotic memristor. *Applied physics A 102*, 4 (2011), 885–889.

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

[26] Flocke, A., and Noll, T. G. Fundamental analysis of resistive nano-crossbars for the use in hybrid nano/CMOS-memory. In *Proceedings of the 33rd European Solid-State Circuits Conference* (2007), pp. 328–331.

[27] Gray, P. R., Hurst, P. J., Lewis, S. H., and Meyer, R. G. *Analysis and Design of Analog Integrated Circuits*. Wiley; 5th edition, 2009.

[28] Ho, Y., Huang, G. M., and Li, P. Dynamical properties and design analysis for nonvolatile memristor memories. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2010).

[29] Hodgkin, A., and Huxley, A. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology 117* (1952), 500–544.

[30] Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the USA 79*, 8 (1982), 2554–2558.

[31] Itoh, M., and Chua, L. O. Memristor oscillators. *International Journal of Bifurcation and Chaos 18*, 11 (2008), 3183–3206.

[32] Jo, S. H., Chang, T., Ebong, I., Bhadviya, B. B., Mazumder, P., and Lu, W. Nanoscale memristor device as synapse in neuromorphic systems. *Nano Lett., 10* (2010), 1297–1301.

[33] Jo, S. H., Kim, K. H., Chang, T., Gaba, S., and Lu, W. Si memristive devices applied to memory and neuromorphic circuits. In *Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS 2010* (2010), pp. 13–16.

[34] Jo, S. H., Kim, K.-H., and Lu, W. High-density crossbar arrays based on a Si memristive system. *Nano Letters 9*, 2 (2009), 870–874.

[35] Jo, S. H., Kim, K. H., and Lu, W. Programmable resistance switching in nanoscale two-terminal devices. *Nano Lett., 9* (2009), 496–500.

[36] Jo, S. H., and Lu, W. CMOS compatible nanoscale nonvolatile resistance switching memory. *Nano Letters 8*, 2 (2008), 392–397.

[37] Kanerva, P. *Sparse Distributed Memory.* MIT Press, Cambridge, Mass., 1988.

[38] Kanerva, P. Sparse distributed memory and related models. In *Associative Neural Memories: Theory and Implementation*, M. H. Hassoun, Ed. New York: Oxford University Press, 1993, pp. 50–76.

[39] Kanerva, P. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation, 1*, 2 (2009), 139–159.

[40] Karnaugh, M. The map method for synthesis of combinational logic circuits. *Trans. AIEE, Commun. & Electron. 72*, 1 (1953), 593–598.

[41] Keeler, J. D. Comparison between Kanerva's SDM and Hopfield-type neural networks. *Cognitive Science 12* (1988), 299–329.

[42] Kim, K., Shin, S., and Kang, S.-M. Stateful logic pipeline architecture. In *Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS 2011* (May 2011), pp. 2497–2500.

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

[44] Kim, K.-H., Hyun Jo, S., Gaba, S., and Lu, W. Nanoscale resistive memory with intrinsic diode characteristics and long endurance. *Applied Physics Letters 96*, 5 (Feb. 2010).

[45] Kinget, P. R. Device mismatch and tradeoffs in the design of analog circuits. *IEEE Journal of Solid-State Circuits 40* (2005), 1212–1224.

[46] Knoblauch, A., Palm, G., and Sommer, F. T. Memory capacities for synaptic and structural plasticity. *Neural Computation 22* (2010), 289–341.

[47] Kohonen, T. *Associative Memory*. Springer-Verlag, 1977.

[48] Kohonen, T. *Content-Addressable Memories*. Springer-Verlag, 1980.

[49] Kramer, M. A. Autoassociative neural networks. *Computers and chemical engineering 16* (1992), 313–328.

[50] Kuekes, P. Material implication: Digital logic with memristors. A presentation in the Memristor and Memristive Systems Symposium at UC Berkeley, 2008.

[51] Kuekes, P. J., Robinett, W., Roth, R., Seroussi, G., Snider, G., and Williams, R. S. Resistor-logic demultiplexers for nanoelectronics based on constant-weight codes. *Nanotechnology 17* (2006), 1–10.

[52] Lai, S. Current status of the phase change memory and its future. In *Proceedings of the IEEE International Electron Devices Meetings, 2003* (2003).

[53] Laiho, M., and Lehtonen, E. Arithmetic operations within memristor-based analog memory. In *Proceedings of the 12th International Workshop on Cellular Nanoscale Networks and Their Applications (CNNA) 2010* (2010).

