Algorithm Development For An Automated Memristor Control On A Neuromorphic Demonstrator Chip

Wei Zhao

Master's Thesis

presented to

Chair of Electronic Design Automation

TUM School of Computation, Information and Technology

Technical University of Munich

Advisor:	**Amro Eldebiky**
Supervisor:	**Prof. Dr.-Ing. Ulf Schlichtmann**
External Supervisor:	**Neethu Kuriakose**

June 19, 2024

Acknowledgement

First and foremost, I extend my sincere gratitude to Neethu Kuriakose, my dedicated thesis supervisor at FZJ. Her guidance, expertise, and unwavering support were instrumental in the successful completion of this thesis. I am deeply appreciative of her contributions to my academic journey.

I am sincerely grateful to Amro Eldebiky at TUM for his role as my external advisor. His valuable mentorship and assistance in organizing and structuring this thesis were invaluable.

I extend my sincere gratitude to Abdelaziz Ammari and Sabitha Kusuma at ZEA-2 for their generous guidance and assistance throughout my master's thesis. Abdelaziz Ammari has consistently provided support whenever challenges arose. His patience and dedication have significantly accelerated my research progress and deepened my understanding of the subject matter.

Additionally, I would like to express my profound appreciation to Dr. André Zambanini, my team leader at ZEA-2, for his valuable feedback throughout the development of this thesis. Dr. Zambanini has always been there whenever I needed guidance or assistance, allowing me to focus more deeply on the research itself.

I am deeply thankful to Dr. Arun Ashok and Christian Grewing for their assistance in correcting my conference paper and master thesis. Their meticulous review was invaluable in refining my work.

I also wish to thank Prof. Dr. Stefan van Waasen, Director of the Central Institute of Engineering, Electronics, and Analytics (ZEA-2) at Forschungszentrum Jülich GmbH, for providing me with the opportunity to work at the institute during my thesis.

I would particularly like to express my gratitude to Prof. Dr. -Ing. Ulf Schlichtmann, who warmly welcomed me during my internship at TUM in my bachelor studies amidst the COVID-19 pandemic. His encouragement to pursue my studies in France and Germany helped solidify my decision to follow an academic path.

Furthermore, I express my gratitude to Dr. Tsun-Ming Tseng, guided me towards an academic career, provided insights into conducting research, and encouraged me during difficult times. I would not have embarked on my master's journey without his support.

My heartfelt thanks to Sergio Duque Biarge and Dr. Simon Tejero Alfageme, who taught me essential skills for exploring new areas efficiently and demonstrated the importance of teamwork and collaboration.

Lastly, I would like to express my sincere thanks to Yin-Ying Ting, who accompanied me during my bachelor's and master's studies, offering emotional support and helping me gain confidence for the next stage of my journey.

Abstract

Advancements on in-memory technology have positioned memristors at the forefront of non-volatile memory applications, necessitating precise control mechanisms to accurately program memristor cells to their respective states. While commercial memristor applications primarily focus on binary data storage, the unique requirements of neuromorphic computing necessitate the ability to handle a wide range of analog values for synaptic emulation. This thesis explores the utilization of a RISC-V processor and pulse-width modulation generators to configure registers for analog conductance control within a crossbar memristor array architecture. The system incorporates mechanisms to read back the programmed resistance values using an analog-to-digital converter, compare these resistance values to the expected outcomes, and minimize any discrepancies, thereby facilitating accurate voltage and current mode operations.

The core contribution of this research is the development of a flexible and efficient control algorithm, along with an error correction algorithm specifically designed for the RISC-V architecture. By leveraging these algorithms, the system can dynamically adjust and correct the states of memristor cells, ensuring high precision and reliability in memory operations tailored for neuromorphic computing.

To validate the control signals generated by the proposed algorithms, a Universal Verification Methodology Framework testbench is employed. This comprehensive verification process ensures that the control mechanisms are accurate and reliable before proceeding to hardware implementation.

The results from our study indicate significant enhancements in control efficiency, demonstrating the potential for seamless integration of RISC-V processors with memristor technology. This integration paves the way for advanced non-volatile memory solutions that are both robust and highly efficient, marking a notable step forward in the field of memory technology and specifically addressing the needs of neuromorphic computing.

Contents

List of Figures

List of Tables

List of Abbreviations

ADC	Analog-to-Digital Converter
AE	Active Electrode
ALU	Arithmetic Logic Unit
APB	Advanced Peripheral Bus
ASIP	Application-Specific Instruction-set Processor
BFD	Binary File Descriptor
CMOS	Complementary Metal-Oxide-Semiconductor
DAC	Digital-to-Analog Converter
DUT	Device Under Test
EDA	Electronic Design Automation
GNU	GNU's Not Unix
GUI	Graphical User Interface
HRS	High-Resistance State
JART	Joint Advanced Resistor Technology
JTAG	Joint Test Action Group
LRS	Low-Resistance State
LSB	Least Significant Bit
MSB	Most Significant Bit
MUX	Multiplexer
OE	Ohmic Electrode
PWM	Pulse Width Modulation
RADAR	Range-Dependent Adaptive Resistance
RRAM	Resistive Random Access Memory
RISC-V	Reduced Instruction Set Computer-V
RTL	Register-Transfer Level
SAR	Successive Approximation Register
SBA	Sigma-Based Allocation
SRAM	Static Random Access Memory
UVMF	Universal Verification Methodology Framework
VMM	Vector-Matrix Multiplication

1 Introduction

Memristors, with their controllable conductivity, non-volatile nature, and high-density properties, hold significant promise for advancing memory technology and neuromorphic computing. However, challenges such as achieving uniform performance across multiple conductance states and ensuring reliability remain substantial obstacles. Additionally, the current state-of-the-art predominantly focuses on methods for programming the cells, neglecting the development of an automatic algorithm for memristor control following the readout process. Consequently, there is a critical need to develop such an automatic mechanism to fully leverage the potential of memristors.

This control algorithm is developed within an integrated microcontroller based on the RISC-V architecture. This framework allows for the configuration of specific registers to efficiently manage three distinct conductance control modes: voltage mode, current mode, and variable mode, thereby ensuring precise operation of the memristor array. The RISC-V model was generated using Synopsys' ASIP Designer CHESSDE, enabling the compiler to assign memory addresses based on the hardware design. The control system is divided into two components: a custom digital block that generates various pulse-shaped waveforms for current and voltage, and an algorithmic component comprising subroutines that supervise the digital control blocks. To ensure accuracy in the programming, an error correction algorithm is introduced to address any discrepancies between the expected state value and the actual value. Following this, a Universal Verification Methodology Framework (UVMF) testbench is employed to verify the correctness of the control signals, ensuring their accuracy and reliability before proceeding with hardware implementation.

The background and fundamentals of memristor technology are explored in the first place (Chapter 2), covering their properties, types, and applications. It also discusses the basic operations involved in memristor analog resistance programming and the control algorithms for both voltage and current modes. The error correction algorithms essential to ensure the reliability of memristor operations are discussed.

Next, the methodology for developing the control algorithm is detailed (Chapter 3). This includes the top-level architecture of the control system, the design of mixed-signal control blocks, and the implementation of digital and analog control circuits. The integration of the control algorithm with the RISC-V processor and

the use of Synopsys tools for design and verification are also described.

Then, the simulation setup and results are presented (Chapter 4), explaining the use of UVMF testbench to validate the control signals generated by the proposed algorithms. The simulation flow using Questa Sim and the verification process for both digital and analog blocks are covered, demonstrating the effectiveness of the control algorithm.

The analysis of the simulation results and the verification of the memristor control program's correctness are then discussed (Chapter 5 and 6). Finally, a summary and conclusion of the work are provided (Chapter 7), highlighting the key findings and contributions. Future research directions and prospects for further development of automated memristor control mechanisms are also discussed.

2 Technical and Background

2.1 Memristor Technology: Detailed review of memristor properties, types, and applications

2.1.1 Memristor Properties

Memristors, a blend word for memory resistors, were initially proposed by Leon Chua in 1971 as the fourth fundamental circuit element [1], complementing resistors, capacitors, and inductors. Despite this early theoretical foundation, it was not until 2008 that a practical memristor was realized by a team led by Stanley Williams at HP Labs [2]. Memristors are distinguished by their capability to maintain a resistance state based on the history of electrical flux that determines their conductivity. This unique property opens up promising applications in non-volatile memory, particularly in the form of resistive random-access memory (RRAM), as well as in neuromorphic computing and other advanced electronic systems.

Memristors have the unique ability to switch between different resistance states when a specific threshold voltage or current is surpassed, a property due to the movement of ions or defects within the material. This behavior is crucial for applications in digital memory and logic circuits for several reasons.

Firstly, memristors are ideal for non-volatile memory because they retain their resistance state even when power is turned off, allowing for high-density data storage that significantly increases the capacity of memory devices while maintaining or reducing physical size. Additionally, memristors enable stateful logic operations, performing computations directly within the memory, which enhances speed and efficiency by reducing the need for data transfer between memory and the processor. This is particularly important in the context of in-memory computing, which addresses several critical needs in modern computing. In-memory computing significantly reduces latency, enhances system performance, and achieves substantial energy savings by minimizing data movement [3].

Furthermore, their dynamic resistance switching supports reconfigurable logic circuits that can be programmed and reprogrammed on the fly [4], adding flexibility and adaptability to various computing tasks. The fast switching capabilities of memristors [5], often occurring in the nanosecond range, make them suitable for applications requiring rapid data access and processing.

The most significant advantages of the memristor are summrized in Figure 2.1.

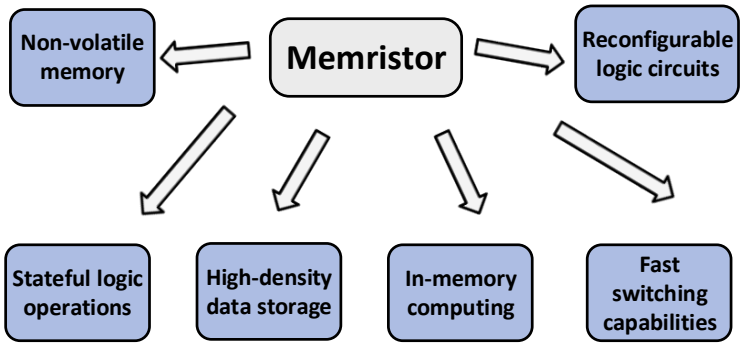
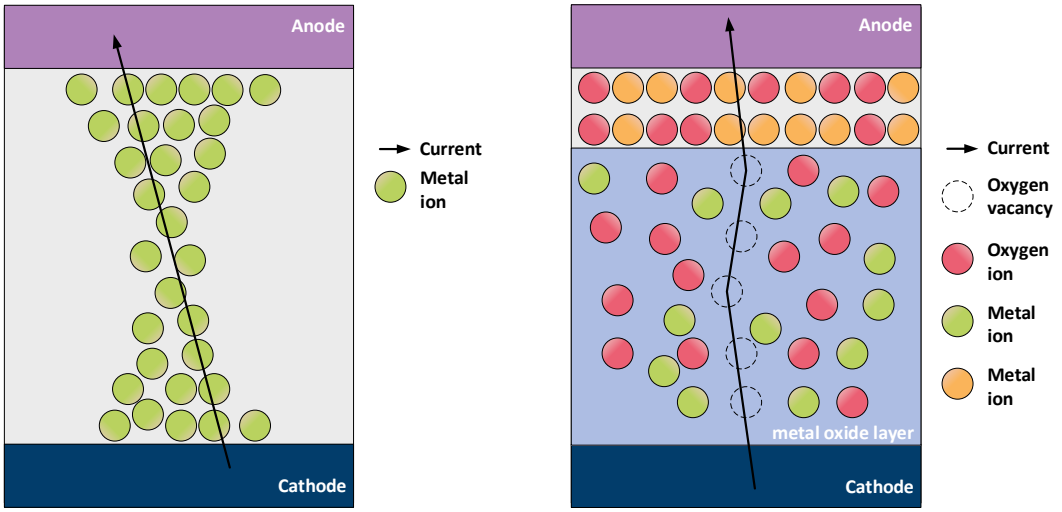


Figure 2.1: Advantages of the memristor

2.1.2 Types of Memristors

Memristors can be broadly categorized based on their underlying mechanisms. In our regard, we focus on metal ion-based [6] and oxygen vacancy filament-based [7] memristors. Metal ion-based memristors operate through the migration of metal ions within the device as illustrated in Figure 2.2a. When a voltage is applied, metal ions such as silver or copper migrate through a solid electrolyte and form a conductive filament between the electrodes, changing the device's resistance state. These memristors typically use metals like silver or copper as one of the electrodes and solid electrolytes such as silver sulfide or copper selenide. The formation and dissolution of metallic filaments control the switching between high and low resistance states. Metal ion-based memristors are known for their high speed due to rapid ion migration, scalability to very small dimensions, and high on/off ratio, making them ideal for non-volatile memory applications and programmable logic devices.

On the other hand, oxygen vacancy filament-based memristors rely on the movement of oxygen vacancies within a metal oxide layer to alter resistance illustrated in Figure 2.2b. When voltage is applied, oxygen ions migrate, leaving behind vacancies that form a conductive filament. Common oxide materials used include titanium dioxide, hafnium oxide, or tantalum oxide. The switching behavior in these memristors is driven by the creation or annihilation of conductive filaments composed of oxygen vacancies. Oxygen vacancy filament-based memristors are valued for their stability, excellent data retention, and compatibility with existing CMOS technology, making them widely used in RRAM and neuromorphic computing. These devices mimic synaptic functions in artificial neural networks due



(a) Metal ion-based [6] (b) Oxygen vacancy filament-based [7]

Figure 2.2: Types of memristors in LRS

to their ability to gradually change resistance states, contributing significantly to the development of advanced electronic systems.

2.1.3 Memristor Applications

The traditional von Neumann architecture, which has been the foundation of most computing systems since its inception, separates memory and processing units. This separation leads to the "von Neumann bottleneck," where system performance is limited by the bandwidth and speed of data transfer between the Central processing unit (CPU) and memory. This bottleneck hinders the efficiency and scalability required for advanced computing tasks, especially those involving large datasets and real-time processing.

In contrast, neuromorphic computing seeks to emulate the neural architecture of the human brain within artificial systems, integrating computation in memory. This approach allows for more efficient and parallel processing. Memristors hold significant promise for advancing neuromorphic computing due to their unique capability to adjust their resistance states in response to voltage changes, effectively mimicking the weights of synapses [8, 9]. This functionality facilitates learning processes in artificial neural networks, enabling the creation of highly efficient, adaptive learning systems that can process and store information in a manner akin to the human brain.

A critical advantage of memristors in neuromorphic computing is their ability to

perform in-memory computing [3]. This means that data storage and processing occur in the same physical location, significantly reducing the need for data transfer between separate memory and processing units. Such integration leads to faster processing speeds and enhanced overall system efficiency, both of which are crucial for real-time data processing in neuromorphic applications.

2.2 Memristor Control Mechanisms

The fundamental operations and control algorithms for memristor programming will be explored in the section, focusing on analog resistance programming, voltage and current control strategies, and the implementation of error correction algorithms. The discussion includes the mechanisms for setting and resetting memristors, the application of various control methods to achieve desired resistance states, and the correction techniques used at the cell, array, and arithmetic levels to ensure accurate and reliable memristor operation.

2.2.1 Basic Operations of Memristor Analog Resistance Programming

Each memristor is equipped with two ports, Active Electrode (AE) and Ohmic Electrode (OE). In this work, OE is connected to the ground. When a positive voltage is applied to the AE, the reset process initiates, tuning the memristor to the high-resistance state (HRS). Conversely, when a negative voltage is applied, the set process activates, adjusting the corresponding memristor to the low-resistance state (LRS).
Electroforming, set, reset, and read operations are critical processes in the functioning of memristors [10].
Electroforming is the initial process where a high voltage of 3 V is applied to create a conductive filament within the memristor, typically involving the migration of ions or vacancies that form a path between the electrodes. This step is essential to activate the memristor for subsequent switching operations.
The set operation in memristors is generally more linear compared to the reset operation [11]. This distinction arises from the different mechanisms involved in each process. The set operation typically involves the formation of conductive filaments within the memristive material, which tends to follow a more predictable, linear behavior as the resistance changes from HRS to LRS, often used to program multilevel states. Conversely, the reset operation, which involves the dissolution or disruption of these filaments, is often more abrupt and nonlinear due to the complex dynamics of filament breakage and the variability in how the material responds to the applied voltage. Additionally, the read operation shares similar

settings with the reset operation but employs a smaller voltage to avoid changing the resistance state of the memristor cells.

The memristor model used in this thesis is the deterministic version of the JART VCM v1b model [11]. The amplitudes (Amp) and pulse widths for both voltage and current modes, as illustrated in Table 2.1.

Table 2.1: The condition of voltage and current pulses

	Reset	**Set**	**Read**
V mode	Amp: 1.5 V Pulse width: 10 ns - 1 ms	Amp: −0.75 V Pulse width: 10 ns - 1 ms	Amp: 250 mV Pulse width: 10 ns - 1 ms
I mode	Amp: 1.5 V Pulse width: 10 ns - 1 ms	Amp: 5.2 µA - 250 µA Pulse width: 1 ms - 500 s	Amp: 250 mV Pulse width: 10 ns - 1 ms

2.2.2 Voltage and Current Control Algorithm

Memristors in voltage control alter their resistance state based on the amplitude and duration of the applied voltage. In this thesis, these voltages are applied in the form of pulses. There are three primary methods for programming memristor cells as illustrated in Figure 2.3. The first method employs a fixed-voltage set and reset pulses algorithm [12], wherein pulses are applied repeatedly until the cell resistance falls within the target range. The second method utilizes an incremental step pulse width algorithm [13], where the pulse width is progressively increased with each iteration. Finally, the third method involves adjusting the pulse amplitude, following a similar strategy as incremental step pulse width algorithm[13].

Current controlled memristors change their resistance state based on the magnitude and duration of the applied current [14]. Similar to voltage controlled memristors, the application of a current exceeding a specific threshold induces changes in the internal structure. The current control may provide more linear transitions than the voltage control in set operation in certain memristor materials [15], thus becoming a trend while setting the memristor cells. The current mode differs from the voltage mode in handling low resistance tuning. Before performing a write operation in low resistance mode, the memristor must be switched to HRS using the voltage mode mechanism.

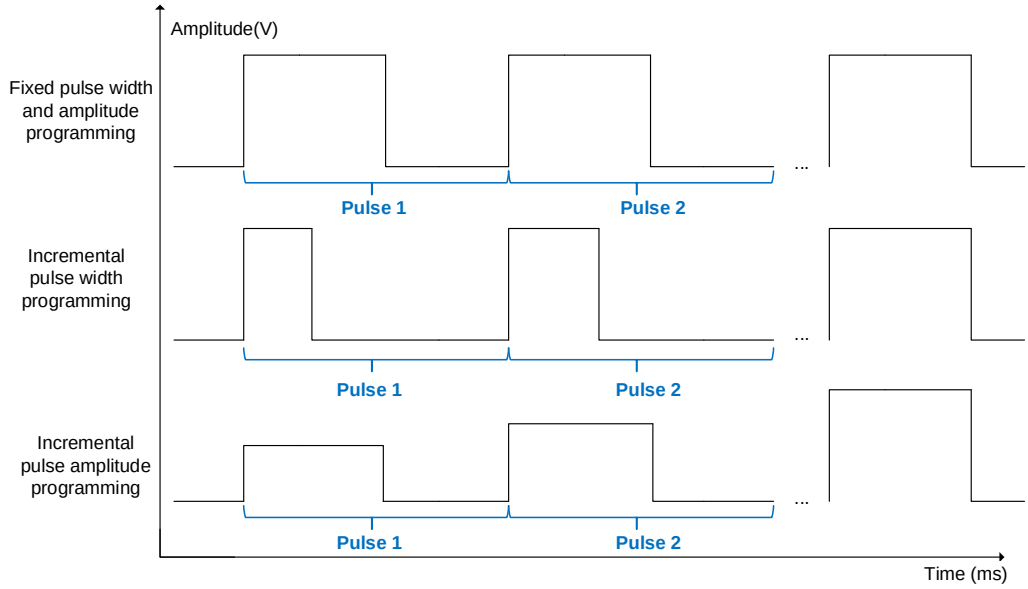


Figure 2.3: Different voltage control algorithms

2.3 Error Correction Algorithm

To address the discrepancy between the expected and actual multi-bit levels of a memristor, an error correction algorithm is utilized. This algorithm operates on three levels: the cell level, the memristor array level, and the arithmetic level. Each level targets specific aspects of the correction process, ensuring that the memristor functions accurately and reliably within its intended applications.

2.3.1 Multi-bit Level Correction Programming

Multi-bit storage in memristors extends beyond the traditional HRS and LRS by incorporating intermediate resistance states. This allows for higher-density storage, as each cell can store more information. In the digital domain, this means the stored information is not confined to binary 0 and 1, but can include multiple levels, such as values from 0 to 3 or even up to 127 [16].

Range-Dependent Adaptive Resistance (RADAR) tuning facilitates the fast and energy-efficient programming of multi-bit per cell RRAM arrays [17] as illustrated in Figure 2.4. It employs a combination of coarse-and-fine-grained resistance tuning. When the desired state is significantly different from the current state, a coarse control pulse with a large amplitude is used to quickly adjust the resistance close to the target range. Subsequently, fine control pulses are applied to precisely

reach the desired bit state.

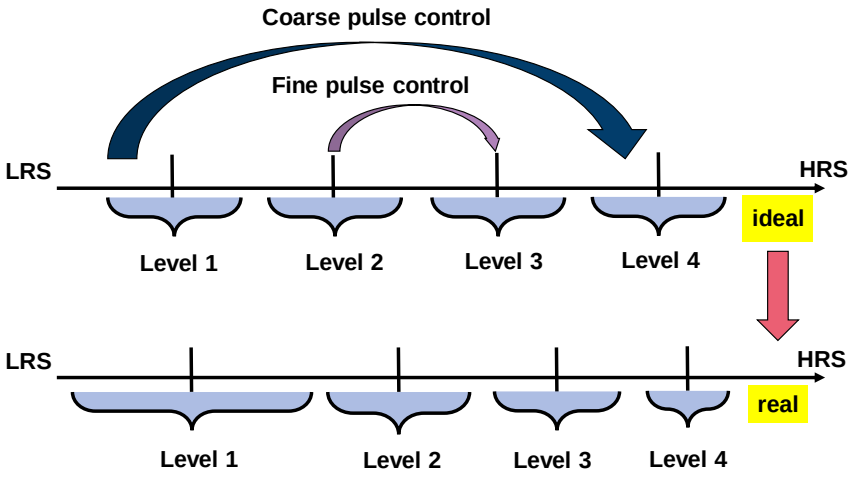


Figure 2.4: Error correction algorithm on the cell level

Ideally, the resistance ranges would be evenly distributed. However, in practice, the variance in cell resistance values tends to be higher for cells with higher resistance. The Sigma-Based Allocation (SBA) technique [18] addresses this by using non-uniform resistance ranges while maintaining consistent sensing gaps between these ranges. This approach ensures more reliable and accurate multi-bit storage in memristors.

2.3.2 Array Level Correction Programming

In memristor-based storage systems, error correction at the cell level differs significantly from that at the array level. One primary challenge is identifying erroneous cells within the array. Additionally, while programming one device, the resistance of other devices in the array may drift from their target values [19], complicating the programming process. Furthermore, accurately reading the conductance of an individual cell requires highly precise read out circuitry [19]. These factors are critical when developing effective programming strategies for memristors.

A representative approach to addressing these issues is Gradient Descent-Based Programming (GDP) [19]. The GDP method mitigates these challenges by directly minimizing the Vector-Matrix Multiplication (VMM) error using gradient descent with synthetic random input data as illustrated in Figure 2.5. The figure shows the

operation of a crossbar array in an Analog In-Memory Computing (AIMC) core, where VMM is performed. The crossbar array consists of horizontal lines, representing the input rows that receive voltage pulses, and vertical lines, representing the output columns that collect the resulting current pulses. At each intersection of the horizontal and vertical lines, there is a unit-cell, which modulates the conductance level. The conductance levels, indicated by the color gradient in the figure, range from low (light colors) to high (dark colors), corresponding to the weight of each unit-cell in the VMM operation.

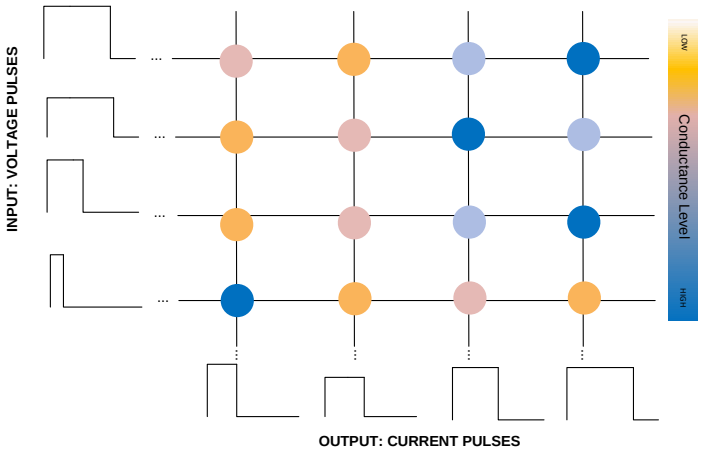


Figure 2.5: Error correction algorithm on the array level

The input voltage pulses are applied across the rows, and depending on the conductance of the unit-cells, currents accumulate along the columns. These accumulated currents are then read out as output current pulses.

Rather than focusing on individual unit-cell conductance, GDP optimizes overall VMM accuracy. The process involves initializing the conductances, performing batched VMMs, quantifying the VMM error as a loss function, and calculating gradients to iteratively adjust the programming pulses. This approach updates all unit cells in parallel, thereby eliminating the divergence issue encountered in conventional methods. Furthermore, by focusing on minimizing the VMM error rather than the conductance of individual cells, GDP avoids the need for highly precise conductance readings, which are typically challenging due to the limitations of Analog-to-Digital Converters (ADC). As a result, this method significantly enhances the programming efficiency and accuracy, particularly in systems with high-resistance memristive devices where conventional iterative methods struggle with accuracy due to low conductance values.

2.3.3 Arithmetic Level Error Programming

The arithmetic code offers an alternative correction algorithm [20]. Traditional Hamming-code-based error correction is ineffective for these circuits due to the analog nature of their computations. In arithmetic coding, data obtained from the ADC output is encoded by multiplying it by an integer A, which represents the expected output. During processing, the encoded data is evaluated by checking the residue of (N%A), where N denotes the measurement result. A non-zero residue indicates an error, as it suggests that the computation result is not correctly divisible by the predefined integer A. An error correction table is then constructed by matching residues with the most likely arithmetic errors [20], enabling effective correction based on error probabilities.

The Table 2.2 illustrates an error-correcting scheme with $A = 11$. Each row represents a possible residue (N%A) when dividing the erroneous value by 11. The "Arithmetic error" column specifies the corresponding arithmetic error in terms of powers of 2, indicating the magnitude and direction of the error. The notation $B_i : \pm 1$ indicates which bit position (B) should be corrected and whether it should be incremented (+1) or decremented (-1). This approach allows for efficient identification and correction of errors based on the residue of the division.

Table 2.2: Error correcting table. A=11

N%A	**Arithmetic error**		
1	$+2^0 = +1$	$\Rightarrow$	$B_0 : +1$
2	$+2^1 = +2$	$\Rightarrow$	$B_1 : +1$
3	$-2^3 = -8$	$\Rightarrow$	$B_3 : -1$
4	$+2^2 = +4$	$\Rightarrow$	$B_2 : +1$
5	$+2^4 = +16$	$\Rightarrow$	$B_4 : +1$
6	$-2^4 = -16$	$\Rightarrow$	$B_4 : -1$
7	$-2^2 = -4$	$\Rightarrow$	$B_2 : -1$
8	$+2^3 = +8$	$\Rightarrow$	$B_3 : +1$
9	$-2^1 = -2$	$\Rightarrow$	$B_1 : -1$
10	$-2^0 = -1$	$\Rightarrow$	$B_0 : -1$

2.4 ASIP Designer Software Introduction

CHESSDE is a tool in electronic design automation (EDA). It translates high-level languages like C into hardware descriptions. These descriptions are suitable

for processors based on architectures like RISC-V. The Graphical user interface (GUI) of the integrated development environment CHESSDE is shown in Figure 2.6. High-level programming code can be converted into a hardware design that can be synthesized, simulated, and implemented on a chip. Additionally, CHESSDE can optimize the processor model structure based on feedback from simulations and implementations [21]. This optimization process ensures that the design meets the required specifications and performance criteria, enhancing the efficiency and functionality of the final hardware product. Essentially, CHESSDE bridges the gap between software and hardware by enabling the design, verification, and testing of complex integrated circuits and systems.

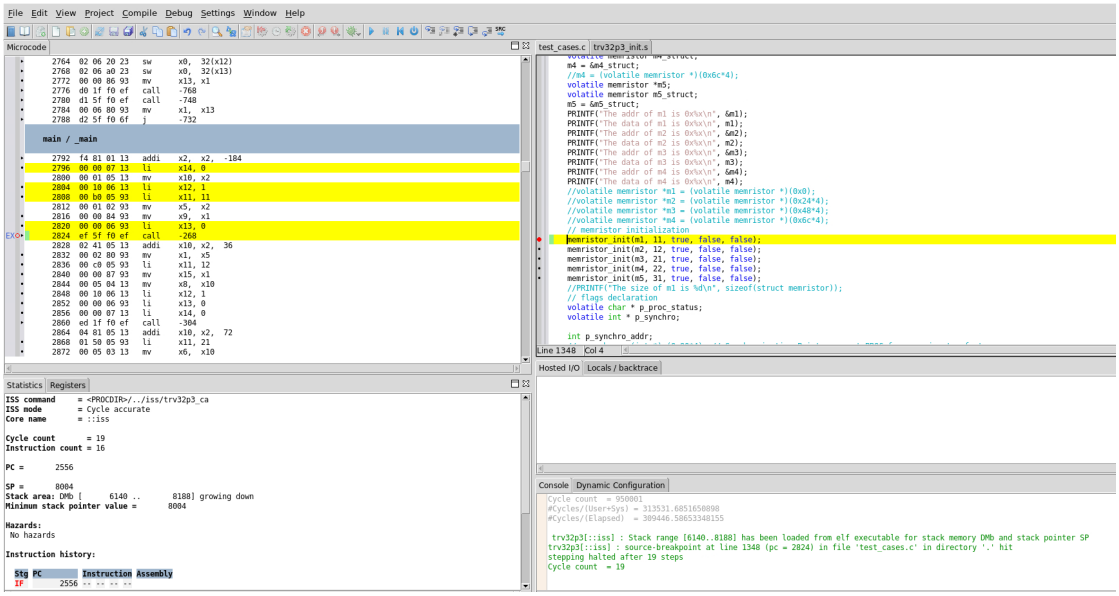


Figure 2.6: The GUI of CHESSDE

2.5 Universal Verification Methodology Framework

UVMF is a comprehensive and modular verification methodology widely used in hardware design and verification. UVMF promotes a structured approach where different verification tasks are handled by specific components, as illustrated in Figure 2.7.

Key components of the UVMF testbench architecture include:

- **Sequence**: Generates stimuli for the verification environment, defining the order and conditions under which various transactions or operations are sent to the Device Under Test (DUT). In this work, the sequence focuses on

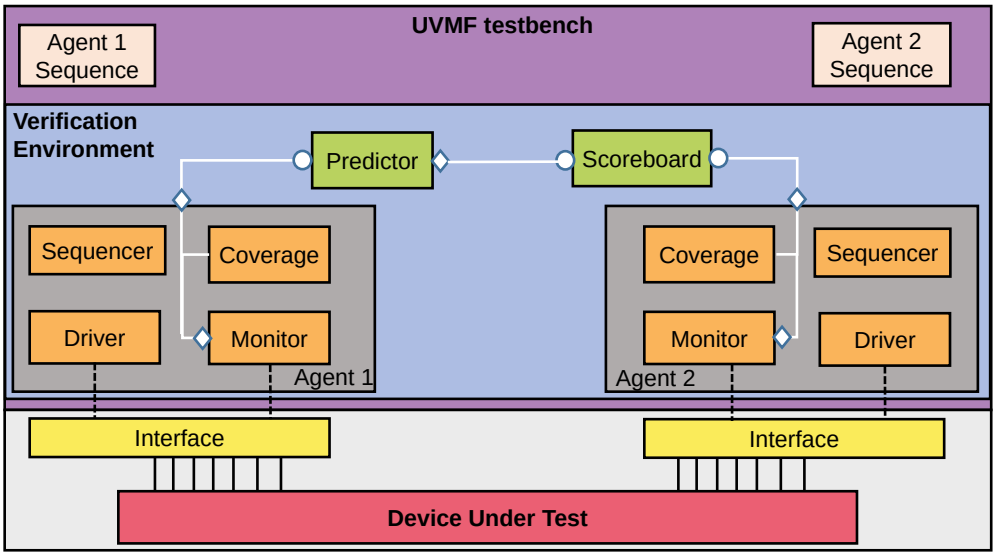


Figure 2.7: Concept of a UVMF Test Bench Architecture

controlling the memristor, managing resets, configuring the clock, and polling mechanisms to verify the DUT's readiness and status.

- **Sequencer**: Manages the flow of sequences, sending the generated stimuli from the sequence to the driver. It acts as an intermediary, ensuring the correct delivery of sequence items.

- **Driver**: Converts high-level sequence items from the sequencer into low-level transactions that can be sent to the DUT through the virtual interface.

- **Monitor**: Passively observes the signals from the DUT, capturing responses and converting them into a format that can be analyzed by the rest of the testbench.

- **Virtual Interfaces**: Connect the physical interface of the DUT to verification components like drivers and monitors.

- **Predictor**: Acts as a model that simulates the expected behavior of the DUT. It generates predicted responses based on the stimuli provided and compares them to the actual outputs of the DUT. This helps in identifying discrepancies between the expected and actual behavior early in the verification process.

- **Scoreboard**: Checks the correctness of the DUT's behavior by comparing the actual output (captured by the monitors) against the expected output generated by the Predictor.
- **Functional Coverage**: Measures how much of the design's functionality has been exercised by the testbench.

Verification components created using UVMF are reusable across different projects and levels of the design hierarchy, enhancing the efficiency of the verification process.

Usage of Connection Points:

- **Round Connection Points**: These are interface points that require a function to be implemented. They provide a local parameter that allows the function to be called.
- **Rhombic Connection Points**: These are interface points that implement functions which are called from round connection points (ports) and can also pass through different levels of the hierarchy.

Verification components created using UVMF can be reused across different projects and different levels of the design hierarchy. This reduces the effort required to develop new testbenches and enhances the efficiency of the verification process.

3 Methodology

3.1 Top Level Architecture of the Demonstrator Chip

The control algorithm will be applied to the Dolphin DAD (NT2DAD1) chip fabricated using 28 nm bulk CMOS technology [22]. The top-level architecture, based on [23] is illustrated in Figure 3.1. The fifth memristor is included to test different combinations of current compliance transistor widths with resistances and is used only in the variable mode.

This architecture serves as a foundational prototype for developing memristor interfaces before integrating them into a more complex and costly chip that will host significantly longer arrays of memristors. The standalone prototype allows us to refine and validate the interfaces prior to their incorporation into the larger system.

The DIG_TOP module integrates several critical components: Static random-access memory (SRAM), RISC-V processor with trv32p3 model, and Joint Test Action Group (JTAG) Interface [23]. The SRAM primarily stores instructions and data from the memristor control code, while the RISC-V processor executes control algorithms and manages configurations. External access is facilitated by the JTAG interface, which handles commands for reading and writing configuration settings independently of the RISC-V, as well as reading status information.

A dedicated MEMCTRL_CTRL block manages the configuration and control signals applied on the memristors, CONFIG_MCTRL handles register configurations, and the pulse-width modulation (PWM) module is tasked with generating write and read pulses. ADC_CTRL manages the 2-channel ADC for the first and second columns of the memristor array. These signals are handled by the Analog Memristor Control Block, which provides control to the memristor array. Finally, the output current from the memristor array is converted by the ADCs and transmitted to the MEMCTRL_CTRL block to supply essential feedback information.

Communication and control are further enhanced by the customized Advanced Peripheral Bus (APB) Interface [23], which enables data transfer between the processor and the configuration registers.

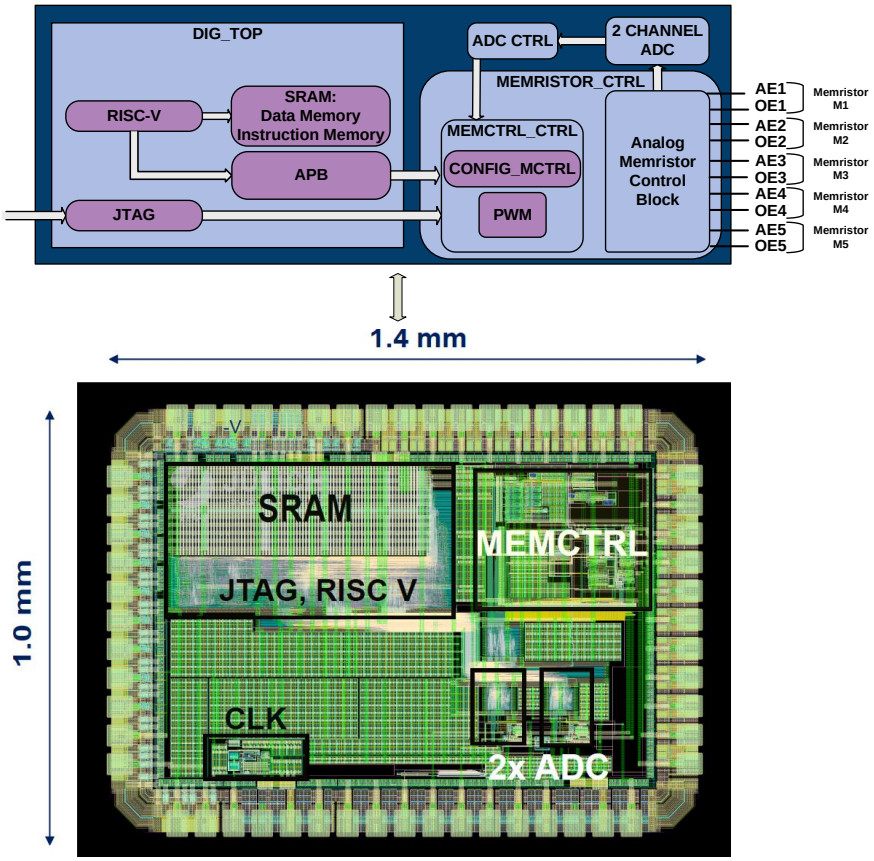


Figure 3.1: System architecture for a chip controlling a 2×2 memristor array and the chip's layout that went into production [22]

3.2 RISC-V: trv32p3 Model Description

The trv32p3 is a 32-bit RISC-V processor model designed as part of the ASIP workflow to provide a balance between flexibility and efficiency. The architecture is based on a modular design that includes a RISC-V core, which can be customized through ASIP Designer to meet specific application requirements. The processor integrates essential components such as the Arithmetic logic unit (ALU), Address Generation Unit (AGU), and APB to interact with peripheral modules like memory and configuration registers.

3.2.1 Memory Utilization and Addressing

The trv32p3 processor features a structured memory interface that allows it to handle both program and data memory effectively. The memory is divided into

Instruction Memory (IM) and Data Memory (DM), each serving distinct functions in the processor's operation. The IM is primarily used to store the instructions that the processor executes, while the DM is used for storing variables and intermediate data generated during computation.

Memory addressing in trv32p3 is managed through the AGU, which is responsible for calculating memory addresses for load/store instructions. The AGU ensures that the correct memory locations are accessed during program execution, facilitating efficient data retrieval and storage. The processor supports a range of addressing modes, including immediate, register-indirect, and base-offset addressing, which provide flexibility in how data is accessed and manipulated.

3.2.2 Programming of the trv32p3 Processor

Programming the trv32p3 model involves writing code in a language compatible with the RISC-V instruction set, typically in C. The compiled code is then loaded into the IM, where it is executed by the processor. The trv32p3 can be programmed to perform a variety of tasks, from simple arithmetic operations to more complex memristor control functions, depending on the application's requirements. To program the processor for specific tasks, developers can leverage the ASIP Designer's tools to define custom instructions and optimize the processor's instruction set for their application.

3.2.3 Control Blocks from the Program

The trv32p3 processor interacts with various control blocks through the APB. To address these control blocks from within a program, specific memory-mapped addresses are assigned to each control block. The processor can access these addresses using load and store instructions, allowing the program to configure peripheral devices or read their status. For instance, to write a value to a configuration register, the program would store the value at the corresponding memory-mapped address of that register. Similarly, to read the status of a peripheral, the program would load the value from the appropriate address into a register for processing.

3.3 Top Level Algorithm Development

The depicted algorithm as shown in Figure 3.2 begins with the initialization phase, setting up essential configurations such as the pulse parameters for the system. This is followed by the electroforming step, which conditions the memristors for optimal functionality. Subsequently, the algorithm performs memristor selection to select the desired memristor for operation. At the read/write decision point, the

algorithm distinguishes between read and write operations. For read mode, the process continues directly to read resistance, where the memristor's resistance is measured. The algorithm configures the appropriate voltage, current, or variable mode for write operations and performs the reset/set operation. The variable mode is further divided into variable voltage and variable current modes, where only the transistor widths are modified. However, the underlying mechanism remains the same as in voltage and current modes. After the writing process, the system reads the resistance to verify if the desired state has been achieved. If the desired state is not reached, the write-read sequence is repeated until the target state is obtained. Once the target state is reached, the process concludes.

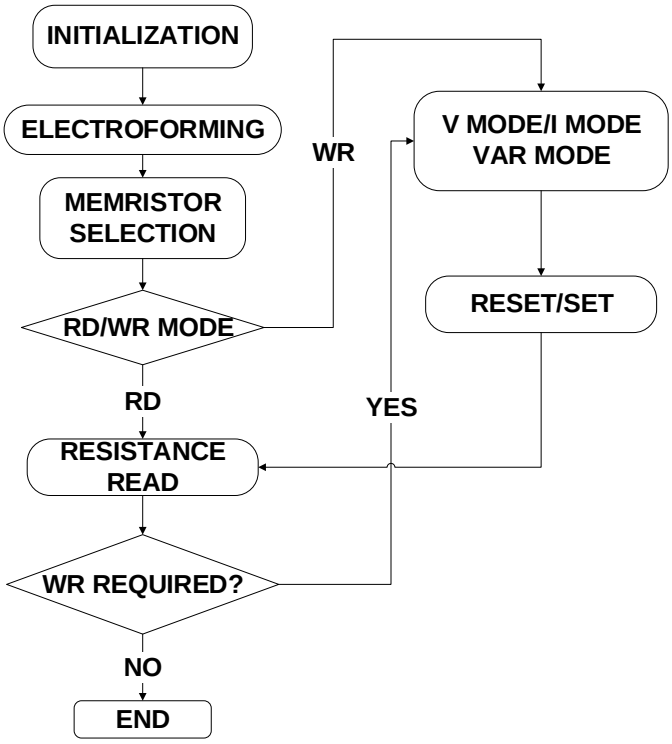


Figure 3.2: Control algorithm flowchart

3.4 Mixed Signal Control

The mixed-signal control system in this work is primarily divided into two components: memristor control and ADC control as shown in Figure 3.3. Each successful write or read operation of the selected memristor requires precise register config-

uration on both the analog and digital sides of the memristor and ADC control systems. The memristor control is further subdivided into analog and digital memristor control. In the digital block, PWM signals are generated to control pulse width, while in the analog block, an operational amplifier (opamp) ensures the pulses achieve the necessary amplitude. To facilitate the selection of memristors and their respective operational modes, the multiplexer (MUX) selects memristors across both rows and columns, as well as their operational modes. The ADC control is responsible for measuring the total current resulting from each column.

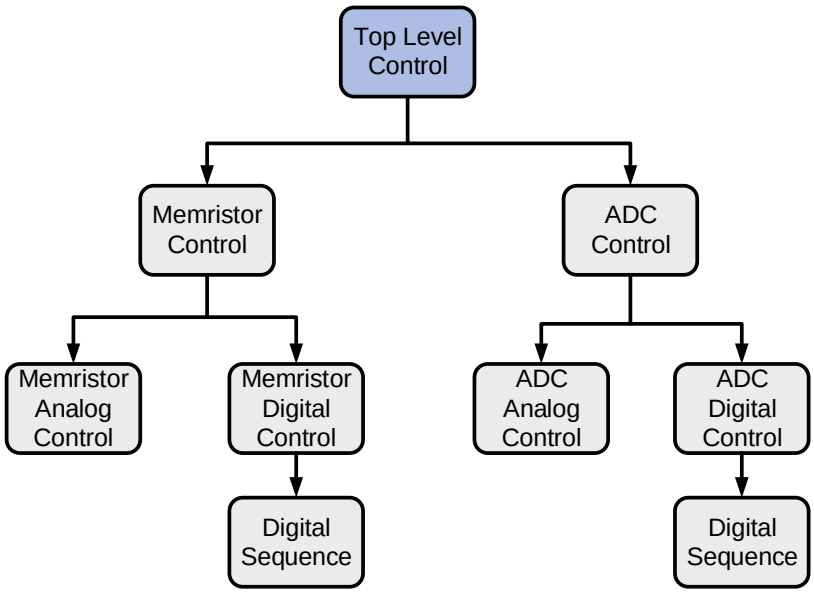


Figure 3.3: The overview of the mixed-signal control system

The various registers utilized in the mixed control algorithm are presented in Table 3.1 below.

Table 3.1: Register types and their purposes

Registers	Count	Purpose
Analog ADC registers	14	Enable and select measurement ports
Digital ADC registers	4	Select test mode and work mode of ADC
Analog memristor control registers	16	Select memristors and modes of operation
Digital memristor control registers	12	Control the pulse width

3.4.1 ADC Control Block

The analog and digital control blocks of the ADC are discussed in this subsection, focusing on protection during memristor operations and the management of test and working modes to ensure accurate data handling.

3.4.1.1 Analog ADC Control Block

The analog circuit of the ADC is shown in Figure 3.4. While the memristor control operation requires a 1.8V power supply for the reset and set processes, the ADC operates at a lower voltage of 0.9V VDD. To isolate the ADC from the higher voltage used by the memristor circuitry and prevent any potential damage, a switch is implemented, as indicated by the red line in the figure. This switch effectively separates the two power domains, ensuring that the ADC is only exposed to the lower voltage. This protective pulse applied via the switch should have a slight delay compared to the read pulses but share the same period. By utilizing this switch structure, the ADC can be designed using smaller and faster core devices rather than larger IO devices, resulting in significant power savings by operating the ADC at 0.9V instead of 1.8V.

The ADC control circuit plays a crucial role in its overall functionality, particularly in ensuring precise and reliable measurements through the configuration of 14 ADC related registers. These configurations primarily enable the ADC's analog operation and allow for the selection between internal and external input ports for measurements.

It is important to note that the external control circuitry for the ADC is used primarily for calibration purposes, rather than running the ADC itself. The ADC contains an internal digital circuitry that executes the Successive Approximation Register (SAR) algorithm. This internal circuitry is responsible for the ADC's core operation, while the external control influences the SAR DAC during calibration to optimize accuracy and performance. The SAR DAC within the ADC approximates the digital input value in successive steps, using a combination of a binary search

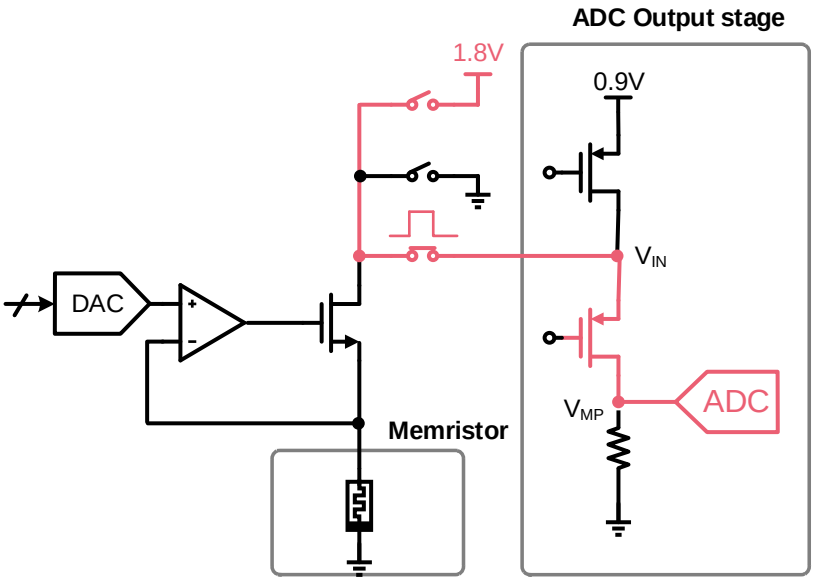


Figure 3.4: The analog circuit of ADC, the red pulse in the figure refers to the protective pulse that is applied to the switch.

algorithm, digital registers, and a comparator to achieve the desired precision efficiently.