[54] Laiho, M., and Lehtonen, E. Cellular nanoscale network cell with memristors for local implication logic and synapses. In *Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS 2010* (2010), pp. 2051–2054.

[55] Laiho, M., Lehtonen, E., and Lu, W. Memristive analog arithmetic within cellular arrays. In *Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS 2012* (2012).

[56] Laiho, M., Lehtonen, E., Russell, A., and Dudek, P. Memristive synapses are becoming reality. In the newsletter of the Institute of Neuromorphic Engineering, 2010.

[57] Lee, M.-J., Lee, C. B., Lee, D., Lee, S. R., Chang, M., Hur, J. H., Kim, Y.-B., Kim, C.-J., Seo, D. H., Seo, S., Chung, U.-I., Yoo, I.-K., and Kim, K. A fast, high-endurance and scalable non-volatile memory device made from asymmetric Ta_2O_{5-x}/TaO_{2-x} bilayer structures. *Nature Materials 10* (2011), 625–630.

[58] Lehtonen, E., and Laiho, M. Stateful implication logic with memristors. In *Proceedings of the 2009 IEEE/ACM International Symposium on Nanoscale Architectures, NANOARCH 2009* (2009), pp. 33–36.

[59] Lehtonen, E., and Laiho, M. CNN using memristors for neighborhood connections. In *Proceedings of the 12th International Workshop on Cellular Nanoscale Networks and Their Applications (CNNA)* (2010), pp. 1–4.

[60] Lehtonen, E., Laiho, M., and Poikonen, J. H. A chaotic memristor circuit. In *Proceedings of the 12th International Workshop on Cellular Nanoscale Networks and Their Applications (CNNA)* (2010).

[61] Lehtonen, E., Poikonen, J. H., and Laiho, M. A CNN approach to computing arbitrary Boolean functions. In *Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS 2010* (2010), pp. 2295–2298.

[62] Lehtonen, E., Poikonen, J. H., and Laiho, M. Two memristors suffice to compute all Boolean functions. *Electronics Letters*, 3 (2010), 239.

[63] Lehtonen, E., Poikonen, J. H., and Laiho, M. Applications and limitations of memristive implication logic. In *Proceedings of the 13th International Workshop on Cellular Nanoscale Networks and Their Applications (CNNA 2012)* (2012).

[64] Lehtonen, E., Poikonen, J. H., and Laiho, M. Implication logic synthesis methods for memristors. In *Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS 2012* (2012).

[65] Lehtonen, E., Poikonen, J. H., Laiho, M., and Kanerva, P. Memristive associative memories. *submitted to IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2012).

[66] Lehtonen, E., Poikonen, J. H., Laiho, M., and Lu, W. Time-dependency of the threshold voltage in memristive devices. In *Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS 2011* (2011), pp. 2245–2248.

[67] Lehtonen, E., Poikonen, J. H., Poikonen, J. K., and Laiho, M. Grayscale CNN computation of Boolean functions. In *Proceedings of First IEEE Latin American Symposium on Circuits and Systems (LASCAS)* (2010), pp. 200–203.

[68] Liang, J., and Wong, H.-S. P. Cross-point memory array without cell selectors – device characteristics and data storage pattern dependencies. *IEEE Transactions on Electron Devices 57*, 10 (2010), 2531–2538.

[69] Likharev, K., Mayr, A., Muckra, I., and Türel, O. CrossNets: high-performance neuromorphic architectures for CMOL circuits. *Annals Of The New York Academy Of Sciences 1006* (2003), 146–163.

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

[71] Likharev, K. K., and Strukov, D. B. CMOL: Devices, circuits, and architectures. In *Introducing Molecular Electronics*, G. Cuniberti, G. Fagas, and K. Richter, Eds. Berlin: Springer, 2005, pp. 447–478.

[72] Linares-Barranco, B., and Serrano-Gotarredona, T. Exploiting memristance in adaptive asynchronous spiking neuromorphic nanotechnology systems. In *Proceedings of the 9th IEEE Conference on Nanotechnology, IEEE-NANO 2009* (2009), pp. 601–604.

[73] Lorenz, E. N. Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences 20* (1963), 130–141.

[74] MacKay, D. J. C. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

[75] May, R. M. Simple mathematical models with very complicated dynamics. *Nature 261* (1971), 459–467.

[76] Mazumder, P., Kang, S. M., and Waser, R. Memristors: Devices, models, and applications. *Proceedings of the IEEE 100*, 6 (2012), 1911–1919.