3.4.1.2 Digital ADC Control Block

The ADC control operates in two modes: test mode, used for verifying the operational status of the ADC, and working mode, which is used to read current information from the memristor array.
The ADC control system in test mode depicted in Figure 3.5 integrates two ADCs with digital processing through an ADC control block, utilizing a FIFO buffer for data management. Initially, the ADC control is enabled, followed by activating the test write mode to test data writing processes. Sample data is then written through the FIFO buffer. Once the writing process is complete, the test write mode is disabled. Subsequently, the system enters the test read mode to prepare for reading the written data from the FIFO. The read operation from the FIFO is enabled, allowing the digital processing unit to access the captured data. Finally, all test modes are disabled, marking the completion of the test process and ensuring the system is ready for regular operation.

When the ADC is in working mode, it will determine whether it depends on the reading pulses from the memristor control as shown in Figure 3.6. The FIFO will

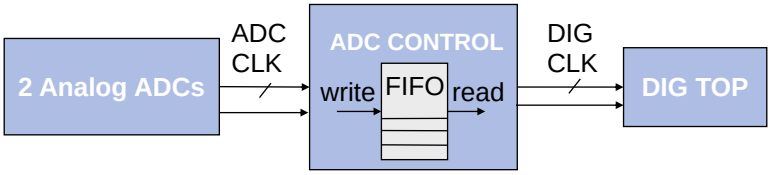


Figure 3.5: ADC digital control: test mode

then operate in one of two modes. If both ADC channels are active, the one-sample mode will be enabled to ensure quicker data transfer.

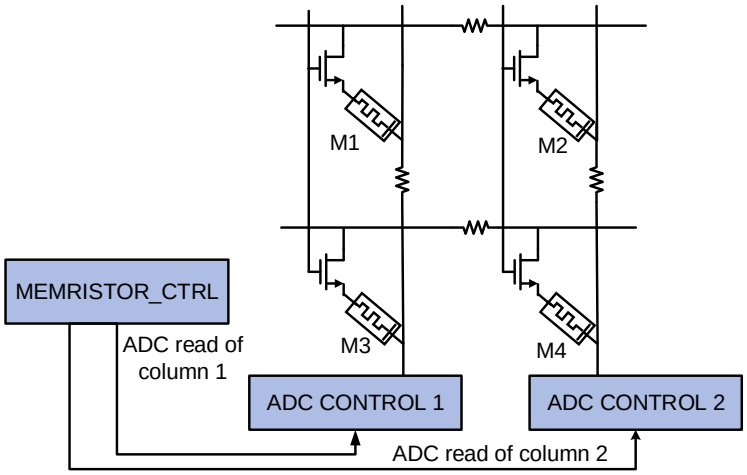


Figure 3.6: ADC digital control: work mode

The sequence of the ADC digital control is straightforward as shown in Figure 3.7.

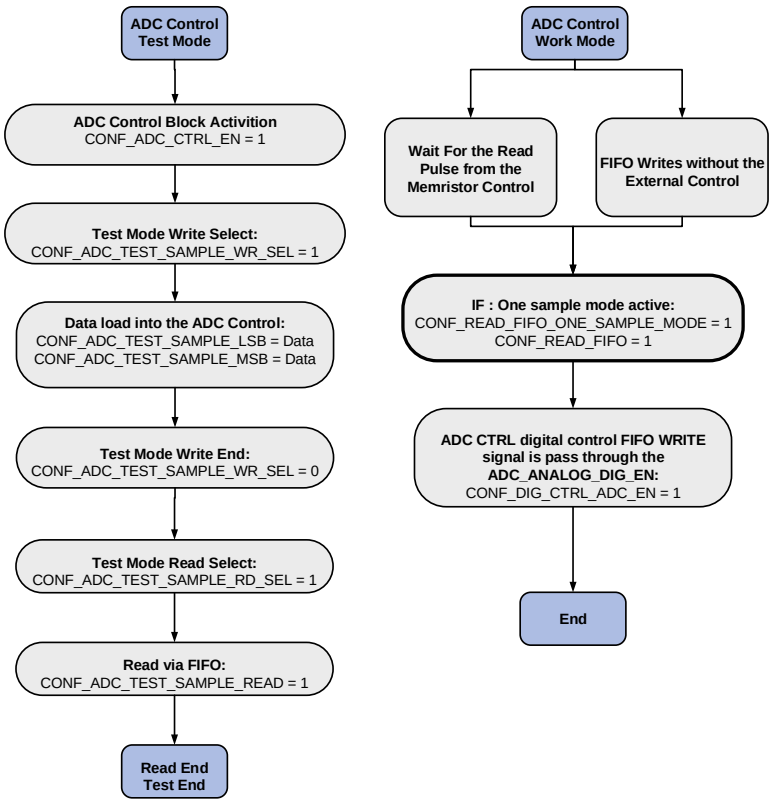


Figure 3.7: ADC Control Sequence

3.4.2 Memristor Control

The following sections explore the analog and digital control blocks, highlighting their roles in voltage and current regulation, register configuration, and PWM signal generation for memristor control.

3.4.2.1 Analog Control Block

For read operations, the amplitude of the read signals is delivered to the opamp input via the DAC, while the supply voltage is regulated by the opamp. The inclusion of the opamp is essential because the DAC alone lacks sufficient driving capability to maintain a stable output signal when current is drawn. During reset and set operations, the opamp functions in voltage follower mode to provide regulation, sourcing the current driving the memristor. Alternatively, the current can be sourced directly from the supply and regulated by the opamp. A third option involves driving the memristor directly from the supply without opamp regulation.

An overview of the analog circuit is provided in Figure 3.8. Two DACs are dedicated to controlling the two rows of memristors, while two ADCs are assigned to the two columns. Channel 1 (CH1) of the PWM control generates PWM pulses for Row 1, and Channel 2 (CH2) handles the PWM pulses for Row 2.

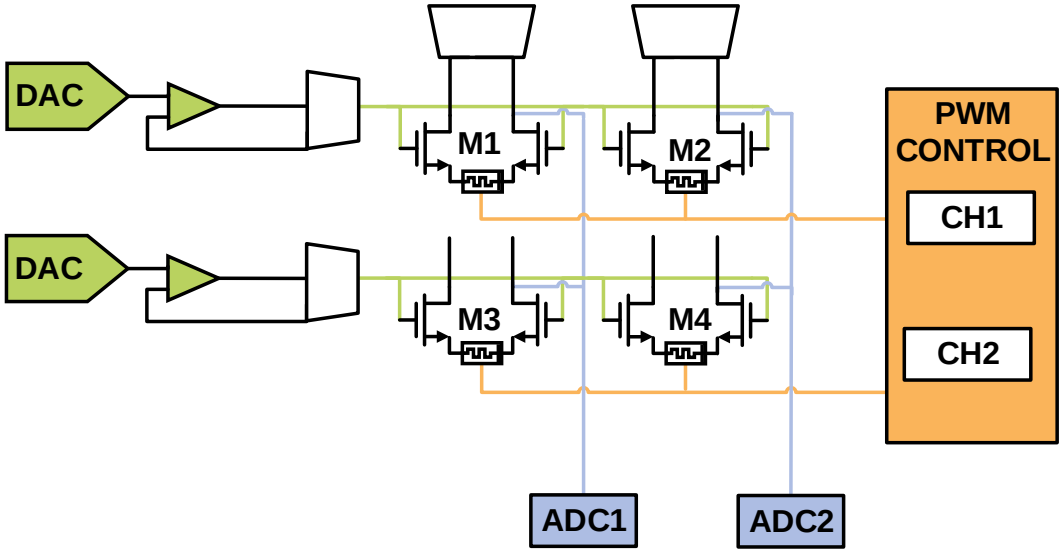


Figure 3.8: The layout of the analog circuit of 2×2 memristor array

The analog registers in the MEMRISTORCTRL configuration manage the operational modes and settings for the opamps, memory states, and current compliance transistors. These registers define whether the opamps are in the high-impedance state, differential mode, or other configurations for both voltage and current modes. They also control the width of current compliance transistors, select memory states (like READ, HRS, and LRS), and choose between internal or external DACs for voltage and current references. An overview is shown in Table 8.1. The number in the Config Register column refers to the index of different analog registers. An overview is shown in Appendix, where the number in the Config Register column refers to the index of different analog registers.

Based on these 15 registers, the memristor array's write and read operation could be achieved by assigning different values into different registers. An overview of the analog memristor control algorithm is shown in Figure 3.9.

3.4.2.2 Digital Control Block

The Figure 3.10 illustrates a digital schematic of the memristor control system, which includes an ADC block. The top left block is the top-level block, responsi-

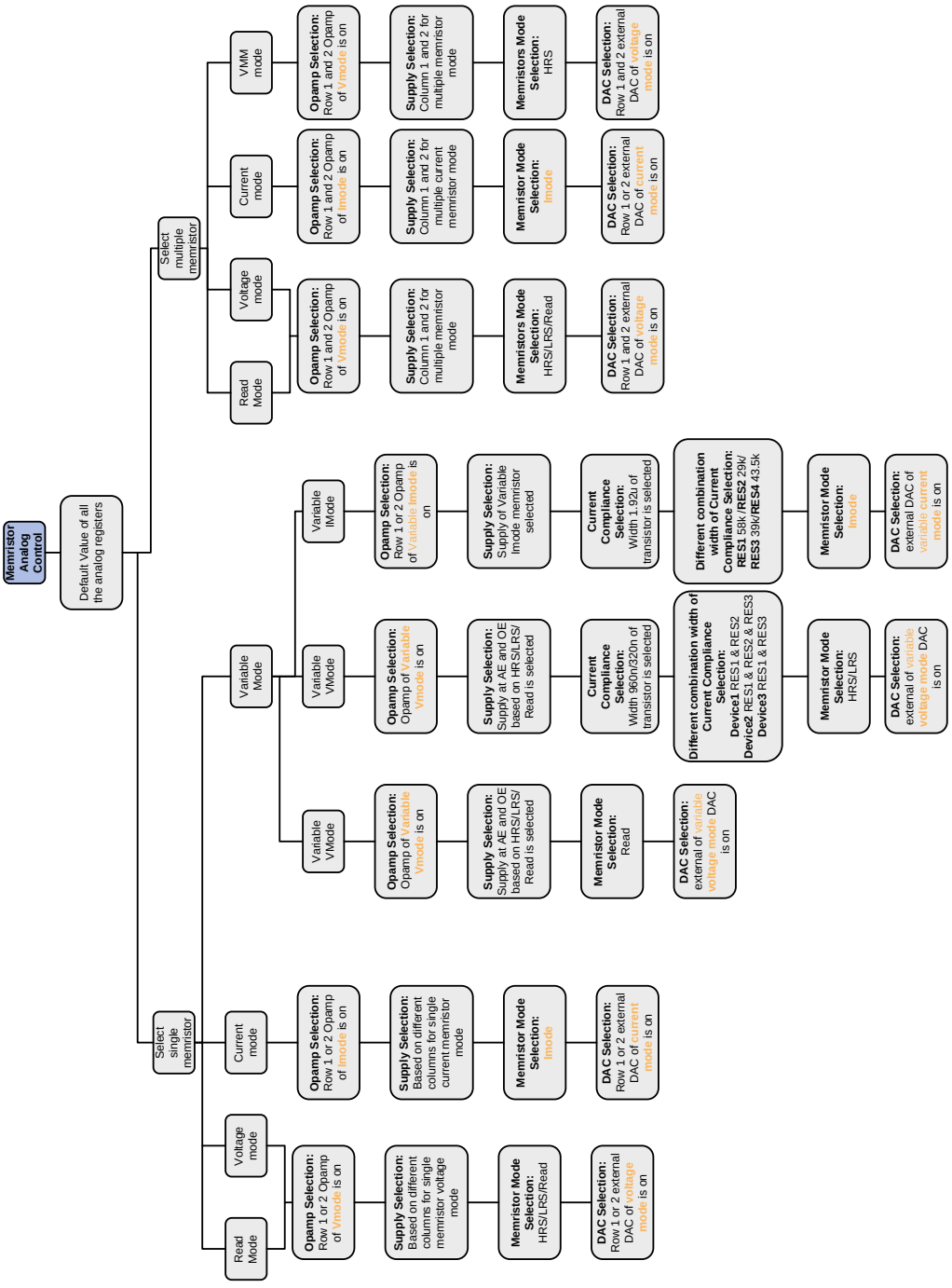


Figure 3.9: The algorithm of analog memristor control

ble for managing data output and status updates within the MEMRISTORCTRL system. This block ensures that processed data is correctly routed and that the status of operations is communicated to other system components. The middle splitter block represents an interface control block, handling the connections and communication interfaces. The top right block is the memristor control block, which manages PWM signals for pulse generation to set and reset different memristors across various operation modes. This memristor control block interfaces with the ADC control block to control the sequence of receiving analog signals, converting them into digital data and providing feedback to the memristor control block.

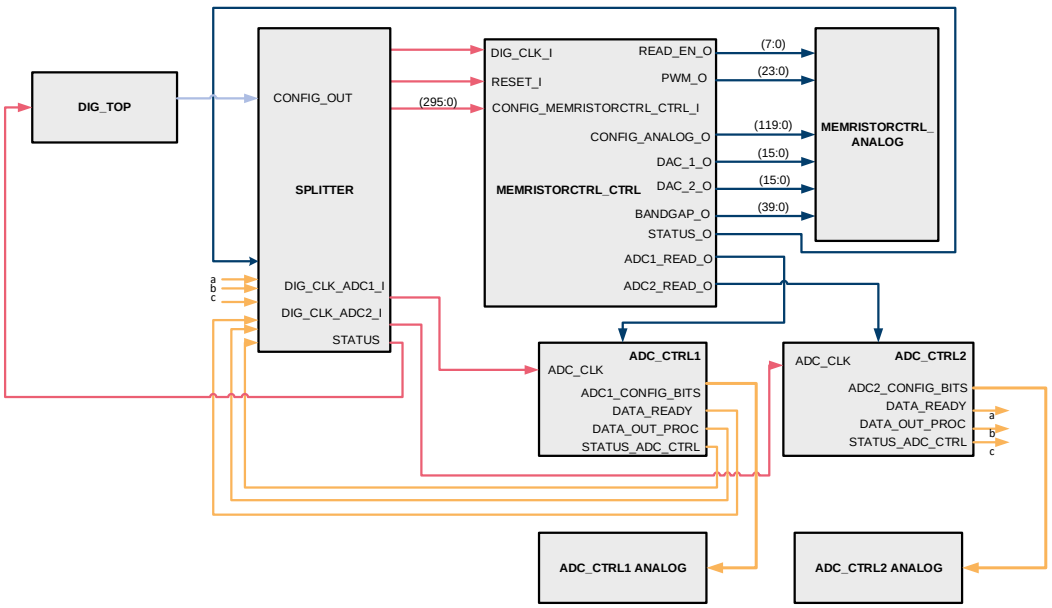


Figure 3.10: The architecture of the digital topcircuit

The primary focus of this work is on the memristor control block. The memristor control digital block receives a digital clock, a reset signal, and 296 configuration bits, which include both analog and digital configuration settings which are then sent to the analog control block. The analog control block subsequently applies various pulses to the selected memristor based on these control signals.

3.4.3 PWM Configuration of Memristors

This subsection delves into the characteristics and working algorithm of the Pulse Width Modulation (PWM) generator used in the memristor control system, as well as the specific behavior of PWM signals in different operational modes. It begins

by discussing the design, configuration, and operational algorithm of the PWM generator, including how it produces and controls pulse signals for the memristor array. Following this, the discussion shifts to the application of PWM signals across various modes—voltage, current, and variable—highlighting how these signals are utilized during different phases of memristor operation, such as reset, set, and read processes.

3.4.3.1 Characteristics of PWM Generator and its Working Algorithm

The 2×2 memristor array of memristor M1, M2, M3, and M4 are arranged as illustrated in Figure 3.11. The PWM and complementary PWMB pulses are produced by the PWM generator featuring 16-bit registers, operating at a frequency of 10 MHz. The clock divider is configurable with values that are powers of 2, up to a factor of 32, to adapt to different ranges of pulse parameters such as pulse widths.

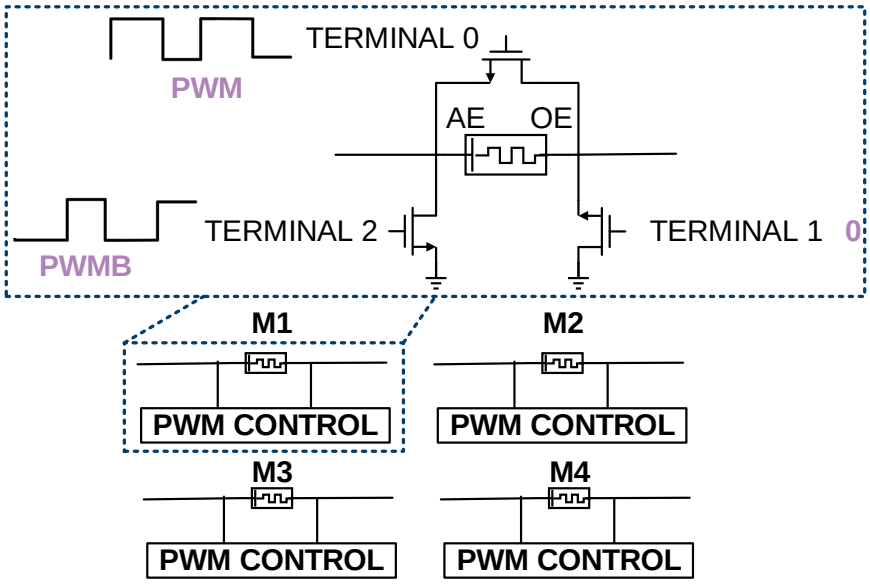


Figure 3.11: 2×2 memristor array layout

Figure 3.12 illustrates the configuration of two distinct PWM generators connected to four memristors. The first PWM generator (ROW1 PWM GENERATOR) controls the output pulse width of memristor M1 and memristor M2, while the second generator (ROW2 PWM GENERATOR) controls memristor M3 and memristor M4. In the variable mode of M5, either ROW1 PWM GENERATOR or ROW1 PWM GENERATOR could be selected.

Each PWM generator is designed to create two distinct pulse signals, PULSE1

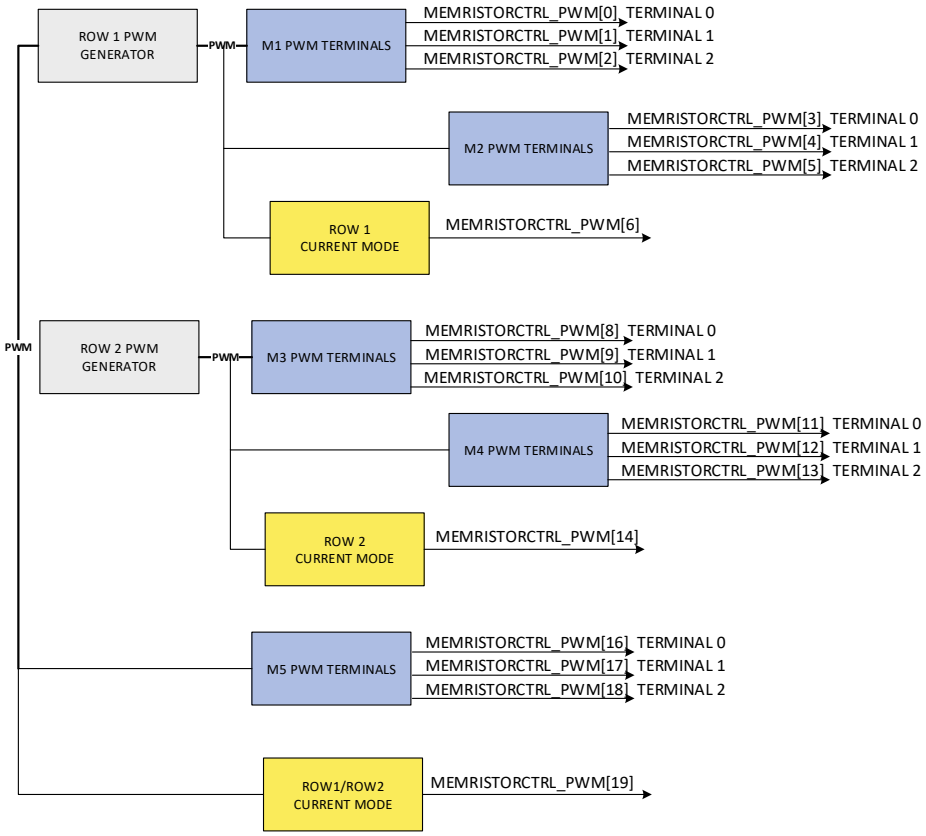


Figure 3.12: PWM Terminal Configuration, MEMRISTORCTRL_PWM ports will be used for simulation purpose

and PULSE2, based on various control inputs and timing parameters as shown in Figure 3.13.

It operates with a reset and is driven by a clock signal (CLK). The module starts generating pulses when enabled (PULSE_GEN_EN) and can be controlled to start and stop based on the PULSE_START_EN signal. The width and period of PULSE1 are determined by the parameters PULSE_1_WIDTH_CNT_VAL and PULSE_1_PERIOD_CNT_VAL, respectively.

The width and delay of PULSE2 in the PWM generator are determined by the parameters PULSE_2_WIDTH_CNT and PULSE_2_DELAY_CNT. PULSE2 begins after the delay period, which means it will start once the main counter PUL_T_CNTR reaches the delay count. The width of PULSE2 defines how long PULSE2 remains high after the delay period. Specifically, PULSE2 remains high as long as the difference between the current count and the delay count is less than the width of PULSE2, ensuring that PULSE2 is generated after a specific delay

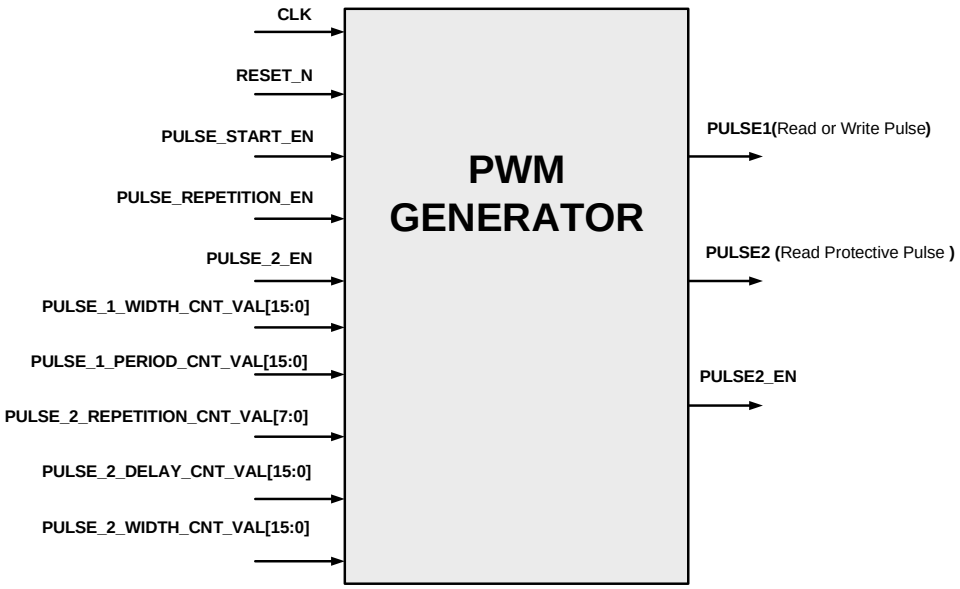


Figure 3.13: The interface of the PWM generator

from the start of PULSE1 and lasts for the defined width period.

The flowchart in Figure 3.14 describes the sequential process of generating two pulse signals, PULSE1 and PULSE2, based on various inputs and timing parameters. Initially, the module resets and initializes internal flags and counters upon a low reset signal. When the pulse generation is enabled, the module checks for a rising edge on the start enable signal. If detected, the pulse enable flag is set, starting the main pulse generation process. The counter increments with each clock cycle and PULSE1 is activated for a duration specified by PULSE_1_WIDTH_CNT_VAL. If PULSE2 is enabled, it activates after a delay of PULSE_2_DELAY_CNT and remains active for a duration of PULSE_2_WIDTH_CNT. The process includes handling pulse repetition if enabled (PUL_REP_EN), where the repetition counter (PUL_REP_CNTR) keeps track of the number of pulses generated.

3.4.3.2 The PWM Pulses in Different Operation Modes

In voltage mode, during the reset operation of a memristor, the PWM signal and its complementary PWMB signal are applied to Terminal 0 and Terminal 2 of the active memristor, respectively, as illustrated in Figure 3.11. In this configuration, the terminals of inactive memristors are set to low, high, and high levels for Terminal 0, Terminal 1, and Terminal 2, respectively, as shown in Figure 3.15a, where

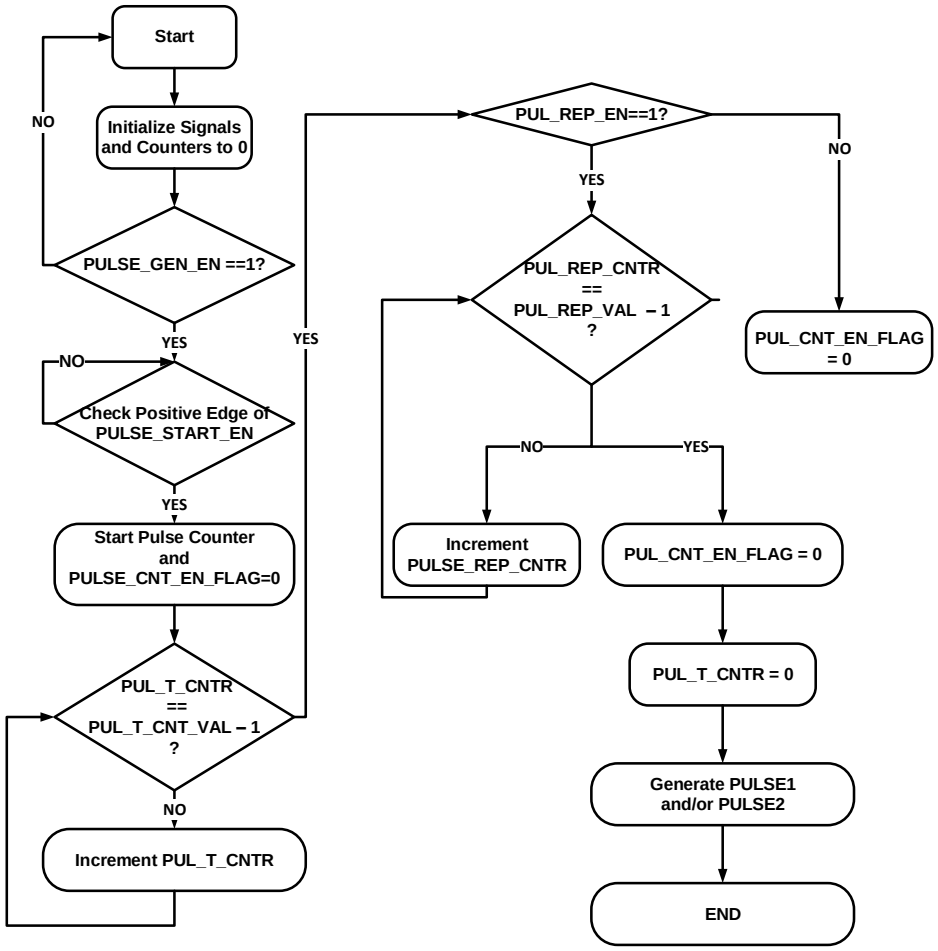


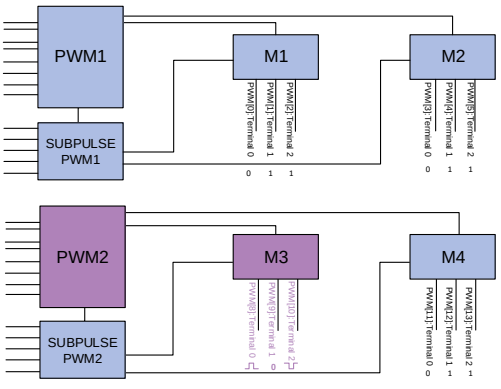
Figure 3.14: The PWM generator algorithm

M3 is undergoing the reset operation. During the set operation, such as for M1, as depicted in Figure 3.15b, the PWMB signal is instead applied to Terminal 1. The PWM pulses for the read operation are identical to those used during the reset operation, as shown in Figure 3.15c. However, to prevent short circuits, protective pulses are applied to a switch connected to the memristors, ensuring the circuit is activated at the appropriate time for safe operation.

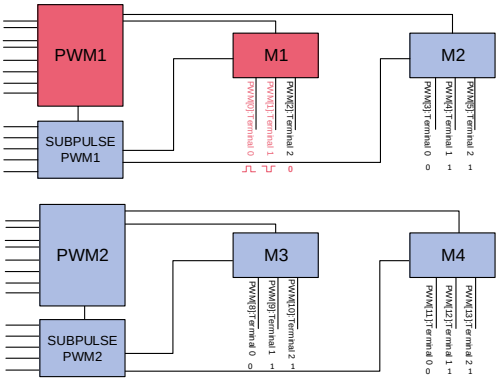
In current mode, the active memristors are set to a low level on all terminals, while the inactive memristors retain the same terminal configuration as in voltage mode (low, high, high). The PWM signals are emitted from the current mode terminals corresponding to the first and second rows. If M1 and/or M2 are activated, the PWM signal is transmitted through the first-row terminal, and similarly for the second row with M3 and/or M4. An example of programming M1 and M3 in

current mode is shown in Figure 3.15d.

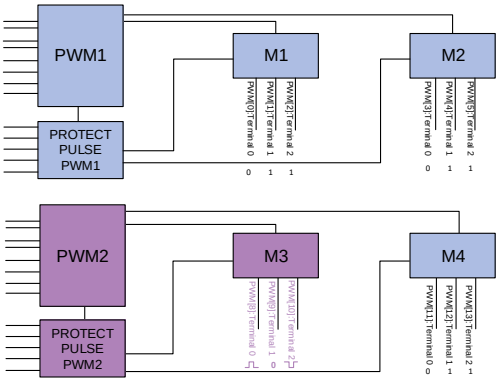
In variable mode, only the fifth memristor array is utilized. This mode serves as a test configuration to evaluate different transistor widths. The variable voltage mode operates similarly to voltage mode, while the variable current mode adheres to the same principles as current mode.



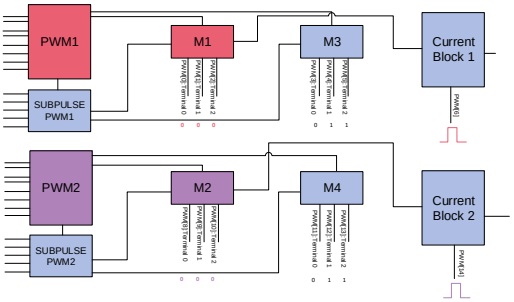
(a) PWM conditions for reset operation



(b) PWM conditions for set operation



(c) PWM conditions for read operation



(d) PWM conditions for current operation

Figure 3.15: PWM conditions for different operational modes

3.4.4 Memristor Control Digital Sequence Configuration

To effectively manage the PWM pulses for various operational modes, configuration bits from ten registers, as shown in Appendix, have been incorporated on the digital side to facilitate distinct mode operations. For each operation, the

corresponding bits must be set to a high level, while the remaining bits remain unchanged. The sequence of the write control process shown in Figure 3.16 from a digital perspective is as follows:

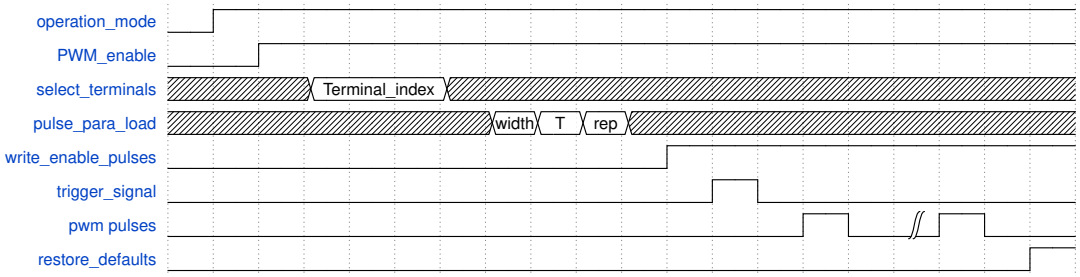


Figure 3.16: Write mode sequence.

- Activation of the operation mode (voltage or current mode).
- Activation of the PWM generator for the relevant terminals.
- Configuration of pulse width, period, and repetition values in the registers.
- Selection of terminals of the memristors for PWM pulse generation.
- Enabling repetition and writing pulses for the corresponding memristors.
- Enabling the trigger signal of the corresponding row to start the counters.
- Restore all the configuration bits to the default value when the write process is finished.

As for the read sequence in Figure 3.17, the protective pulse would be applied.

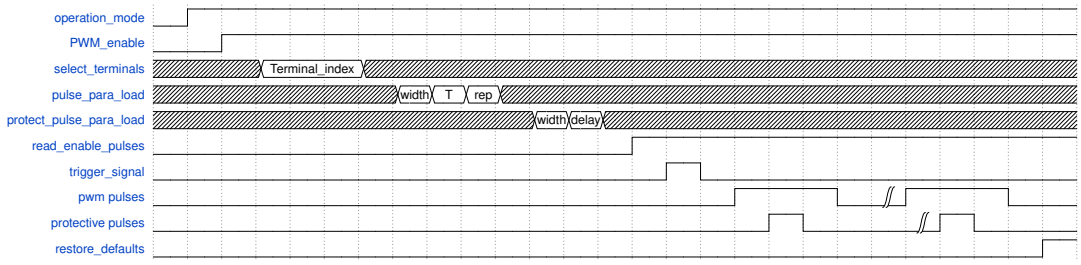


Figure 3.17: Read mode sequence

- Activation of the PWM generator for the relevant terminals.

- Enabling the protective pulse of the corresponding memristor.
- Configuration of pulse width, period, and repetition values of the read pulse as well as the width and delay of the protective pulse in the registers.
- Selection of terminals of the memristors for PWM pulse generation.
- Enabling repetition and read pulses for the corresponding memristors.
- Enabling the trigger signal of the corresponding row to start the counters.
- Restore all the configuration bits to the default value when the read is done.

3.4.5 Programming of Digital Memristor Control

The digital control system for applying the PWM pulses on the selected memristors is a critical component of this work, providing the foundational framework for managing complex operations and ensuring precise control over the hardware. Its robust design and functionality are essential for achieving the desired performance and accuracy in various applications. The digital control system presented here includes initialization, configuration, and operation of memristors, allowing for various functionalities such as PWM generation, read/write operations, and mode management.

3.4.5.1 The Sequence of Initialization

Before performing write or read operations, the pulse period range must be determined, and the corresponding clock divider will be used to reduce the clock frequency. Once the clock divider is set, the PWM block needs to be activated for the subsequent read and write operations as illustrated in Figure 3.18.

3.4.5.2 The Sequence of Write Operation

The voltage mode sequence as shown in Figure 3.19 involves the process of configuring and managing voltage pulses. Initially, the pulse parameters such as pulse period, width, and repetition are loaded into specific configuration registers. Each trigger signal is followed by a delay to ensure precise timing. The pulse repetition is enabled, and the start counter is activated to execute the pulses, which are then applied to selected terminals based on whether the pulses are to be set or reset. This mode focuses on controlling the voltage pulses for both channels (M1/M2 and M3/M4). After the pulses are executed, the system waits for the program to finish and then restores all configuration bits to their default values, ensuring the system's readiness for subsequent operations. It is important to note that

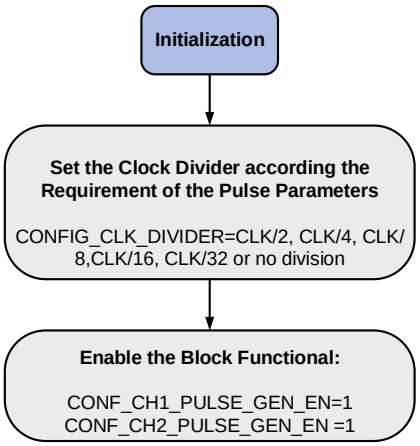


Figure 3.18: Initialization sequence of memristor control block

the start trigger signals must be enabled simultaneously to ensure synchronization when writing to two channels concurrently.

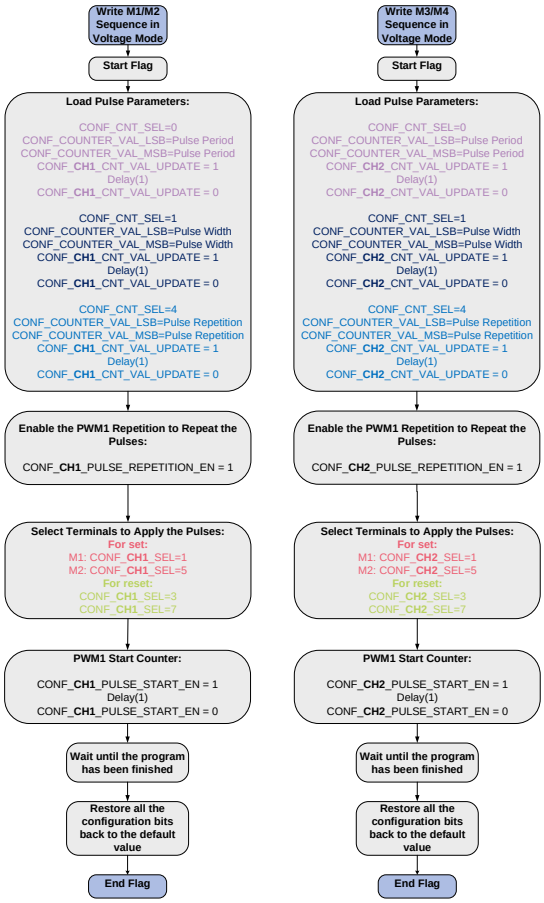


Figure 3.19: Voltage mode write sequence

The current mode sequence illustrated in Figure 3.20 is designed to manage current pulses. Like the voltage mode, it starts with loading pulse parameters and enabling pulse repetition through specific configuration settings. However, in the current mode, the pulses are applied to terminals that remain at a low level, with PWM pulses driven from the current block once the current mode is selected. Channels are shared such that M1 and M2 use one channel, while M3 and M4 use the other.

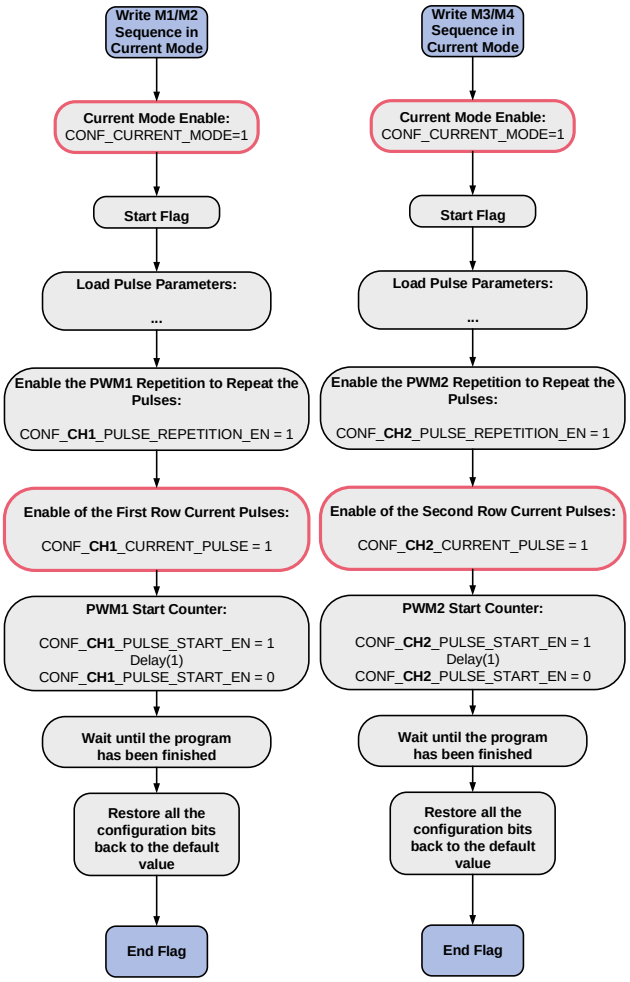


Figure 3.20: Current mode Write sequence

Figure 3.21 is in variable mode, the fifth memristor is used to test different transistor widths. It can utilize either the PWM1 generator or the PWM2 generator. The voltage mode and current mode for the fifth memristor follow the same pattern as the previous sequences.

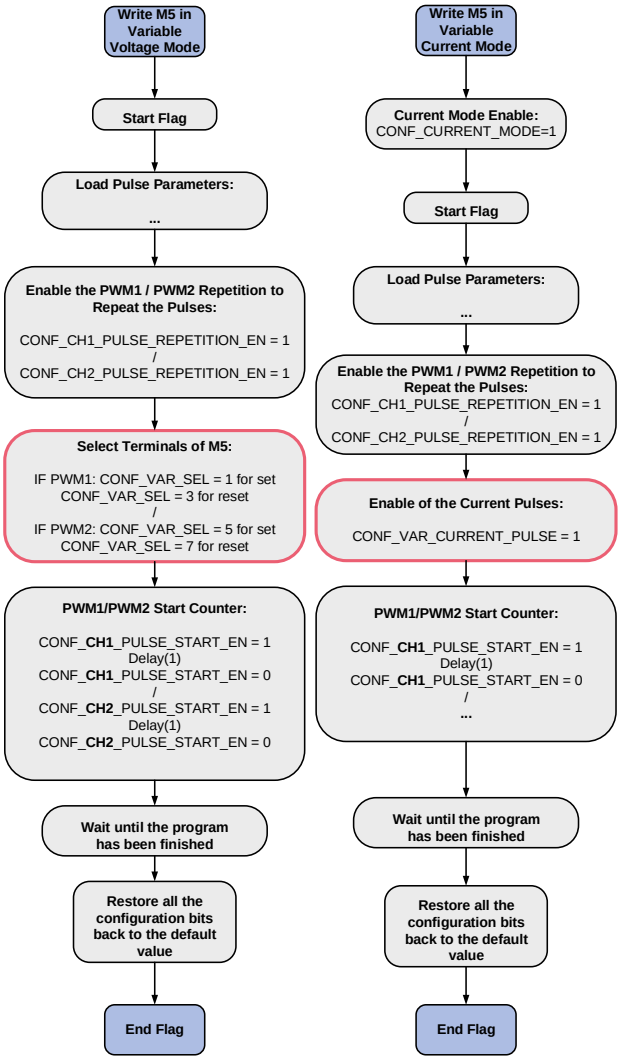


Figure 3.21: Variable mode write sequence

3.4.5.3 The Sequence of Read Operation

The read sequence in Figure 3.22 is similar to the reset sequence in voltage mode. In addition to loading the pulse parameters, the parameters for protective pulses must also be considered. The protective pulse should include a delay relative to the read pulse, resulting in a smaller width than the read pulses. Once the read pulses are completed, the memristor control block sends a read enable signal to the ADC control block.

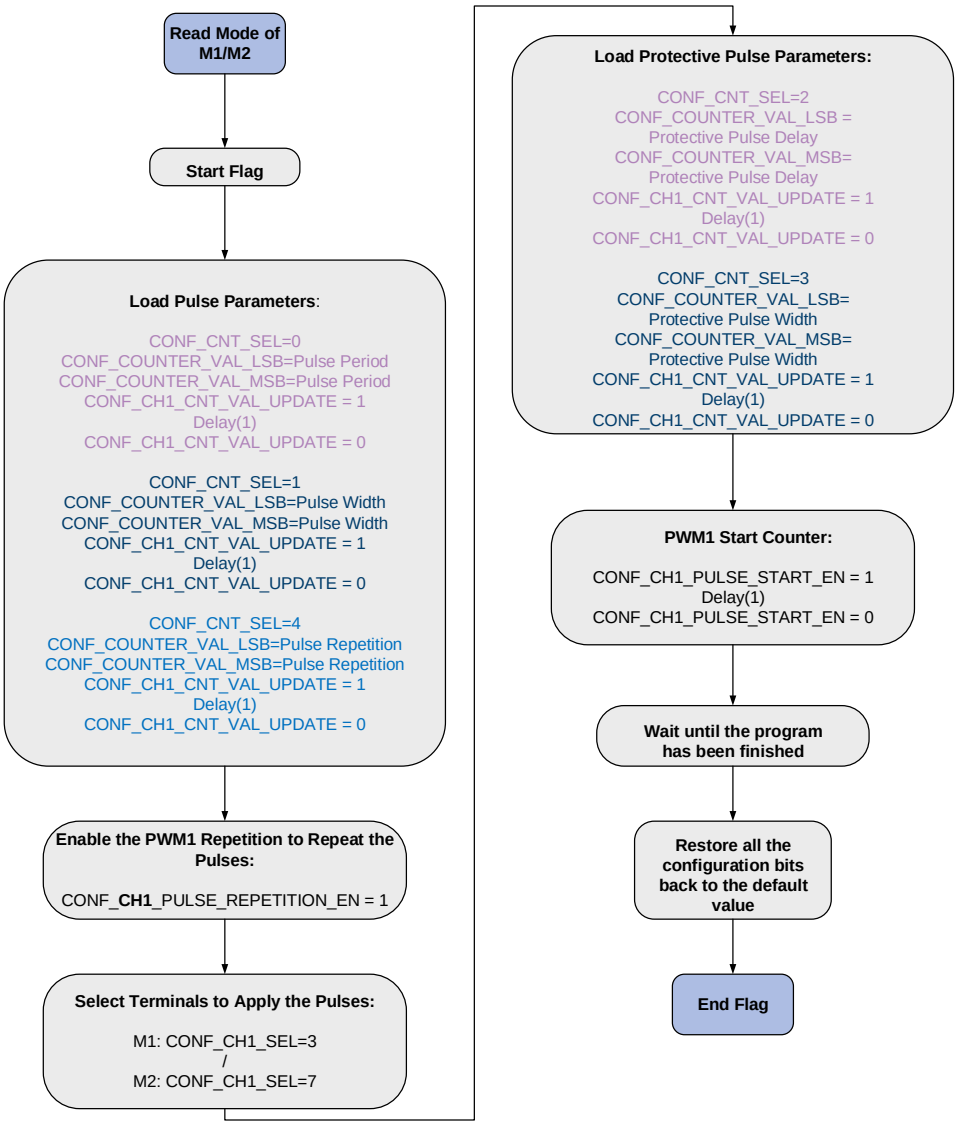


Figure 3.22: Read mode sequence

3.4.5.4 Program Overview

The control system is written in C and leverages low-level memory operations to manipulate the state and behavior of individual memristors in a grid. The system includes:

- Macros and Definitions: These provide easy access to various configuration and control registers. To facilitate memory-mapped register access and bit

manipulation, the system uses macros. These definitions allow for concise and readable code, simplifying hardware interaction.

```
#define DUT_OFFSET 0x0002000 *4
#define CONFIG_MEMRISTORCTRL_DIG_1 16 *4
#define CONFIG_MEMRISTORCTRL_DIG_2 17 *4
#define CONFIG_MEMRISTORCTRL_DIG_3 18 *4
#define CONFIG_MEMRISTORCTRL_DIG_4 19 *4
...
//CONFIG_MEMRISTORCTRL_DIG_1
#define CLK_DIVIDER_MASK                0x07
//#define CLK_DIVIDER_OFFSET              0

//CONFIG_MEMRISTORCTRL_DIG_2
#define CH1_PULSE_GEN_EN_MASK           0x01
#define CH1_PULSE_START_EN_MASK         0x02
```

Listing 3.1: Macros and definitions

- Utility function: The delay function is crucial for ensuring that trigger signals, such as the initiation of counter signals and update signals, remain in an active high state for at least one clock cycle. This function simulates the behavior of a counter to achieve the desired wait time, measured in clock cycles. The ratio between actual clock cycles and cycle counts in the processor model is approximately 7:1, accounting for the delay.