[77] McCluskey, E. J. Minimization of Boolean functions. *Bell. Syst. Tech. Journal 35*, 5 (1956), 1417–1444.

[78] Mellor, C. HP's faster-than-flash memristor at least two years away. The Register, 07/09/2012. Available online at http://www.theregister.co.uk/2012/07/09/hp_memristor_and_photons/, 2012.

[79] Micron Technology, Inc. Micron announces availability of phase change memory for mobile devices. A press release, 07/18/2012. Available online at http://investors.micron.com/releasedetail.cfm?ReleaseID=692563, 2012.

[80] Mohr, P., Taylor, B., and Newell, D. Codata recommended values of the fundamental physics constants: 2006. *Reviews of Modern Physics, 80* (2008), 633–730.

[81] Olshausen, B. A., and Field, D. J. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature 381* (1996), 607–609.

[82] Pagiamtzis, K., and Sheikholeslami, A. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits 41*, 3 (2006), 712–727.

[83] Pänkäälä, M., Laiho, M., and Hasler, P. Compact floating-gate learning array with STDP. In *Proceedings of International Joint Conference on Neural Networks, IEEE-IJCNN 2009* (2009), pp. 2409–2415.

[84] Pershin, Y. V., and DiVentra, M. Neuromorphic, digital and quantum computation with memory circuit elements. *Proceedings of the IEEE 100*, 6, 2071–2080.

[85] Pershin, Y. V., and Ventra, M. D. Experimental demonstration of associative memory with memristive neural networks. *Neural Networks 23* (2010), 881–886.

[86] Pickett, M. D., Strukov, D. B., Borghetti, J. L., Yang, J. J., Snider, G. S., Stewart, D. R., and Williams, R. S. Switching dynamics in titanium dioxide memristive devices. *Journal of Applied Physics 106* (2009), 074508–074508–6.

[87] Pierret, R. F. *Semiconductor Device Fundamentals.* Addison Wesley, 1996.

[88] Poikonen, J. H., Laiho, M., and Paasio, A. MIPA4k: A 64x64 cell mixed-mode image processor array. In *Proceedings of the IEEE International Symposium on Cirucits and Systems, ISCAS 2009* (2009), pp. 1927–1930.

[89] Poikonen, J. H., Lehtonen, E., and Laiho, M. On synthesis of Boolean expressions for memristive devices using sequential implication logic. *IEEE Transactions on computer-aided design of Integrated Circuits and Systems* (2012).

[90] Robinett, W., Snider, G. S., Stewart, D. R., Straznicky, J., and Williams, R. S. Demultiplexers for nanoelectronics constructed from nonlinear tunneling resistors. *IEEE Transactions on Nanotechnology 6*, 3 (2007), 280–290.

[91] Roska, T. Cellular wave computers for brain-like spatial-temporal sensory computing. *IEEE Circuits and Systems Magazine, 5*, 2 (2005), 5–19.

[92] Roska, T. Cellular wave computers for nano-tera-scale technology – beyond Boolean, spatial-temporal logic in million processor devices. *Electronics Letters Vol. 43*, 8 (2007), 427–429.

[93] Scharfetter, D. L., Ko, P.-K., and Jeng, M.-C. BSIM: Berkeley short-channel IGFET model for MOS transistors. *IEEE Journal of Solid State Circuits 22*, 4 (1987), 558–566. Transistor models available online at http://www-device.eecs.berkeley.edu/bsim/.

[94] Sinha, A., Kulkarni, M. S., and Teuscher, C. Evolving nanoscale associative memories with memristors. In *Proceedings of the 11th IEEE International Conference on Nanotechnology* (2011), pp. 860–864.

[95] Snaider, J., and Franklin, S. Extended sparse distributed memory. In *Proceedings of the Biological Inspired Cognitive Architectures 2011* (2011).

[96] Snider, G., Amerson, R., Carter, D., Abdalla, H., Qureshi, M. S., Lveill, J., Versace, M., Ames, H., Patrick, S., Chandler, B., Gorchetchnikov, A., and Mingolla, E. From synapses to circuitry: Using memristive memory to explore the electronic brain. *Computer 44*, 2 (2011), 21–28.

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

[98] Snider, G. S. Spike-timing-dependent learning in memristive nanodevices. In *Proceedings of the 2008 IEEE/ACM International Symposium on Nanoscale Architectures, NANOARCH 2008* (2008), pp. 85–92.