  For example, when an argument of 200 is passed to the delay function, the delay between consecutive operations will be expected to be 200 clock cycles. To achieve this, the processor performs the loop operation 28 times. This built-in calculation ensures precise timing control, which is essential for the accurate synchronization of various operations in digital systems.

```
void delay(int clock_cycle)
{for (volatile int i = 0 ; i < clock_cycle/7-1; i++)
   ↪ {}}
```

Listing 3.2: Utility function for delay

- Struct Definition: A data structure to encapsulate the pulse and protective pulse parameters and states of a memristor.

```
typedef struct {
int mem_pos; // memristor position
```

```
3     int pulse_w; // the pulse width
4     int pulse_T; // the pulse period
5     int pulse_rep; //the pulse repetition
6     int subpulse_w; // the protective pulse width
7     int subpulse_delay; // the protective pulse delay
8     int ext_DAC; //selection of external DAC
9     int int_DAC; //selection of internal DAC
10    bool on; // selection of ON/OFF state of memristor
11    } memristor;
```

Listing 3.3: Memristor struct definition

- Initialization and Configuration Functions: Functions are provided to initialize and configure individual memristor pulse parameters and their operational settings. The compiler will assign an address within the range of $0x2000 \times 4$ to ensure all variables are stored within the DUT, allowing for subsequent modifications in the RISC-V processor model within ASIP Designer. Upon completion of the digital control testing, the initialization and configuration functions will be transitioned to the high level mixed signal management place.

```
1     volatile memristor *m1;
2     volatile memristor m1_struct;
3     m1 = &m1_struct;
4     memristor_init(m1, 11, true, false, false);
5     // 11 refers to  (1,1) of the memristor array
6     ...
7     memristor_init(m5, 31, true, false, false);
8     // the fifth memristor
```

Listing 3.4: Memristor array declaration

```
1     void memristor_init(volatile memristor *m, int
         ↪ mem_pos, bool ext_DAC, bool int_DAC, volatile
         ↪ bool on) {
2          m->mem_pos = mem_pos;
3          m->pulse_w = 313;
4          m->pulse_T = 625;
5          m->pulse_rep = 3;
6          m->subpulse_w = 200;
7          m->subpulse_delay = 30;
8          m->ext_DAC = ext_DAC;
9          m->int_DAC = int_DAC;
```

```
10          m->on = on;}
```

Listing 3.5: Memristor array initiation

- Declaration of the Flag and Synchronization Point: A flag named `p_proc_status` is defined to act as a notification signal, indicating the start and end of each memristor programming operation during the simulation stage. The flag and synchronization point are currently written from the UVMF test bench for debugging purposes. In the real implementation, these will be replaced by a delay function to wait for the generation of PWM pulses for each memristor.

  The synchronization point ensures that the processor and sequence are synchronized, guaranteeing the completeness of each programming process and facilitating the debugging process. The `volatile` type is used to inform the compiler not to optimize these bits. The address for storing the flag will be allocated by the ASIP Designer's compiler within the DUT's address space, while the synchronization point will be allocated in a designated memory area dedicated to processor testing and synchronization purposes.

```
1     volatile char * p_proc_status;
2     volatile int * p_synchro;
3     int p_synchro_addr;// Synchronization Point: prevent
         ↪  PROC from running too fast
4     p_synchro = &p_synchro_addr;
5     *p_synchro = 0x0;
6     p_proc_status = (char *) (0x208a *4); // Processor
         ↪ Test
7     *p_proc_status = 0x0;
8     *(p_proc_status +4) = 0x0;
```

Listing 3.6: Flag and synchronization point declaration

- For each programming process, the terminals of non-active memristors should be set to (low, high, high). Therefore, the functions `void DUT_resetChDefault1(int mem_pos)` and `void DUT_setChDefault2(int mem_pos)` are introduced to reset Terminal 1 of the unselected memristors to a low level and Terminals 2 and 3 of the unselected memristors to a high level. Following this, `config_init_ch` will be applied to all memristors. Although it might be simpler to write this in a loop, it is recommended to write it straightforwardly to prevent compiler optimization in ASIP Designer. The fifth memristor operates exclusively in variable mode. To avoid confusion, its functionality is managed separately from the others.

```
void config_init_ch(volatile memristor *m1, volatile
    ↪ memristor *m2, volatile memristor *m3, volatile
    ↪ memristor *m4, volatile memristor *m5)
{
 if (m1->on == false){
    DUT_resetChDefault1(11);
    DUT_setChDefault2(11);}
  else{
    DUT_resetChDefault1(11);
    DUT_resetChDefault2(11);}
    ...
if (m5->on == false){
    DUT_resetChDefault1_var_mode();
    DUT_setChDefault2_var_mode();}
  else{
    DUT_resetChDefault1_var_mode();
    DUT_resetChDefault2_var_mode();}
}
```

Listing 3.7: Initialization of PWM terminals

- Write Functions: The write functions are responsible for loading pulse parameters into the PWM register, enabling the repetition bit for the designated pulse, and selecting the corresponding memristor terminals to apply the pulse. The only difference between the set and reset functions is the selection of terminals.

```
        void write_digital_set(volatile memristor *m) {
            pwm_generator(m);
            DUT_setChPulseRepetitionEn(m->mem_pos);
            DUT_setCh_Pulse(m->mem_pos);
            return;
        }
```

Listing 3.8: The set function overview

The counter will begin once the above configuration bits have been loaded or enabled. Below is an example of the setting operation for memristor M1 in voltage mode.

```
      *p_proc_status = 0x11; // start flag, use 16 LSB
      *(p_proc_status +4) = 0x11;
      m1-> on = true; // m1 is active for programming
```

```
4        config_init_ch(m1, m2, m3, m4, m5);
5        write_digital_set(m1);
6        DUT_setChPulseStartEn(11); //start trigger signal
7        delay(1); //make sure one clock cycle duration of
            ↪ the trigger signal
8        DUT_resetChPulseStartEn(11);
9        while (*p_synchro == 0x00);
10       *p_synchro = 0x00;
11       write_restore_default(m1, m2, m3, m4);
12       *p_proc_status = 0x22; // end flag, use 16 LSB
13       *(p_proc_status +4) = 0x22;
14       while (*p_synchro == 0x00); // synchronization of
            ↪ the current programming process and
            ↪ differentiation from the subsequent
            ↪ programming process
15       *p_synchro = 0x00;
```

Listing 3.9: Reset of M1 in Voltage Mode

In current mode, the configuration bit for current mode needs to be turned on at the beginning of the programming sequence, while other steps follow a similar strategy.

```
1    void DUT_setCurrentMode(){
2        int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_3);
3        *p_control |= CURRENT_MODE_MASK;
4        return;
5    }
```

Listing 3.10: Bit configuration of the current mode

The variable mode is divided into variable voltage mode and variable current mode. The distinction lies in using different registers to select the terminals of the fifth memristor to drive the pulse.

- Read operation: the read operation involving setting up the parameters for the protective pulses as well as enable its corresponding PWM generator. Here is an example of M1 read.

```
1        *p_proc_status = 0x11; // start flag, use 16 LSB
2        *(p_proc_status +4) = 0x11;
3        m1-> on = true;
4        config_init_ch(m1, m2, m3, m4, m5);
```

```
5         read_digital(m1); //read pulses
6         DUT_setChPulse2En(11);
7         sub_pulse(m1); //loading pulse parameters of the
              ↪ protective pulses
8         DUT_setChPulseStartEn(11);
9         delay(100);
10        DUT_resetChPulseStartEn(11);
11        while (*p_synchro == 0x00);
12        *p_synchro = 0x00;
13        write_restore_default(m1, m2, m3, m4);
14        DUT_resetChPulse2En(11);
15        *p_proc_status = 0x22; // end flag, use 16 LSB
16        *(p_proc_status +4) = 0x22;
17        while (*p_synchro == 0x00);
18        *p_synchro = 0x00;
```

Listing 3.11: Read peration of M1

3.5 Top-Level Code Description

The top-level code provided is a C/C++ program for controlling and testing a grid of memristors. The program begins by setting the clock divider using the `DUT_setClkDivider(CLK_DIV)` function.

Next, the program declares and initializes five memristor devices (`M1` to `M5`). Each memristor is represented by a volatile pointer and a corresponding structure. This setup allows the program to directly manipulate the memory locations associated with each memristor, ensuring precise control over their states.

Following the declarations, each memristor is initialized with specific parameters using the `memristor_init` function. The parameters include the position of the memristor and various boolean flags that define its initial state. This initialization is necessary to prepare the memristors for the upcoming read and write operations.

The Pulse Width Modulation (PWM) block is then enabled and the repetition of PWM pulses is set for all channels using the `DUT_set_allChPulseGenEn()` and `DUT_set_allChPulseRepetitionEn()` functions. Enabling the PWM block is essential for generating the pulses required for memristor operations.

The program proceeds to perform a series of read and write operations on the memristors. First, analog and digital read operations are executed on individual memristors (`M1` to `M4`). These operations are followed by analog and digital read operations on pairs of memristors. These multiple read operations are necessary to test the interaction between different memristors.

Similarly, the program performs analog and digital voltage write set and reset operations on individual memristors (`M1` to `M4`), followed by the same operations on pairs of memristors. These write operations test the ability of the system to accurately set and reset the voltage states of the memristors.
In addition to voltage write operations, the program also performs analog and digital current write operations on both individual and multiple memristors. These operations ensure that the system can handle current-based control of the memristors, which is crucial for certain types of memristor applications.
The program also includes specialized handling for variable mode operations. It performs analog and digital read operations in variable mode, as well as voltage and current write set and reset operations. The variable mode operations focus on memristor `M5`, which is not included in the standard array for test purposes but follows the same strategy as the voltage and current modes.
Finally, the program executes VMM operations in both analog and digital modes. These operations are essential for testing the computational capabilities of the memristor array.
It is important to note that no ADC is involved in the C program for now due to time constraints. The ADC will be tested separately in Cadence.
Overall, this main function sets up and tests the memristor grid using various read and write operations in both voltage and current modes and includes specialized handling for variable mode and VMM operations. The structured approach will ensure comprehensive testing of the memristor array's functionality.

```
1  int main() {
2      DUT_setClkDivider(CLK_DIV);
3
4      // 2*2 memristor and m5 for variable mode grid
           ↪ declaration and initialization
5      volatile memristor m1_struct, m2_struct, m3_struct,
           ↪ m4_struct, m5_struct;
6      volatile memristor *m1 = &m1_struct, *m2 = &m2_struct,
           ↪  *m3 = &m3_struct, *m4 = &m4_struct, *m5 = &
           ↪ m5_struct;
7
8      // memristor position initialization
9      memristor_init(m1, 11, true, false, false);
10     memristor_init(m2, 12, true, false, false);
11     memristor_init(m3, 21, true, false, false);
12     memristor_init(m4, 22, true, false, false);
13     memristor_init(m5, 31, true, false, false);
14
15     // Enable PWM block
```

```
16      DUT_set_allChPulseGenEn();
17      DUT_set_allChPulseRepetitionEn();
18
19      ////Programming of memristor array start here////
20
21      volatile memristor* memristors[] = {m1, m2, m3, m4};
22      for (int i = 0; i < 4; ++i) {
23          analog_read_single_memristor(memristors[i]);
24          digital_read_single_memristor(memristors[i]);
25      }
26
27      volatile memristor* mem_pairs[][2] = {{m1, m4}, {m1,
            ↪ m3}, {m2, m4}, {m2, m3}};
28      for (int i = 0; i < 4; ++i) {
29          analog_read_multiple_memristor(mem_pairs[i][0],
                ↪ mem_pairs[i][1]);
30          digital_read_multiple_memristor(mem_pairs[i][0],
                ↪ mem_pairs[i][1]);
31      }
32
33      for (int i = 0; i < 4; ++i) {
34          analog_voltage_write_set_single_memristor(
                ↪ memristors[i]);
35          digital_voltage_write_set_single_memristor(
                ↪ memristors[i]);
36          analog_voltage_write_reset_single_memristor(
                ↪ memristors[i]);
37          digital_voltage_write_reset_single_memristor(
                ↪ memristors[i]);
38      }
39
40      for (int i = 0; i < 4; ++i) {
41          analog_voltage_write_set_multiple_memristor(
                ↪ mem_pairs[i][0], mem_pairs[i][1]);
42          digital_voltage_write_set_multiple_memristor(
                ↪ mem_pairs[i][0], mem_pairs[i][1]);
43          analog_voltage_write_reset_multiple_memristor(
                ↪ mem_pairs[i][0], mem_pairs[i][1]);
44          digital_voltage_write_reset_multiple_memristor(
                ↪ mem_pairs[i][0], mem_pairs[i][1]);
45      }
46
47      for (int i = 0; i < 4; ++i) {
```

```
48          analog_current_write_single_memristor(memristors[i
               ↪ ]);
49          digital_current_write_single_memristor(memristors[
               ↪ i]);
50          analog_current_write_multiple_memristor(mem_pairs[
               ↪ i][0], mem_pairs[i][1]);
51          digital_current_write_multiple_memristor(mem_pairs
               ↪ [i][0], mem_pairs[i][1]);
52      }
53
54      analog_variable_read();
55      digital_variable_read(m5);
56
57      analog_variable_V_write_reset();
58      digital_variable_V_write_reset(m5);
59
60      for (int i = 1; i <= 3; ++i) {
61          analog_variable_V_write_set(i);
62          digital_variable_V_write_set(m5);
63      }
64
65      for (int i = 1; i <= 4; ++i) {
66          analog_variable_I_write_set(i);
67          digital_variable_I_write_set(m5);
68      }
69
70      analog_VMM();
71      digital_VMM();
72  }
```

Listing 3.12: Read operation of M1

4 Simluation Setup

In this simulation, trv32p3 model, a 32-bit RISC-V microcontroller featuring a 3-stage pipeline is being utilized [24]. This model serves as a foundational benchmark for future research endeavors. Figure 4.1 illustrates the workflow for simulating and visualizing the output pulses of an ASIP design using Synopsys tools, GNU BFD (Binary File Descriptor) Library, and QuestaSim.

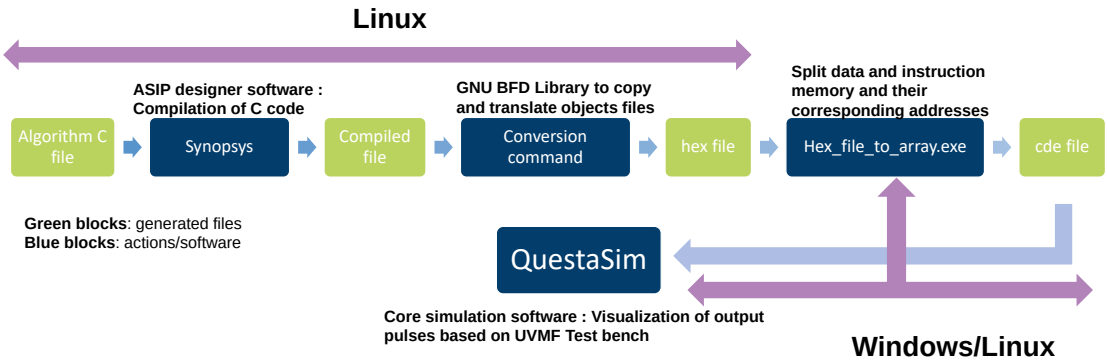


Figure 4.1: The simulation flow of PWM pulses

The process begins in a Linux environment with the programming of an algorithm in C, which is compiled into a suitable file, which is subsequently translated into object files using the GNU BFD Library. To note, ASIP Designer creates an RTL (Register Transfer Level) of a processor model described using nML language. the tool does not translate the C code into RTL. These object files are then converted into a hex file containing the binary data of the compiled algorithm. The hex file is then processed by a tool (Hex_file_to_array.exe), which splits the data and instruction memory along with their corresponding addresses. If data memory is present, each cde file is generated to represent a contiguous data set, providing

the start and end addresses specific to that set.

QuestaSim, a core simulation software on Windows, is then used to visualize the output pulses using a UVMF test bench. The green blocks in the figure represent generated files at different stages, while the blue blocks represent actions or software tools used in the workflow, spanning both Linux and Windows environments to complete the design, compilation, conversion, and simulation processes for the ASIP design.

4.1 UVMF Test Bench Overview

DUT is composed of several key components is illustrated in Figure 4.2:

- **UVMF_TEST_TOP**: Serves as the overarching test that encapsulates all other components and environments.
- **DIG_TOP_Environment**: Describes the digital top-level functionality and includes the following essential components:
  - **Universal Asynchronous Receiver-Transmitter (UART)**: Handles serial communication.
  - **JTAG Interface**: Facilitates communication with the chip.
  - **Sleep Control Module**: Synchronizes the processor and sequence.
  - **Memory Controller for Instruction**: Manages instructions associated with the control blocks.
  - **Memory Controller for Data**: Manages data associated with the control blocks.
- **Memristor Status Environment**: Dedicated to monitoring the status of the memristor control system, including:
  - **MEMRISTORCTRL_STAT_out_agent**: Monitors the PWM waveforms generated for memristor control.
  - **MEMRISTORCTRL_CONF_in_agent**: Monitors the input configuration registers related to memristor control.

  This environment focuses solely on monitoring and does not include any drivers.

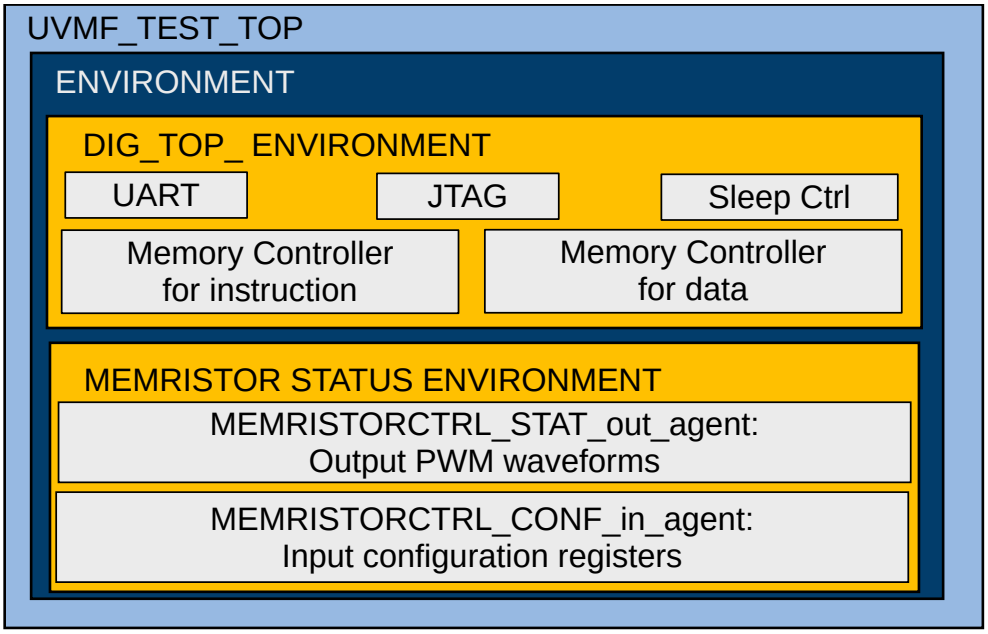


Figure 4.2: UVMF testbench blocks overview

4.1.1 The Generation of hex Code and cde Code

The memristor control is implemented in C and compiled using CHESSDE software. Upon successful compilation, an executable code is generated. It is important to note that not all instructions in the generated code directly correspond to the original C code. Some instructions are introduced by the compiler, for instance, to manage the software stack in memory. Additionally, the compiler performs various code transformations and optimizations during the compilation process.

To verify the correct generation of instructions, it is crucial to inspect the DUT memory and confirm changes in the values of the corresponding registers prior to compilation in CHESSDE.

For instance, one can validate whether the update counter trigger signal of PWM1 generator is set to 1 when new pulse parameters are loaded into the relevant storing registers. The DUT base address is *0x2000*×4. The specific register address containing the update counter trigger signal, located at the 3^{rd} bit from the LSB, is at 22×4 in decimal. Therefore, the sum of these addresses, *0x2000*×4 and 22×4, results in *0x8058*.

By utilizing the ASIP Designer Debugging interface, the defined memory block can be examined, as illustrated in the figure below. A value of 1 in the third LSB (*0x0000 0100*) corresponds to 8 in decimal, thereby confirming the correctness of the update trigger.

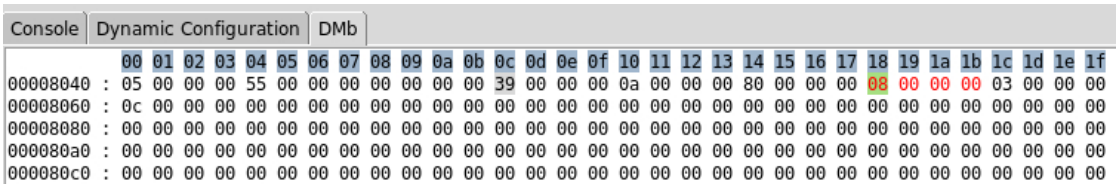


Figure 4.3: The debugging interface of ASIP designer

The code will be further translated to hex file using the command:

```
objcopy -O ihex input_file output_file.hex
```

The command converts an object file named into the Intel hex format, saving the result as "output_file.hex". The "objcopy" utility is used to copy and translate object files between different formats, with the "O ihex" option specifying the Intel HEX format, which is commonly used for programming microcontrollers.

In the Intel HEX file format, each line represents a record with specific fields. The format of a typical record is as follows: An example is shown in Figure 4.4:

```
1  :10000000372100001301C1FF33000000330000005E
2  :10001000330000003300000EF00801913000000DF
3  :100020006F0000009301B00063 8CA104 9301C00035
4  :10003000638CA102B781000093810105930250 01F6
5  :1000400003A201006 38CA200930260016394A204E6
6  :10005000136542 0023A0A100678000001365120011
7  :10006000 23A0A100678000003785000013 05C504A8
8  :10007000832105 0093E121002320350067800000E3
9  :10008000378500001305850483210500 93E101 08ED
```

Figure 4.4: An example of hex code

Take the first record line as an example: The colon character (":") indicates the beginning of the record. The next 2 hex digits ("10") represent the number of data bytes; "10" in hex means the record length is 16 bytes. Then, "0000" specifies the load address, which is *0x0000*. The following "00" indicates the record type is data. There are also record types like "01" which indicates the end of the file. The sequence "372100001301C1FF3300000033000000" consists of 16 bytes of data. The line ends with "5E", which is the checksum. It is important to note that if the addresses are not consecutive, it indicates a split between the memories. Each segment of data must be translated separately into different code files according to the instruction memory and data memory requirements.

Then the C program "Hex_file_to_array.exe" reads a hex file, processes its contents to extract data bytes, and writes the formatted data to an output file

as illustrated in Figure 4.5. It begins by counting the lines in the hex file, then reads each line to extract relevant information such as byte count, address, record type, and data bytes. The data bytes are combined into 32-bit words, ensuring any special characters are skipped during the process. The combined data is then written to the output file, with the start and end memory addresses printed to the console. This functionality is useful for preparing data memories and instruction memory initialization.

```
1   00002137
2   FFC10113
3   00000033
4   00000033
5   00000033
6   00000033
7   545010EF
```

Figure 4.5: An example of cde code

4.2 Testbench: The Memristor Control Verification Code

This section provides an exa of the components and configurations essential for setting up and executing the testbench utilized in the validation of the memristor control system. It will address the parameters required for the testbench input, the architecture of the System Verilog code employed for simulation, and the synchronization mechanisms implemented to ensure precise coordination throughout the test sequence.

4.2.1 The Input of the Testbench

When retrieving the starting and ending addresses of the instruction and data memories, along with their corresponding CDE files, these elements will be used as parameters within the testbench, as shown in Figure 4.6. Additional parameters include the memory address for the SLEEPCTRL module and the maximum number of clock cycles the testbench can execute.
In the implementation phase, one option is to pass the CDE file to MATLAB, which will run an executable file that interfaces with JTAG to initialize the in-

struction and data memories. Alternatively, the CDE file could be initialized directly without using JTAG.

```
// memristorctrl
parameter string TEST_MEMCTRL_INSTR_MEMRISTORCTRL_WITH_END_NOTIF_INIT_FILE  =
"mem_ctrl_inst.cde";
parameter int TEST_MEMCTRL_INSTR_MEMRISTORCTRL_WITH_END_NOTIF_START_ADDR    = 'h0;
parameter int TEST_MEMCTRL_INSTR_MEMRISTORCTRL_WITH_END_NOTIF_END_ADDR      = 'h7d9;

parameter string TEST_MEMCTRL_DATA_MEMRISTORCTRL_WITH_END_NOTIF_INIT_FILE   =
"mem_ctrl_data.cde";
parameter int TEST_MEMCTRL_DATA_MEMRISTORCTRL_WITH_END_NOTIF_START_ADDR     = 'h0;
parameter int TEST_MEMCTRL_DATA_MEMRISTORCTRL_WITH_END_NOTIF_END_ADDR       = 'h2;
```

Figure 4.6: The input block from the testbench

4.2.2 The Overview of the Memristor Control System Verilog Code

The System Verilog code is used here as the testbench language for validating the analog and digital design. In this thesis, the main focus of this test bench is to define and implement a test sequence for initializing and configuring a processor's SRAM without using JTAG, while including end notification and memristor control sequences as illustrated in Figure 4.7. The sequence involves configuring various control agents such as JTAG, MEMCTRL, SLEEPCTRL, and MEMRISTORCTRL, ensuring proper synchronization and setting up initial configurations. The sequence waits for specific conditions and status flags to be met, applies configurations, and updates various control registers to simulate the behavior and interactions of these components during testing. The notif_config function is repeatedly called to manage synchronization and configuration updates throughout the process, ensuring the system's correct operation and monitoring status changes.

4.2.3 The Flag and Synchronisation Point Configuration

The notif_config function as shown in Figure 4.8 in the testbench sequence is used to synchronize and coordinate various components by setting and monitoring specific notification flags. The function first uses a watchdog to set a counter, ensuring there is enough time to generate all the pulses. While the counter is counting, the data memory is set to register mode. A value of 1 is written to a specific data memory address defined in the C code to notify RISC that a synchronization point has been reached. Once the synchronization point is achieved, the data memory is switched back to APB mode. After the counter finishes counting, the watchdog is reset and RISC-V is released. It handles configuration changes, ensures proper

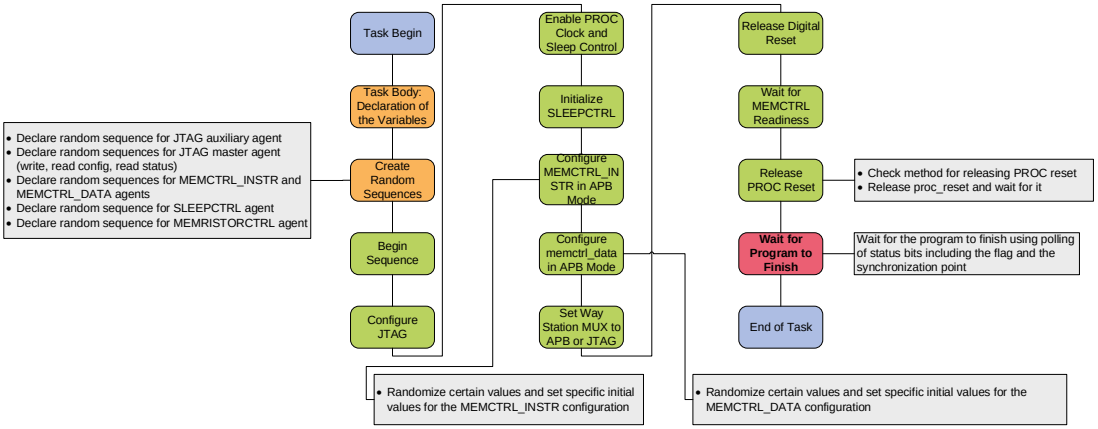


Figure 4.7: Memristor control pulse verification algorithm

start and end notifications, and maintains synchronization using sleep control and memory control settings.

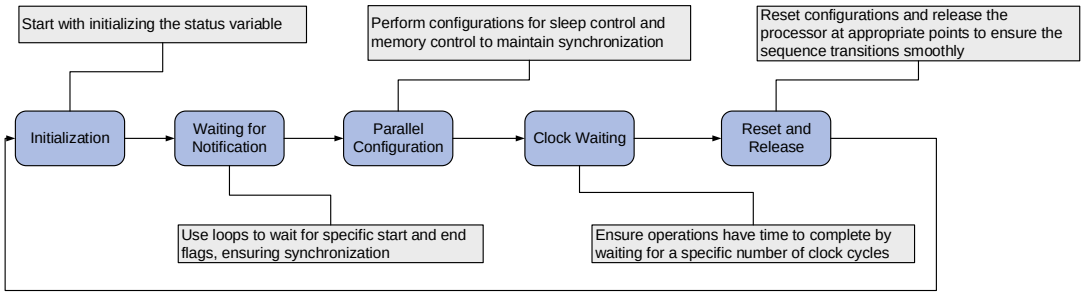


Figure 4.8: Notification algorithm overview

5 Digital Simulation Result and Discussion

The simulation is performed with Questa Sim. The first figure (Figure 5.1) illustrates the set operation in voltage mode for individual memristors. This diagram shows the sequence of setting and reading memristors M1, M2, M4, and M4. The read pulses occur on the same terminals as the reset pulses. For the reset operation in voltage mode (Figure 5.2), Terminals 1 and 2 switch to achieve inverse amplitudes at the memristor ends. The algorithm also supports simultaneous writing and reading of two channels, as shown in Figure 5.3 and Figure 5.4. In this setup, M1 and M2 belong to one channel, while M3 and M4 belong to another, allowing M1 and M3 or M1 and M4 to be written and read simultaneously, significantly reducing programming time for the memristor cells.

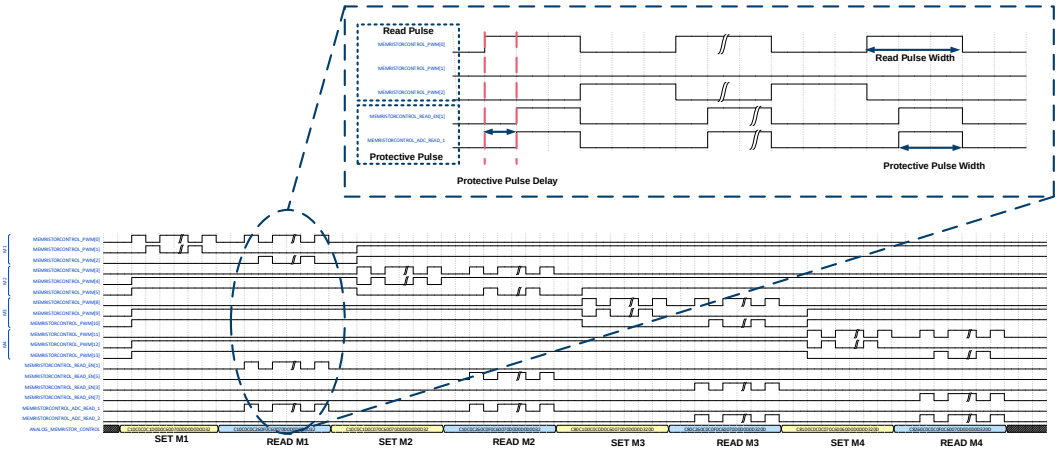


Figure 5.1: Voltage mode: PWM waveforms for the set operation and read of M1, M2, M3 and M4 in memristor array

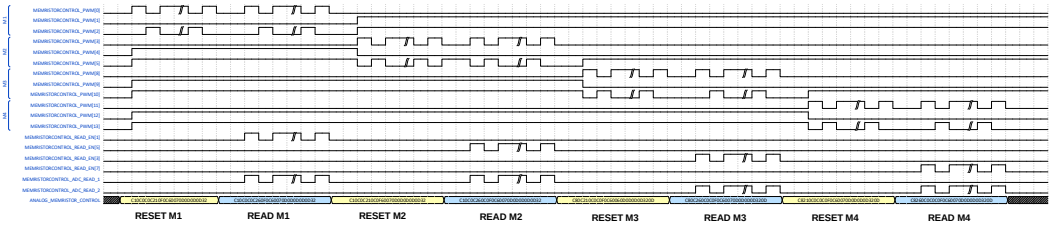


Figure 5.2: Voltage mode: PWM waveforms for the reset operation and read of M1, M2, M3, and M4 in memristor array

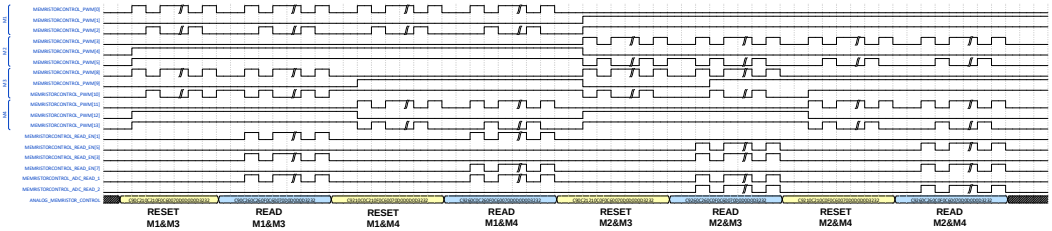


Figure 5.3: Voltage mode: PWM waveforms for the reset operation and read of M1M3, M1M4, M2M3 and M2M4 in memristor array

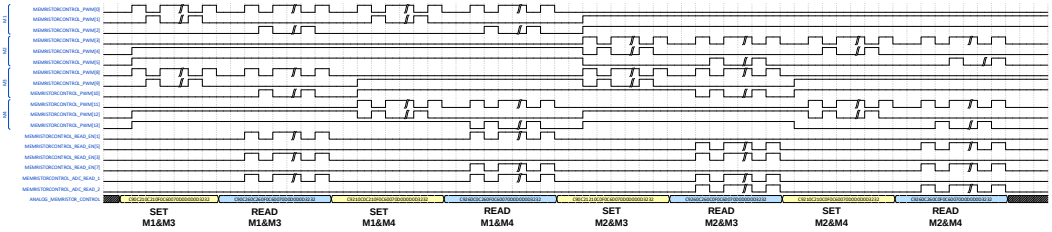


Figure 5.4: Voltage mode: PWM waveforms for the set operation and read of M1M3, M1M4, M2M3 and M2M4 in memristor array

In the current mode (Figure 5.5 and Figure 5.6), the PWM signals are applied to the current block terminals instead of directly to the memristors. In this mode, m1 and m2 share one current source, while m3 and m4 share another. Similar to voltage mode, current mode supports simultaneous reading and writing without needing to switch terminals.

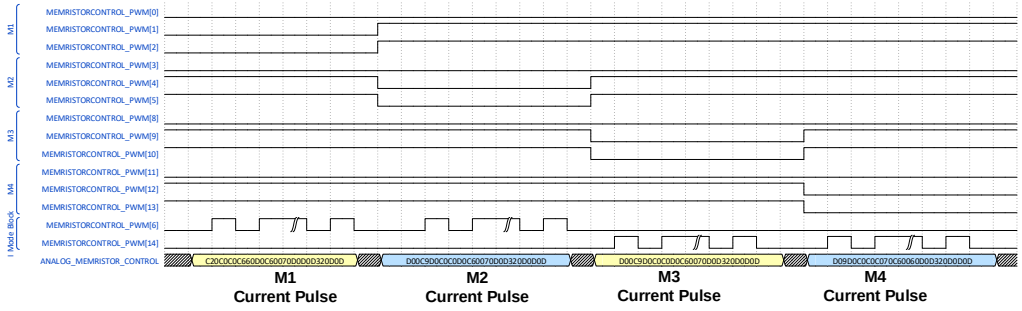


Figure 5.5: Current mode: PWM waveforms of M1, M2, M3 and M4 in memristor array

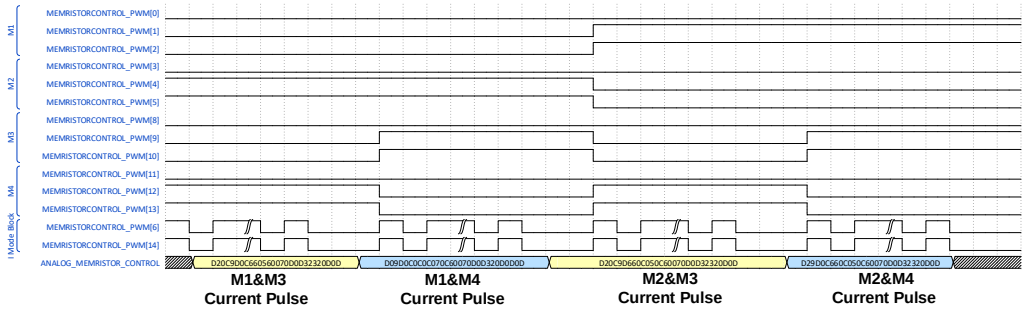


Figure 5.6: Current mode: PWM waveforms of M1M3, M1M3, M2M3 and M2M4 in memristor array

The variable mode uses only M5 (Figure 5.7), which is not included in the array for test purposes, but it follows the same strategy as the voltage mode and current mode.

Different devices are employed here to test for device-to-device variability. Device 1 has several configurations based on different resistor combinations. When configured with RES1 (58 kΩ) for LRS/IMODE, it is represented by code 54 (*0x36*). Another setup for Device 1 involves RES1 and RES2 (29 kΩ), represented by code 52 (*0x34*). When Device 1 is configured with RES1, RES2, and RES3 (39 kΩ), the code 48 (*0x30*) is used. Lastly, Device 1 can also be configured with RES1 and RES3, indicated by the code 49 (*0x31*). Device 2, when paired with RES2 for LRS/IMODE, is denoted by the code 45 (*0x2D*). This specific setup highlights the unique characteristics of Device 2 with RES2. Device 3, configured with RES3 for LRS/IMODE, is represented by the code 27 (*0x18*). This setup is tailored to demonstrate the performance of Device 3 with RES3. The value for

RES4 is 43.5 kΩ for IMODE.

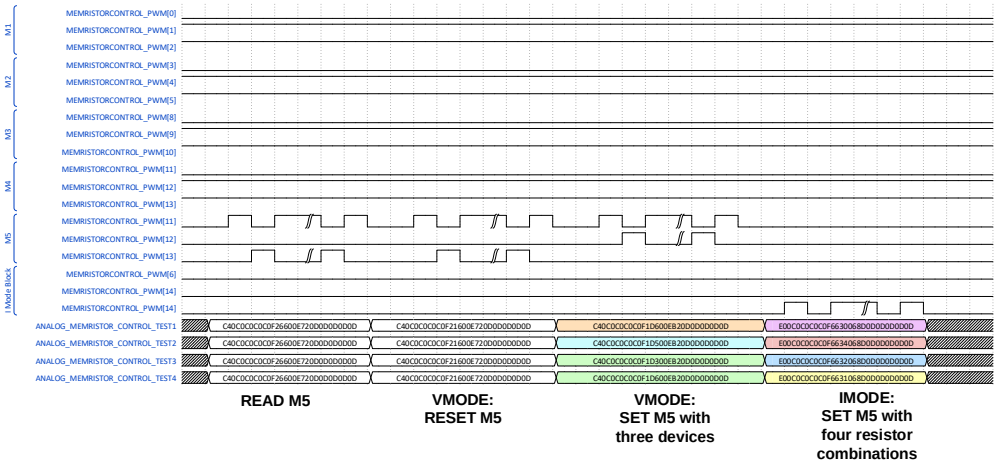


Figure 5.7: Variable mode: PWM waveforms of M5 for reset, read and set on voltage mode and the PWM pulses of current mode

The simulation results demonstrate the effectiveness and correctness of the memristor control program within the UVMF testbench. This verification confirms that the designed control signals are accurate and reliable, paving the way for further testing on a demonstrator chip. The ADC component can be directly tested on the chip due to its straightforward sequence, ensuring seamless integration and testing. The memristor control algorithm has been comprehensively validated at the simulation level, showcasing its readiness for hardware implementation. The simulation flow, which involves using Synopsys tools and Questa Sim, ensures that the digital and analog control blocks operate as intended, supporting various control modes including voltage, current, and variable modes.

6 Analog Simulation Result and Discussion

The purpose of the analog simulation is to assess whether the digital signals are accurately transmitted to the ends of the memristors. In this simulation, a specific test is performed as follows: programming cells are sequentially processed in the order of M1, M2, M3, followed by M1M3 and M2M4. For each programming cell, the sequence of operations applied to the corresponding memristor cells includes: read, reset, read, set, and read. The pulse width is configured to 1 ms, with a period of 2 ms and a repetition count of 1.
To verify the correctness of programming, a readback of the current is performed in read mode. If the read voltage of 250 mV divided by the current yields a resistance of approximately 2 kΩ, the memristor is considered to be in LRS. Conversely, a resistance of around 65 kΩ indicates that the memristor is in HRS. The coding strategy employed in the Verilog file follows the same logical sequence as outlined in the C code. This ensures consistency across both digital and analog simulations, aligning the digital signal processing with the analog behavior of the memristors.

The simulation results are illustrated in Figures 6.1 and 6.2. The PWM signals `d_PWM_GND_ROW1_o<0>,<1>,<2>` correspond to terminals 0, 1, and 2 of M1; `d_PWM_GND_ROW1_o<3>,<4>,<5>` correspond to terminals 0, 1, and 2 of M2; furthermore, `d_PWM_GND_ROW2_o<0>,<1>,<2>` correspond to terminals 0, 1, and 2 of M3; and `d_PWM_GND_ROW1_o<3>,<4>,<5>` correspond to terminals 0, 1, and 2 of M4. The differential voltages applied to the memristors M1, M2, M3, and M4 are observable at `(/AE1/OE1)`, `(/AE2/OE2)`, `(/AE3/OE3)`, and `(/AE4/OE4)` respectively.

Based on the simulation results, the appropriate read, reset, and set pulses are applied to the respective memristor ends, with the correct PWM pulses distributed across different terminals.

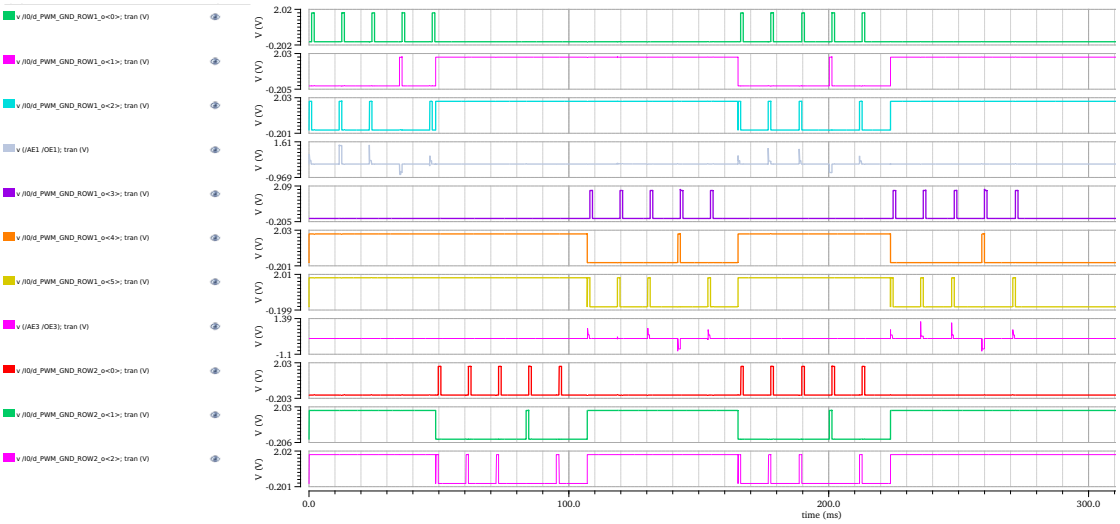


Figure 6.1: Voltage mode: PWM waveforms for the read, reset, read, set, read operation in memristor array

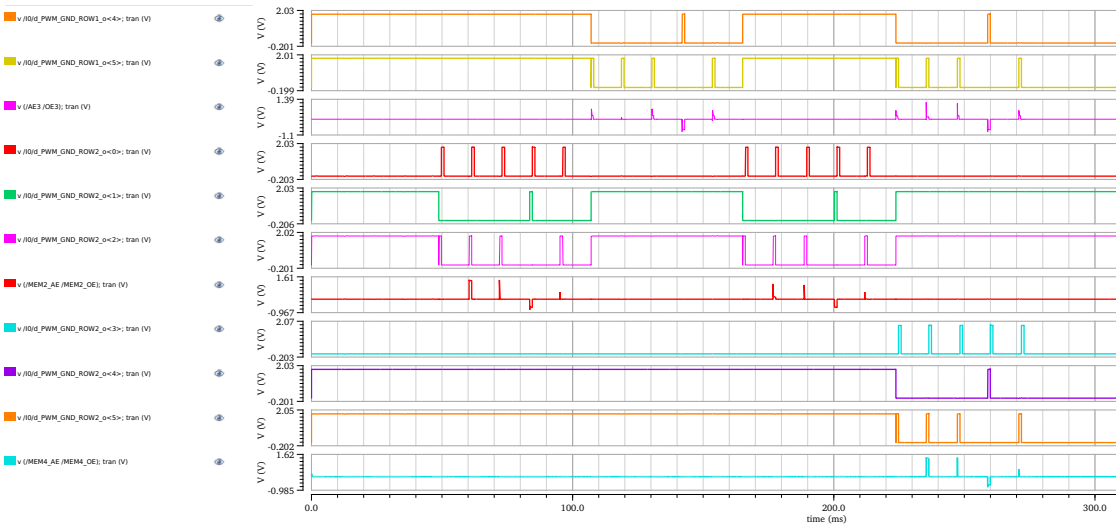


Figure 6.2: Voltage mode: PWM waveforms for the read, reset, read, set, read operation in memristor array

7 Conclusion

This thesis has explored the intricate mechanisms of memristor programming, leveraging advanced RISC-V processors and PWM generators to control these innovative technologies. By utilizing both voltage and current modes to manage memristor arrays, this research has made significant strides in advancing non-volatile memory and neuromorphic computing systems.

The proposed control algorithms have been rigorously validated through comprehensive simulations using Questa Sim and Cadence. These simulations demonstrate the system's ability to accurately program individual memristors and perform simultaneous operations across multiple columns, thereby enhancing the efficiency and speed of memory operations.

Additionally, this thesis emphasizes the RISC-V framework, showcasing its potential as a robust system-on-chip platform for developing cutting-edge computing systems. The seamless integration of analog and digital control blocks, guided by the algorithmic subroutines, represents a notable advancement in electronic design automation.

While error correction is not required for HRS and LRS, achieving stable intermediate states remains a challenge that warrants further investigation. To address the stability and variability of these intermediate states, future research could explore the application of fuzzy logic for error correction. Fuzzy logic, with its capacity to handle imprecise information, aligns well with the characteristics of memristors and could provide a viable approach to improving the precision of memristor state management.

Bibliography

[1] L. Chua. "Memristor-The missing circuit element". In: *IEEE Transactions on Circuit Theory* 18.5 (1971), pp. 507–519. DOI: 10.1109/TCT.1971.1083337.

[2] Dmitri B. Strukov et al. "The missing memristor found". In: *Nature* 453 (2008), pp. 80–83.