[99] Snider, G. S., and Robinett, W. Crossbar demultiplexers for nanoelectronics based on n-hot codes. *IEEE Transactions on Nanotechnology 4*, 2 (2005), 249–254.

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

[101] Strukov, D. B., and Likharev, K. K. Prospects for terabit-scale nanoelectronic memories. *Nanotechnology 16* (2005), 137–148.

[102] Strukov, D. B., and Likharev, K. K. A reconfigurable architecture for hybrid CMOS/Nanodevice circuits. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays* (2006), pp. 131–140.

[103] Strukov, D. B., and Likharev, K. K. Defect-tolerant architectures for nanoelectronic crossbar memories. *Journal of Nanoscience and Nanotechnology 7* (2007), 151–167.

[104] Strukov, D. B., Snider, G. S., Stewart, D. R., and Williams, R. S. The missing memristor found. *Nature 453* (2008), 80–83.

[105] Strukov, D. B., Stewart, D. R., Borghetti, J., Li, X., Pickett, M., Ribeiro, G. M., Robinett, W., Snider, G., Strachan, J. P., Wu, W., Xia, Q., Yang, J. J., and Williams, R. S. Hybrid CMOS/memristor circuits. In *Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS 2010* (2010), pp. 1967–1970.

[106] Strukov, D. B., and Williams, R. S. Exponential ionic drift: fast switching and low volatility of thin-film memristors. *Applied Physics A 94* (2009), 515–519.

[107] Strukov, D. B., and Williams, R. S. Four-dimensional address topology for circuits with stacked multilayer crossbar arrays. *Proceedings of the National Academy of Sciences of the United States of America 106*, 48 (2009), 20155–20158.

[108] Türel, O. Devices and circuits for nanoelectronic implementation of artificial neural networks. PhD dissertation, Stony Brook University, 2007.

[109] Türel, O., Lee, J. H., Ma, X., and Likharev, K. K. Architectures for nanoelectronic neural networks: New results. In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN'04)* (2004), pp. 28–30.

[110] Türel, O., Muckra, I., and Likharev, K. Possible nanoelectronic implementation of neuromorphic networks. In *Proceedings of the International Joint Conference on Neural Networks, 2003* (2003), pp. 365–370.

[111] Vontobel, P. O., Robinett, W., Kuekes, P. J., Stewart, D. R., Straznicky, J., and Williams, R. S. Writing to and reading from a nano-scale crossbar memory based on memristors. *Nanotechnology 20*, 42 (2009).

[112] Whitehead, A. N., and Russell, B. *Principia Mathematica*. Cambridge University Press, 1927 (2nd edition).

[113] Willshaw, D. J., Buneman, O. P., and Longuet-Higgins, H. C. Non-holographic associative memory. *Nature 222* (1969), 960–962.

[114] Xia, Q., Robinett, W., Cumbie, M. W., Banerjee, N., Cardinali, T. J., Yang, J. J., Wu, W., Li, X., Tong, W. M., Strukov, D. B., Snider, G. S., Medeiros-Ribeiro, G., and Williams, R. S. Memristor – CMOS hybrid integrated circuits for reconfigurable logic. *Nano Letters Vol. 9*, 10 (2009), 3640–3645.

[115] Xu, C., Dong, X., Jouppi, N. P., and Xie, Y. Design implications of memristor-based RRAM cross-point structures. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe (DATE) 2011* (2011), pp. 1–6.

[116] Yang, J. J., Pickett, M. D., Li, X., Ohlberg, D. A. A., Stewart, D. R., and Williams, R. S. Memristive switching mechanism for metal/oxide/metal nanodevices. *Nature Nanotechnology, 3* (2008), 429–433.

[117] Yang, Y., Gao, P., Gaba, S., Chang, T., Pan, X., and Lu, W. Observation of conducting filament growth in nanoscale resistive memories. *Nature Communications 3* (2012).

[118] Yi, W., Perner, F., Qureshi, M. S., Abdalla, H., Pickett, M. D., Yang, J. J., Zhang, M.-X. M., Medeiros-Ribeiro, G., and Williams, R. S. Feedback write scheme for memristive switching devices. *Applied Physics A 102* (2011), 973–982.

[119] Young, H. D., and Freedman, R. A. *University Physics with Modern Physics, 12th Edition*. Addison Wesley, 2007.

[120] Zhong, Z., Wang, D., Cui, Y., Bockrath, M., and Lieber, C. Nanowire crossbar arrays as address decoders for integrated nanosystems. *Science 302* (2003), 137–139.