[3] Mahdi Zahedi et al. "MNEMOSENE: Tile Architecture and Simulator for Memristor-based Computation-in-memory". In: *ACM Journal on Emerging Technologies in Computing Systems* 18 (July 2022), pp. 1–24. DOI: 10.1145/3485824.

[4] Julien Borghetti et al. "'Memristive' switches enable 'stateful' logic operations via material implication". In: *Nature* 464.7290 (2010), pp. 873–876.

[5] Ulrich Böttger et al. "Picosecond multilevel resistive switching in tantalum oxide thin films". In: *Scientific reports* 10 (Oct. 2020), p. 16391. DOI: 10.1038/s41598-020-73254-2.

[6] Sung Jo, Kuk-Hwan Kim, and Wei Lu. "Programmable Resistance Switching in Nanoscale Two-Terminal Devices". In: *Nano letters* 9 (Jan. 2009), pp. 496–500. DOI: 10.1021/nl803669s.

[7] Jianhua Joshua Yang, Dmitri Strukov, and Duncan Stewart. "Memristive Devices for Computing". In: *Nature nanotechnology* 8 (Jan. 2013), pp. 13–24. DOI: 10.1038/nnano.2012.240.

[8] J. Joshua Yang, Dmitri B. Strukov, and Duncan R. Stewart. "Memristive devices for computing." In: *Nature nanotechnology* 8 1 (2013), pp. 13–24.

[9] Sung-Hyun Jo et al. "Nanoscale memristor device as synapse in neuromorphic systems." In: *Nano letters* 10 4 (2010), pp. 1297–301.

[10] F. Cüppers et al. "Exploiting the switching dynamics of HfO2-based ReRAM devices for reliable analog memristive behavior". In: *APL Materials* 7.9 (Sept. 2019), p. 091105. ISSN: 2166-532X. DOI: 10.1063/1.5108654. eprint: https://pubs.aip.org/aip/apm/article-pdf/doi/10.1063/1.5108654/14561333/091105_1_online.pdf.

[11] F. Cüppers et al. "Exploiting the switching dynamics of HfO2-based RRAM devices for reliable analog memristive behavior". In: *APL Materials* 7.9 (2019), p. 091105. DOI: 10.1063/1.5108654.

[12] Ming-Hsiu Lee et al. “Resistance control of transition metal oxide resistive memory (TMO ReRAM)”. In: Oct. 2016, pp. 182–185. DOI: 10.1109/ICSICT.2016.7998873.

[13] Kazuhide Higuchi, Tomoko Iwasaki, and Ken Takeuchi. “Investigation of Verify-Programming Methods to Achieve 10 Million Cycles for 50nm HfO2 ReRAM”. In: (May 2012). DOI: 10.1109/IMW.2012.6213665.

[14] Albert Cirera et al. “Current Driven Random Exploration of Resistive Switching Devices, an Opportunity to Improve Bit Error Ratio”. In: June 2023, pp. 1–4. DOI: 10.1109/CDE58627.2023.10339522.

[15] Héctor García et al. “Current Pulses to Control the Conductance in RRAM Devices”. In: *IEEE Journal of the Electron Devices Society* 8 (2020), pp. 291–296. DOI: 10.1109/JEDS.2020.2979293.

[16] E.R. Hsieh et al. “High-Density Multiple Bits-per-Cell 1T4R RRAM Array with Gradual SET/RESET and its Effectiveness for Deep Learning”. In: Dec. 2019, pp. 35.6.1–35.6.4. DOI: 10.1109/IEDM19573.2019.8993514.

[17] Binh Q. Le et al. “RADAR: A Fast and Energy-Efficient Programming Technique for Multiple Bits-Per-Cell RRAM Arrays”. In: *IEEE Transactions on Electron Devices* 68.9 (2021), pp. 4397–4403. DOI: 10.1109/TED.2021.3097975.

[18] Binh Q. Le et al. “Resistive RAM With Multiple Bits Per Cell: Array-Level Demonstration of 3 Bits Per Cell”. In: *IEEE Transactions on Electron Devices* 66.1 (2019), pp. 641–646. DOI: 10.1109/TED.2018.2879788.

[19] J. Büchel et al. “Gradient descent-based programming of analog in-memory computing cores”. In: *2022 International Electron Devices Meeting (IEDM)*. 2022, pp. 33.1.1–33.1.4. DOI: 10.1109/IEDM45625.2022.10019486.

[20] Yixuan Hu et al. “Error Correction Scheme for Reliable RRAM-Based In-Memory Computing”. In: *2021 5th IEEE Electron Devices Technology Manufacturing Conference (EDTM)*. 2021, pp. 1–3. DOI: 10.1109/EDTM50988.2021.9420944.

[21] Synopsys. *Efficient Bluespec RISC-V Processor Verification for Highest Coverage Closure: A Comprehensive Case Study*. https://www.synopsys.com/verification/solutions/risc-v.html. 2023.

[22] This work is part of the PhD research conducted by Neethu Kuriakose, focusing on memristor analog design, and Anugerah Firdauzi, specializing in ADC design. Both researchers are affiliated with the Central Institute of Engineering, Electronics and Analytics - Electronic Systems (ZEA-2), Forschungszentrum Jülich GmbH, Germany.

[23] Jonas Mair et al. “Rapid Prototyping Platform for Integrated Circuits for Quantum Computing”. In: *2024 International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*. Forschungszentrum Jülich, Germany, 2024.

[24] Synopsys, Inc. *ASIP Designer and Pre-Verified ASIP Models*. 2024.

8 Appendix

Table 8.1: Summary of memristor control analog registers

Config Register	**Description**
1	Defines modes of OPAMP for Row 1 in VMODE
2	Defines modes of OPAMP for Row 2 in VMODE
3	Defines modes of OPAMP for Row 1 in IMODE
4	Defines modes of OPAMP for Row 2 in IMODE
5	Defines modes of OPAMP for M5 in IMODE
6	Defines modes of OPAMP for M5 in VMODE
7	Selects widths of current compliance transistor for M5
8	Selects combination width of current compliance transistor with resistances in VAR block for IMODE
9	Selects memory states (READ/HRS/LRS) for M5
10	Supply select for column 1 and column 2
11/12/13/14	Selects mode of operation for M1/2/3/4
15	Selects internal/external DACs

Table 8.2: Summary of memristor control digital registers

Config Bits	**Description**
CLK_DIVIDER	CLK divider select value to reduce the clock frequency
CH1_PULSE_GEN_EN	PWM1 enable to make the block functional
CH1_PULSE_START_EN	PWM1 start trigger signal to start the counters
CH1_PULSE_REPETITION_EN	PWM1 repetition enable to repeat the pulses
CH1_PULSE_2_EN	PWM1 pulse2 enable to generate the read enable sub pulse
CH2_PULSE_GEN_EN	PWM2 enable to make the block functional

Table 8.2: (continued)

Config Register	Description
CH2_PULSE_START_EN	PWM2 start trigger signal to start the counters
CH2_PULSE_REPETITION_EN	PWM2 repetition enable to repeat the pulses
CH2_PULSE_2_EN	PWM2 pulse2 enable to generate the read enable sub pulse
SET_RESET_MODE	Set Reset Mode Enable
CURRENT_MODE	Current Mode enable
OPERATION_MODE	Operation mode enable
READ_EN_CH1	1st Row constant read value (when the pulse is not applied to the row)
READ_EN_CH2	2nd Row constant read value (when the pulse is not applied to the row)
CH1_CURRENT_DEFAULT	The 1st row default/constant value to drive in current mode (when Pulse is not applied)
CH1_CURRENT_PULSE	1st row current pulse enable (To drive the pulse in the current mode)
CH1_1_DEFAULT_1	1st row 1st memristor(M1) constant value for terminal 1, when the pulse is not applied
CH1_1_DEFAULT_2	1st row 1st memristor(M1) constant value for terminal 2 & 3, when the pulse is not applied
CH1_2_DEFAULT_1	1st row 2nd memristor(M2) constant value for terminal 1, when the pulse is not applied
CH1_2_DEFAULT_2	1st row 2nd memristor(M2) constant value for terminal 2 & 3, when the pulse is not applied
CH1_SEL	Value to select the terminals of M1 & M2(1st row) to drive the pulse
CH2_CURRENT_DEFAULT	The 2nd row default/constant value to drive in current mode (when Pulse is not applied)
CH2_CURRENT_PULSE	2nd row current pulse enable (To drive the pulse in the current mode)

Table 8.2: (continued)

Config Register	Description
CH2_1_DEFAULT_1	2nd row 1st memristor(M3) constant value for terminal 1, when the pulse is not applied
CH2_1_DEFAULT_2	2nd row 1st memristor(M3) constant value for terminal 2 & 3, when the pulse is not applied
CH2_2_DEFAULT_1	2nd row 2nd memristor(M4) constant value for terminal 1, when the pulse is not applied
CH2_2_DEFAULT_2	2nd row 2nd memristor(M4) constant value for terminal 2 & 3, when the pulse is not applied
WRITE_CH1_1	1st row M1 Write enable signal
WRITE_CH1_2	1st row M2 Write enable signal
WRITE_CH2_1	2nd row M3 Write enable signal
WRITE_CH2_2	2nd row M4 Write enable signal
CH2_SEL	Value to select the terminals of M3 & M4(2nd row) to drive the pulse
VAR_SEL	Value to select the terminals of variable Memristor to drive the pulse
VAR_DEFAULT_1	Variable memristor constant value for terminal 1, when the pulse is not applied
VAR_DEFAULT_2	Variable memristor constant value for terminal 2 & 3, when the pulse is not applied
VAR_CURRENT_DEFAULT	Variable Memristor default/constant value to drive in current mode (when Pulse is not applied)
VAR_CURRENT_PULSE	Variable Memristor current pulse enable (To drive the pulse in the current mode)
CNT_SEL	Value to select the counter value to load in the registers
CH1_CNT_VAL_UPDATE	Update the value of the counters for 1st row
CH2_CNT_VAL_UPDATE	Update the value of the counters for 2nd row

Table 8.2: (continued)

Config Register	**Description**
INV_PULSE	Option to invert the main pulse applied on terminals
COUNTER_VAL_LSB	First byte of counter value
COUNTER_VAL_MSB	Second byte of counter value
AUTO_DEFAULT_VAL_SET	To select the setting of default values programmed on terminals when the pulse counter is done

```
1
2  #include <stdbool.h>
3  #include <stdlib.h>
4  #include <stdint.h>
5  #include "stdio.h"
6  #include <math.h>
7  #define DUT_OFFSET 0x0002000 *4
8
9  #define CONFIG_MEMRISTORCTRL_ANALOG_1   0   *4
10 #define CONFIG_MEMRISTORCTRL_ANALOG_2   1   *4
11 #define CONFIG_MEMRISTORCTRL_ANALOG_3   2   *4
12 #define CONFIG_MEMRISTORCTRL_ANALOG_4   3   *4
13 #define CONFIG_MEMRISTORCTRL_ANALOG_5   4   *4
14 #define CONFIG_MEMRISTORCTRL_ANALOG_6   5   *4
15 #define CONFIG_MEMRISTORCTRL_ANALOG_7   6   *4
16 #define CONFIG_MEMRISTORCTRL_ANALOG_8   7   *4
17 #define CONFIG_MEMRISTORCTRL_ANALOG_9   8   *4
18 #define CONFIG_MEMRISTORCTRL_ANALOG_10  9   *4
19 #define CONFIG_MEMRISTORCTRL_ANALOG_11  10  *4
20 #define CONFIG_MEMRISTORCTRL_ANALOG_12  11  *4
21 #define CONFIG_MEMRISTORCTRL_ANALOG_13  12  *4
22 #define CONFIG_MEMRISTORCTRL_ANALOG_14  13  *4
23 #define CONFIG_MEMRISTORCTRL_ANALOG_15  14  *4
24 #define CONFIG_MEMRISTORCTRL_ANALOG_16  15  *4
25
26 #define CONFIG_MEMRISTORCTRL_DIG_1      16  *4
27 #define CONFIG_MEMRISTORCTRL_DIG_2      17  *4
28 #define CONFIG_MEMRISTORCTRL_DIG_3      18  *4
29 #define CONFIG_MEMRISTORCTRL_DIG_4      19  *4
30 #define CONFIG_MEMRISTORCTRL_DIG_5      20  *4
31 #define CONFIG_MEMRISTORCTRL_DIG_6      21  *4
32 #define CONFIG_MEMRISTORCTRL_DIG_7      22  *4
33 #define CONFIG_MEMRISTORCTRL_DIG_8      23  *4
34 #define CONFIG_MEMRISTORCTRL_DIG_9      24  *4
35 #define CONFIG_MEMRISTORCTRL_DIG_10     25  *4
36 #define CONFIG_MEMRISTORCTRL_DIG_11     26  *4
37 #define CONFIG_MEMRISTORCTRL_DIG_12     27  *4
38
39 #define CONFIG_MEMRISTORCTRL_DAC1_1     28  *4
40 #define CONFIG_MEMRISTORCTRL_DAC1_2     29  *4
41 #define CONFIG_MEMRISTORCTRL_DAC2_1     30  *4
42 #define CONFIG_MEMRISTORCTRL_DAC2_2     31  *4
```

```
43
44 #define CONFIG_MEMRISTORCTRL_Bandgap_1  32  *4
45 #define CONFIG_MEMRISTORCTRL_Bandgap_2  33  *4
46 #define CONFIG_MEMRISTORCTRL_Bandgap_3  34  *4
47 #define CONFIG_MEMRISTORCTRL_Bandgap_4  35  *4
48 #define CONFIG_MEMRISTORCTRL_Bandgap_5  36  *4
49
50 //CONFIG_MEMRISTORCTRL_DIG_1
51 #define CLK_DIVIDER_MASK                0x07
52 //#define CLK_DIVIDER_OFFSET              0
53
54 //CONFIG_MEMRISTORCTRL_DIG_2
55 #define CH1_PULSE_GEN_EN_MASK           0x01
56 #define CH1_PULSE_START_EN_MASK         0x02
57 #define CH1_PULSE_REPETITION_EN_MASK    0x04
58 #define CH1_PULSE_2_EN_MASK             0x08
59 #define CH2_PULSE_GEN_EN_MASK           0x10
60 #define CH2_PULSE_START_EN_MASK         0x20
61 #define CH2_PULSE_REPETITION_EN_MASK    0x40
62 #define CH2_PULSE_2_EN_MASK             0x80
63 #define CH1_CH2_PULSE_2_EN_MASK         0x88
64 #define CH1_CH2_PULSE_START_EN_MASK     0x22
65
66 //CONFIG_MEMRISTORCTRL_DIG_3
67 #define SET_RESET_MODE_MASK             0x01
68 #define CURRENT_MODE_MASK               0x02
69 #define OPERATION_MODE_MASK             0x04
70 #define READ_EN_CH1_MASK                0x08
71 #define READ_EN_CH2_MASK                0x10
72 #define CH1_CURRENT_DEFAULT_MASK        0x20
73 #define CH1_CURRENT_PULSE_MASK          0x40
74 #define CH1_1_DEFAULT_1_MASK            0x80
75
76 //CONFIG_MEMRISTORCTRL_DIG_4
77 #define CH1_1_DEFAULT_2_MASK            0x01
78 #define CH1_2_DEFAULT_1_MASK            0x02
79 #define CH1_2_DEFAULT_2_MASK            0x04
80 #define CH1_SEL_MASK                    0x38
81 #define CH1_SEL_OFFSET                  3
82 #define CH2_CURRENT_DEFAULT_MASK        0x40
83 #define CH2_CURRENT_PULSE_MASK          0x80
84
85 //CONFIG_MEMRISTORCTRL_DIG_5
```

```
86 #define CH2_1_DEFAULT_1_MASK            0x01
87 #define CH2_1_DEFAULT_2_MASK            0x02
88 #define CH2_2_DEFAULT_1_MASK            0x04
89 #define CH2_2_DEFAULT_2_MASK            0x08
90 #define WRITE_CH1_1_MASK                0x10
91 #define WRITE_CH1_2_MASK                0x20
92 #define WRITE_CH2_1_MASK                0x40
93 #define WRITE_CH2_2_MASK                0x80
94 #define WRITE_ALL_CH_MASK               0xf0
95
96 //CONFIG_MEMRISTORCTRL_DIG_6
97 #define CH2_SEL_MASK                    0x07
98 #define CH2_SEL_OFFSET                  0
99 #define VAR_SEL_MASK                    0x38
100 #define VAR_SEL_OFFSET                  3
101 #define VAR_DEFAULT_1_MASK              0x40
102 #define VAR_DEFAULT_2_MASK              0x80
103
104 //CONFIG_MEMRISTORCTRL_DIG_7
105 #define VAR_CURRENT_DEFAULT_MASK        0x01
106 #define VAR_CURRENT_PULSE_MASK          0x02
107 #define CNT_SEL_MASK                    0x1c
108 #define CNT_SEL_OFFSET                  2
109 #define CH1_CNT_VAL_UPDATE_MASK         0x20
110 #define CH2_CNT_VAL_UPDATE_MASK         0x40
111 #define INV_PULSE_MASK                  0x80
112
113 //CONFIG_MEMRISTORCTRL_DIG_8
114 #define COUNTER_VAL_LSB_MASK            0xff
115 //#define COUNTER_VAL_LSB_OFFSET
116
117 //CONFIG_MEMRISTORCTRL_DIG_9
118 #define COUNTER_VAL_MSB_MASK            0xff
119 //#define COUNTER_VAL_LSB_OFFSET
120
121 //CONFIG_MEMRISTORCTRL_DIG_10
122 #define AUTO_DEFAULT_VAL_SET_MASK       0x01
123
124
125 //int control_reg = 0;
126
127 ////global variable declaration////
128 #define CLK_DIV                         5
```

```
129
130 void delay()
131 {
132     for (volatile int i = 0 ; i < pow(2,CLK_DIV)/7-1; i++)
133     {
134     }
135     return;
136 }
137
138 void delay_wait_for_pwm()
139 {
140     for (volatile int i = 0 ; i < 700000/7-1; i++)
141     {
142     }
143     return;
144 }
145
146 //////memristor type ////////////////////////
147 typedef struct {
148     int mem_pos;
149     int pulse_w;
150     int pulse_T;
151     int pulse_rep;
152     int subpulse_w;
153     int subpulse_delay;
154     int ext_DAC;
155     int int_DAC;
156     bool on;
157 } memristor;
158
159 ////END DIGITAL PART////
160
161
162 //// Default Mode Start ////
163
164 //1 is driven to 1st row M terminal 2 and 3 when the pulse
    ↪  is not enabled
165 void DUT_resetChDefault1_var_mode(){
166
167     int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_DIG_6);
168     *p_control &= ~ VAR_DEFAULT_1_MASK; //
        ↪ DUT_setCh22Default2
```

```
169     delay();
170
171     return;
172 }
173
174 void DUT_setChDefault1_var_mode(){
175
176     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_6);
177     *p_control |= ~ VAR_DEFAULT_1_MASK; //
            ↪ DUT_setCh22Default2
178     delay();
179
180     return;
181 }
182
183 //0 is driven to 1st row M terminal 2 and 3 when the pulse
        ↪  is not enabled
184 void DUT_setChDefault2_var_mode(){
185
186     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_6);
187     *p_control |= VAR_DEFAULT_2_MASK; //
            ↪ DUT_setCh22Default2
188     delay();
189
190     return;
191 }
192
193 void DUT_resetChDefault2_var_mode(){
194
195     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_6);
196     *p_control &= ~ VAR_DEFAULT_2_MASK; //
            ↪ DUT_setCh22Default2
197     delay();
198
199     return;
200 }
201
202 //1 is driven to 1st row M terminal 1 when the pulse is
        ↪ not enabled
203 void DUT_setChDefault1(int Ch){
```

```
204     switch(Ch)
205     {
206         case 11:
207             {
208             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_3);
209             *p_control |= CH1_1_DEFAULT_1_MASK; //
                    ↪ DUT_setCh11Default1
210             delay();
211             break;
212             }
213         case 12:
214             {
215             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_4);
216             *p_control |= CH1_2_DEFAULT_1_MASK; //
                    ↪ DUT_setCh12Default1
217             delay();
218             break;
219             }
220         case 21:
221             {
222             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_5);
223             *p_control |= CH2_1_DEFAULT_1_MASK; //
                    ↪ DUT_setCh21Default1
224             delay();
225             break;
226             }
227         case 22:
228             {
229             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_5);
230             *p_control |= CH2_2_DEFAULT_1_MASK; //
                    ↪ DUT_setCh22Default1
231             delay();
232             break;
233             }
234         case 31:
235         {
236             DUT_setChDefault1_var_mode();
237             break;
238         }
```

```
239     }
240     return;
241 }
242 
243 //0 is driven to 1st row M terminal 1 when the pulse is
    ↪ not enabled
244 void DUT_resetChDefault1(int Ch){
245     switch(Ch)
246     {
247         case 11:
248         {
249             int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_DIG_3);
250             *p_control &= ~CH1_1_DEFAULT_1_MASK; //
                ↪ DUT_clearCh11Default1
251             delay();
252             break;
253         }
254         case 12:
255         {
256             int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_DIG_4);
257             *p_control &= ~CH1_2_DEFAULT_1_MASK; //
                ↪ DUT_clearCh12Default1
258             delay();
259             break;
260         }
261         case 21:
262         {
263             int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_DIG_5);
264             *p_control &= ~CH2_1_DEFAULT_1_MASK; //
                ↪ DUT_clearCh21Default1
265             delay();
266             break;
267         }
268         case 22:
269         {
270             int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_DIG_5);
271             *p_control &= ~CH2_2_DEFAULT_1_MASK; //
                ↪ DUT_clearCh22Default1
272             delay();
```

```
273             break;
274         }
275         case 31:
276         {
277             DUT_resetChDefault1_var_mode();
278             break;
279         }
280     }
281     return;
282 }
283
284 //1 is driven to 1st row M terminal 2 and 3 when the pulse
    ↪  is not enabled
285 void DUT_setChDefault2(int Ch){
286     switch(Ch)
287     {
288         case 11:
289         {
290             int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_DIG_4);
291             *p_control |= CH1_1_DEFAULT_2_MASK; //
                ↪ DUT_setCh11Default2
292             delay();
293             break;
294         }
295         case 12:
296         {
297             int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_DIG_4);
298             *p_control |= CH1_2_DEFAULT_2_MASK; //
                ↪ DUT_setCh12Default2
299             delay();
300             break;
301         }
302         case 21:
303         {
304             int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_DIG_5);
305             *p_control |= CH2_1_DEFAULT_2_MASK; //
                ↪ DUT_setCh21Default2
306             delay();
307             break;
308         }
```

```
309             case 22:
310             {
311                 int* p_control= (int*)(DUT_OFFSET+
                        ↪ CONFIG_MEMRISTORCTRL_DIG_5);
312                 *p_control |= CH2_2_DEFAULT_2_MASK; //
                        ↪ DUT_setCh22Default2
313                 delay();
314                 break;
315             }
316 
317             case 31:
318             {
319                 DUT_setChDefault2_var_mode();
320                 break;
321             }
322         }
323         return;
324 }
325 
326 
327 //0 is driven to 1st row M terminal 2 and 3 when the pulse
        ↪  is not enabled
328 void DUT_resetChDefault2(int Ch){
329         switch(Ch)
330         {
331             case 11:
332             {
333                 int* p_control= (int*)(DUT_OFFSET+
                        ↪ CONFIG_MEMRISTORCTRL_DIG_4);
334                 *p_control &= ~CH1_1_DEFAULT_2_MASK; //
                        ↪ DUT_clearCh11Default2
335                 delay();
336                 break;
337             }
338             case 12:
339             {
340                 int* p_control= (int*)(DUT_OFFSET+
                        ↪ CONFIG_MEMRISTORCTRL_DIG_4);
341                 *p_control &= ~CH1_2_DEFAULT_2_MASK; //
                        ↪ DUT_clearCh12Default2
342                 delay();
343                 break;
344             }
```

```
345         case 21:
346         {
347              int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_5);
348             *p_control &= ~CH2_1_DEFAULT_2_MASK; //
                    ↪ DUT_clearCh21Default2
349             delay();
350             break;
351         }
352         case 22:
353         {
354             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_5);
355             *p_control &= ~CH2_2_DEFAULT_2_MASK; //
                    ↪ DUT_clearCh22Default2
356             delay();
357             break;
358         }
359         case 31:
360         {
361             DUT_resetChDefault2_var_mode();
362             break;
363         }
364     }
365     return;
366 }
367 /// PWM generation ///
368 //Enable the block functional
369 void DUT_setChPulseGenEn(int Ch){
370     switch(Ch)
371     {
372         case 11:
373         case 12:
374         {
375             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
376             *p_control |= CH1_PULSE_GEN_EN_MASK; //
                    ↪ DUT_setCh1PulseGenEn
377             delay();
378             break;
379         }
380         case 21:
381         case 22:
```

```
382         {
383             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
384             *p_control |= CH2_PULSE_GEN_EN_MASK;     //
                    ↪ DUT_setCh2PulseGenEn
385             delay();
386             break;
387         }
388     }
389     return;
390 }
391
392 void DUT_set_allChPulseGenEn(){
393
394   int* p_control_CH= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_DIG_2);
395   *p_control_CH  |=  (CH1_PULSE_GEN_EN_MASK +
        ↪ CH2_PULSE_GEN_EN_MASK); // DUT_setCh1PulseGenEn
396   delay();
397   return;
398 }
399
400
401 void DUT_setCh1CntValUpdate(){
402     int* p_control= (int*)(DUT_OFFSET+
          ↪ CONFIG_MEMRISTORCTRL_DIG_7);
403     *p_control |= CH1_CNT_VAL_UPDATE_MASK;
404     delay();
405     return;
406 }
407
408 void DUT_clearCh1CntValUpdate(){
409     int* p_control= (int*)(DUT_OFFSET+
          ↪ CONFIG_MEMRISTORCTRL_DIG_7);
410     *p_control &= ~CH1_CNT_VAL_UPDATE_MASK;
411     delay();
412     return;
413 }
414
415 void DUT_setCh2CntValUpdate(){
416     int* p_control= (int*)(DUT_OFFSET+
          ↪ CONFIG_MEMRISTORCTRL_DIG_7);
417     *p_control |= CH2_CNT_VAL_UPDATE_MASK;
```

```
418     delay();
419     return;
420 }
421
422 void DUT_clearCh2CntValUpdate(){
423     int* p_control= (int*)(DUT_OFFSET+
          ↪ CONFIG_MEMRISTORCTRL_DIG_7);
424     *p_control &= ~CH2_CNT_VAL_UPDATE_MASK;
425     delay();
426     return;
427 }
428
429 void DUT_reset_allChCntValUpdate(){
430     int* p_control= (int*)(DUT_OFFSET+
          ↪ CONFIG_MEMRISTORCTRL_DIG_7);
431     *p_control &= ~(CH1_CNT_VAL_UPDATE_MASK +
          ↪ CH2_CNT_VAL_UPDATE_MASK);
432     delay();
433     return;
434 }
435
436 void DUT_set_allCh2CntValUpdate(){
437     int* p_control= (int*)(DUT_OFFSET+
          ↪ CONFIG_MEMRISTORCTRL_DIG_7);
438     *p_control |= (CH1_CNT_VAL_UPDATE_MASK +
          ↪ CH2_CNT_VAL_UPDATE_MASK);
439     delay();
440     return;
441 }
442
443
444 void DUT_ChCntValUpdate(int Ch){
445     switch(Ch)
446     {
447         case 11:
448         case 12:
449             DUT_setCh1CntValUpdate();
450             DUT_clearCh1CntValUpdate();
451             break;
452         case 21:
453         case 22:
454             DUT_setCh2CntValUpdate();
455             DUT_clearCh2CntValUpdate();
```

```
456                 break;
457         }
458         return;
459 }
460
461
462 void DUT_setCntSel(int CntSel){
463         int* p_control= (int*)(DUT_OFFSET+
              ↪ CONFIG_MEMRISTORCTRL_DIG_7);
464         *p_control = ((CntSel << CNT_SEL_OFFSET) &
              ↪ CNT_SEL_MASK) | (*p_control & ~CNT_SEL_MASK);
465         delay();
466         return;
467 }
468
469 void DUT_setCounterValLsb(int CounterValLsb){
470         int* p_control= (int*)(DUT_OFFSET+
              ↪ CONFIG_MEMRISTORCTRL_DIG_8);
471         *p_control = CounterValLsb & COUNTER_VAL_LSB_MASK;
472         delay();
473         return;
474 }
475
476 void DUT_setCounterValMsb(int CounterValMsb){
477         int* p_control= (int*)(DUT_OFFSET+
              ↪ CONFIG_MEMRISTORCTRL_DIG_9);
478         *p_control = (CounterValMsb >> 8) &
              ↪ COUNTER_VAL_MSB_MASK;
479         delay();
480         return;
481 }
482
483 void DUT_resetCounterValMsb(){
484         int* p_control= (int*)(DUT_OFFSET+
              ↪ CONFIG_MEMRISTORCTRL_DIG_9);
485         *p_control =  0x00;
486         delay();
487         return;
488 }
489
490 void DUT_setChPulsePara(int Ch, int pul_w, int pulse_T,
      ↪ int rep){
491         switch(Ch)
```

```
492     {
493         case 11:
494         case 12:
495         {          // pulse period setting : 16bits
496                 DUT_setCntSel(0);
497                 DUT_setCounterValLsb(pulse_T);
498                 DUT_setCounterValMsb(pulse_T);
499                 DUT_ChCntValUpdate(12);
500                  // pulse width setting : 16bits
501                 DUT_setCntSel(1);
502                 DUT_setCounterValLsb(pul_w);
503                 DUT_setCounterValMsb(pul_w);
504                 DUT_ChCntValUpdate(12);
505                 // pulse repeition setting : 8bits
506                 DUT_setCntSel(4);
507                 DUT_setCounterValLsb(rep);
508                 DUT_ChCntValUpdate(12);
509             break;
510         }
511         case 21:
512         case 22:
513         {
514                 // pulse period setting
515                 DUT_setCntSel(0);
516                 DUT_setCounterValLsb(pulse_T);
517                 DUT_setCounterValMsb(pulse_T);
518                 DUT_ChCntValUpdate(22);
519                  // pulse width setting
520                 DUT_setCntSel(1);
521                 DUT_setCounterValLsb(pul_w);
522                 DUT_setCounterValMsb(pul_w);
523                 DUT_ChCntValUpdate(22);
524                 // pulse repeition setting
525                 DUT_setCntSel(4);
526                 DUT_setCounterValLsb(rep);
527                 DUT_ChCntValUpdate(22);
528             break;
529         }
530     }
531     return;
532 }
533
534
```

```
535 void DUT_setChPulseRepetitionEn(int Ch){
536     switch(Ch)
537     {
538         case 11:
539         case 12:
540         {
541             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
542             *p_control |= CH1_PULSE_REPETITION_EN_MASK; //
                    ↪ DUT_setCh1PulseRepetitionEn
543             delay();
544             break;
545         }
546         case 21:
547         case 22:
548         {
549             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
550             *p_control |= CH2_PULSE_REPETITION_EN_MASK; //
                    ↪ DUT_setCh2PulseRepetitionEn
551             delay();
552             break;
553         }
554     }
555     return;
556 }
557
558 void DUT_set_allChPulseRepetitionEn(){
559
560    int* p_control_CH= (int*)(DUT_OFFSET+
          ↪ CONFIG_MEMRISTORCTRL_DIG_2);
561   *p_control_CH |= (CH1_PULSE_REPETITION_EN_MASK +
        ↪ CH2_PULSE_REPETITION_EN_MASK); //
        ↪ DUT_setCh2PulseRepetitionEn
562    delay();
563    return;
564
565
566 }
567
568
569 //Enable start
570 void DUT_setChPulseStartEn(int Ch){
```

```
571     switch(Ch)
572     {
573         case 11:
574         case 12:
575         {
576             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
577             *p_control |= CH1_PULSE_START_EN_MASK; //
                    ↪ DUT_setCh1PulseStartEn
578             delay();
579             break;
580         }
581         case 21:
582         case 22:
583         {
584             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
585             *p_control |= CH2_PULSE_START_EN_MASK; //
                    ↪ DUT_setCh2PulseStartEn
586             delay();
587             break;
588         }
589     }
590     return;
591 }
592
593 void DUT_set_multiple_PulseStartEn(bool COL_1ST, bool
    ↪ COL_2ND)
594 {
595       if (COL_1ST == true && COL_2ND == false)
596       {
597             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
598             *p_control |= CH1_PULSE_START_EN_MASK; //
                    ↪ DUT_setCh1PulseStartEn
599             delay();
600       }
601
602       if (COL_2ND == true && COL_1ST == false )
603       {
604             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
605             *p_control |= CH2_PULSE_START_EN_MASK; //
```

```
                ↪ DUT_setCh1PulseStartEn
606             delay();
607         }
608
609         if (COL_2ND == true && COL_1ST == true )
610         {
611                 int* p_control= (int*)(DUT_OFFSET+
                        ↪ CONFIG_MEMRISTORCTRL_DIG_2);
612             *p_control |= CH1_CH2_PULSE_START_EN_MASK;
613             delay();
614         }
615
616     return;
617 }
618
619 void DUT_set_allChPulseStartEn(){
620
621    int* p_control= (int*)(DUT_OFFSET+
          ↪ CONFIG_MEMRISTORCTRL_DIG_2);
622   *p_control |= CH1_CH2_PULSE_START_EN_MASK;
623   delay();
624
625 }
626
627
628 void DUT_resetChPulseStartEn(int Ch){
629     switch(Ch)
630     {
631         case 11:
632         case 12:
633         {
634             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
635             *p_control &= ~CH1_PULSE_START_EN_MASK; //
                    ↪ DUT_setCh1PulseStartEn
636             delay();
637             break;
638         }
639         case 21:
640         case 22:
641         {
642             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
```

```
643             *p_control &= ~CH2_PULSE_START_EN_MASK; //
                    ↪ DUT_setCh2PulseStartEn
644             delay();
645             break;
646         }
647     }
648     return;
649 }
650
651
652 void DUT_reset_allChPulseStartEn(){
653    int* p_control= (int*)(DUT_OFFSET+
          ↪ CONFIG_MEMRISTORCTRL_DIG_2);
654   *p_control &= ~ CH1_CH2_PULSE_START_EN_MASK;
655   delay();
656
657 }
658
659
660 //enable the sub pulses
661 void DUT_setChPulse2En(int Ch){
662     switch(Ch)
663     {
664         case 11:
665         case 12:
666         {
667             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
668             *p_control |= CH1_PULSE_2_EN_MASK; //
                    ↪ DUT_setCh1Pulse2En
669             delay();
670             break;
671         }
672         case 21:
673         case 22:
674         {
675             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
676             *p_control |= CH2_PULSE_2_EN_MASK; //
                    ↪ DUT_setCh2Pulse2En
677             delay();
678             break;
679         }
```

```
680     }
681     return;
682 }
683
684
685 void DUT_set_allChPulse2En(){
686
687     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_2);
688     *p_control |= CH1_CH2_PULSE_2_EN_MASK;
689     delay();
690     return;
691 }
692
693
694 void DUT_resetChPulse2En(int Ch){
695     switch(Ch)
696     {
697         case 11:
698         case 12:
699         {
700             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
701             *p_control &= ~ CH1_PULSE_2_EN_MASK; //
                    ↪ DUT_setCh1Pulse2En
702             delay();
703             break;
704         }
705         case 21:
706         case 22:
707         {
708             int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_2);
709             *p_control &= ~ CH2_PULSE_2_EN_MASK; //
                    ↪ DUT_setCh2Pulse2En
710             delay();
711             break;
712         }
713     }
714     return;
715 }
716
717
```

```
718 void DUT_reset_allChPulse2En(){
719
720     int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_DIG_2);
721     *p_control &= ~ CH1_CH2_PULSE_2_EN_MASK ; //
        ↪ DUT_setCh1Pulse2En
722     delay();
723
724     return;
725 }
726
727 /// PWM generation for sub pulses ///
728 void DUT_setCh_subPulsePara(int Ch, int sub_pul_delay, int
    ↪  sub_pul_w){
729
730     switch(Ch)
731     {
732         case 11:
733         case 12:
734         {
735                 // pulse delay setting
736                 DUT_setCntSel(2);
737                 DUT_setCounterValLsb(sub_pul_delay);
738                 DUT_setCounterValMsb(sub_pul_delay);
739                 DUT_ChCntValUpdate(12);
740                  // pulse width setting
741                 DUT_setCntSel(3);
742                 DUT_setCounterValLsb(sub_pul_w);
743                 DUT_setCounterValMsb(sub_pul_w);
744                 DUT_ChCntValUpdate(12);
745             break;
746         }
747         case 21:
748         case 22:
749         {
750                 // pulse delay setting
751                 DUT_setCntSel(2);
752                 DUT_setCounterValLsb(sub_pul_delay);
753                 DUT_setCounterValMsb(sub_pul_delay);
754                 //DUT_setCounterValMsb(0);
755                 DUT_ChCntValUpdate(22);
756                  // pulse width setting
757                 DUT_setCntSel(3);
```

```
758                 DUT_setCounterValLsb(sub_pul_w);
759                 DUT_setCounterValMsb(sub_pul_w);
760                 DUT_ChCntValUpdate(22);
761             break;
762         }
763     }
764     return;
765 }
766 
767 void DUT_setCh_subPulsePara_MSB(int Ch, int sub_pul_delay,
    ↪  int sub_pul_w){
768     switch(Ch)
769     {
770         case 11:
771         case 12:
772         {
773                 // pulse delay setting
774                 DUT_setCntSel(2);
775                 DUT_setCounterValMsb(sub_pul_delay);
776                 DUT_ChCntValUpdate(12);
777                  // pulse width setting
778                 DUT_setCntSel(3);
779                 DUT_setCounterValMsb(sub_pul_w);
780                 DUT_ChCntValUpdate(12);
781             break;
782         }
783         case 21:
784         case 22:
785         {
786                 // pulse delay setting
787                 DUT_setCntSel(2);
788                 DUT_setCounterValMsb(sub_pul_delay);
789                 DUT_ChCntValUpdate(22);
790                  // pulse width setting
791                 DUT_setCntSel(3);
792                 DUT_setCounterValMsb(sub_pul_w);
793                 DUT_ChCntValUpdate(22);
794             break;
795         }
796     }
797     return;
798 }
799 //// Channel Selection to Perform PWM ////
```

```
800 void DUT_setCh_Pulse(int Ch){
801     switch(Ch)
802     {
803         case 11:
804         {
805                 int* p_control= (int*)(DUT_OFFSET+
                        ↪ CONFIG_MEMRISTORCTRL_DIG_4); //
                        ↪ DUT_setCh1Sel
806                 *p_control = ((1 << CH1_SEL_OFFSET) &
                        ↪ CH1_SEL_MASK) | (*p_control & ~
                        ↪ CH1_SEL_MASK);
807                 delay();
808             break;
809         }
810         case 12:
811         {
812                 int* p_control= (int*)(DUT_OFFSET+
                        ↪ CONFIG_MEMRISTORCTRL_DIG_4); //
                        ↪ DUT_setCh1Sel
813                 *p_control = ((5 << CH1_SEL_OFFSET) &
                        ↪ CH1_SEL_MASK) | (*p_control & ~
                        ↪ CH1_SEL_MASK);
814                 delay();
815             break;
816         }
817         case 21:
818         {
819                  int* p_control= (int*)(DUT_OFFSET+
                        ↪ CONFIG_MEMRISTORCTRL_DIG_6); //
                        ↪ DUT_setCh2Sel
820                  *p_control = ((1 << CH2_SEL_OFFSET) &
                        ↪ CH2_SEL_MASK) | (*p_control & ~
                        ↪ CH2_SEL_MASK);
821                  delay();
822             break;
823         }
824         case 22:
825         {
826                  int* p_control= (int*)(DUT_OFFSET+
                        ↪ CONFIG_MEMRISTORCTRL_DIG_6); //
                        ↪ DUT_setCh2Sel
827                  *p_control = ((5 << CH2_SEL_OFFSET) &
                        ↪ CH2_SEL_MASK) | (*p_control & ~
```

```
                    ↪ CH2_SEL_MASK);
828                 delay();
829             break;
830         }
831     }
832     return;
833 }
834
835 void DUT_resetCh_Pulse(int Ch){
836     switch(Ch)
837     {
838         case 11:
839         {
840                 int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_4); //
                    ↪ DUT_setCh1Sel
841                 *p_control = ((3 << CH1_SEL_OFFSET) &
                    ↪ CH1_SEL_MASK) | (*p_control & ~
                    ↪ CH1_SEL_MASK);
842                 delay();
843             break;
844         }
845         case 12:
846         {
847                 int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_4); //
                    ↪ DUT_setCh1Sel
848                 *p_control = ((7 << CH1_SEL_OFFSET) &
                    ↪ CH1_SEL_MASK) | (*p_control & ~
                    ↪ CH1_SEL_MASK);
849                 delay();
850             break;
851         }
852         case 21:
853         {
854                 int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_6); //
                    ↪ DUT_setCh2Sel
855                 *p_control = ((3 << CH2_SEL_OFFSET) &
                    ↪ CH2_SEL_MASK) | (*p_control & ~
                    ↪ CH2_SEL_MASK);
856                 delay();
857             break;
```

```
858         }
859         case 22:
860         {
861                 int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_6);
862                 *p_control = ((7 << CH2_SEL_OFFSET) &
                    ↪ CH2_SEL_MASK) | (*p_control & ~
                    ↪ CH2_SEL_MASK);
863                 delay();
864             break;
865         }
866     }
867     return;
868 }
869
870 void DUT_setCh_Pulse_var_mode_pwm1(){
871
872     int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_DIG_6);
873     *p_control = ((1 << VAR_SEL_OFFSET) & VAR_SEL_MASK) |
        ↪ (*p_control & ~VAR_SEL_MASK);
874     delay();
875      return;
876 }
877
878 void DUT_resetCh_Pulse_var_mode_pwm1(){
879
880     int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_DIG_6);
881     *p_control = ((3 << VAR_SEL_OFFSET) & VAR_SEL_MASK) |
        ↪ (*p_control & ~VAR_SEL_MASK);
882     delay();
883      return;
884 }
885
886 void DUT_default_Ch_Pulse_var_mode(){
887
888     int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_DIG_6);
889     *p_control = ((0 << VAR_SEL_OFFSET) & VAR_SEL_MASK) |
        ↪ (*p_control & ~VAR_SEL_MASK);
890     delay();
891      return;
```

```
892 }
893
894 void DUT_default_Ch_Pulse_var_Imode(){
895
896    int* p_control= (int*)(DUT_OFFSET+
       ↪ CONFIG_MEMRISTORCTRL_DIG_7);
897    *p_control &= ~ VAR_CURRENT_DEFAULT_MASK;
898    delay();
899     return;
900 }
901
902
903 void DUT_default_Ch1_Pulse(){
904
905         int* p_control_ch1= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_4);
906         *p_control_ch1 = ((0 << CH1_SEL_OFFSET) &
            ↪ CH1_SEL_MASK) | (*p_control_ch1 & ~
            ↪ CH1_SEL_MASK);
907         delay();
908     return;
909
910 }
911
912 void DUT_default_Ch2_Pulse(){
913
914         int* p_control_ch2= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_6);
915         *p_control_ch2 = ((0 << CH2_SEL_OFFSET) &
            ↪ CH2_SEL_MASK) | (*p_control_ch2 & ~
            ↪ CH2_SEL_MASK);
916         delay();
917     return;
918
919 }
920
921 void DUT_default_Ch_Pulse(int Ch){
922     switch(Ch)
923     {
924         case 11:
925         case 12:
926         {
927                 int* p_control_ch1= (int*)(DUT_OFFSET+
```

```
                    ↪ CONFIG_MEMRISTORCTRL_DIG_4);
928                 *p_control_ch1 = ((0 << CH1_SEL_OFFSET) &
                    ↪ CH1_SEL_MASK) | (*p_control_ch1 & ~
                    ↪ CH1_SEL_MASK);
929                 delay();
930             break;
931         }
932         case 21:
933         case 22:
934         {
935                 int* p_control_ch2= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_6);
936                 *p_control_ch2 = ((0 << CH2_SEL_OFFSET) &
                    ↪ CH2_SEL_MASK) | (*p_control_ch2 & ~
                    ↪ CH2_SEL_MASK);
937                 delay();
938             break;
939         }
940     }
941     return;
942 }
943
944 //VMM MODE
945 void DUT_setVMM(){
946
947      int* p_control= (int*)(DUT_OFFSET+
           ↪ CONFIG_MEMRISTORCTRL_DIG_3);
948      *p_control |= OPERATION_MODE_MASK;
949      delay();
950     return;
951 }
952
953 void DUT_resetVMM(){
954
955      int* p_control= (int*)(DUT_OFFSET+
           ↪ CONFIG_MEMRISTORCTRL_DIG_3);
956      *p_control &= ~ OPERATION_MODE_MASK;
957      delay();
958     return;
959 }
960 /// Enable and Disable read///
961 void DUT_setReadEnCh(int Ch){
962     switch(Ch)
```

```
963     {
964         case 11:
965         case 12:
966         {
967                 int* p_control= (int*)(DUT_OFFSET+
                        ↪ CONFIG_MEMRISTORCTRL_DIG_3);
968                 *p_control |= READ_EN_CH1_MASK; //
                        ↪ DUT_setReadEnCh1
969                 delay();
970             break;
971         }
972         case 21:
973         case 22:
974         {
975                  int* p_control= (int*)(DUT_OFFSET+
                        ↪ CONFIG_MEMRISTORCTRL_DIG_3);
976                 *p_control |= READ_EN_CH2_MASK;//
                        ↪ DUT_setReadEnCh2
977                 delay();
978             break;
979         }
980     }
981     return;
982 }
983
984 void DUT_resetReadEnCh(int Ch){
985     switch(Ch)
986     {
987         case 11:
988         case 12:
989         {
990                 int* p_control= (int*)(DUT_OFFSET+
                        ↪ CONFIG_MEMRISTORCTRL_DIG_3);
991                 *p_control &= ~READ_EN_CH1_MASK; //
                        ↪ DUT_clearReadEnCh1
992                 delay();
993             break;
994         }
995         case 21:
996         case 22:
997         {
998                  int* p_control= (int*)(DUT_OFFSET+
                        ↪ CONFIG_MEMRISTORCTRL_DIG_3);
```

```
999                 *p_control &= ~READ_EN_CH2_MASK; //
                       ↪ DUT_clearReadEnCh2
1000                delay();
1001            break;
1002        }
1003    }
1004    return;
1005}
1006/// Enable and disable write///
1007void DUT_setWriteEnCh(int Ch){
1008    switch(Ch)
1009    {
1010        case 11:
1011        {
1012                int* p_control= (int*)(DUT_OFFSET+
                       ↪ CONFIG_MEMRISTORCTRL_DIG_5);
1013                *p_control |= WRITE_CH1_1_MASK;
1014                delay();
1015            break;
1016        }
1017        case 12:
1018        {
1019               int* p_control= (int*)(DUT_OFFSET+
                          ↪ CONFIG_MEMRISTORCTRL_DIG_5);
1020               *p_control |= WRITE_CH1_2_MASK;
1021               delay();
1022            break;
1023        }
1024        case 21:
1025        {
1026                int* p_control= (int*)(DUT_OFFSET+
                       ↪ CONFIG_MEMRISTORCTRL_DIG_5);
1027                *p_control |= WRITE_CH2_1_MASK;
1028                delay();
1029            break;
1030        }
1031        case 22:
1032        {
1033                int* p_control= (int*)(DUT_OFFSET+
                       ↪ CONFIG_MEMRISTORCTRL_DIG_5);
1034                *p_control |= WRITE_CH2_2_MASK;
1035                delay();
1036            break;
```

```
1037          }
1038      }
1039      return;
1040  }
1041
1042  void DUT_resetWriteEnCh(){
1043
1044    int* p_control= (int*)(DUT_OFFSET+
          ↪ CONFIG_MEMRISTORCTRL_DIG_5);
1045    *p_control &= ~ WRITE_ALL_CH_MASK;
1046    delay();
1047      return;
1048  }
1049
1050  //CONFIG_MEMRISTORCTRL_DIG_10
1051  void DUT_setAutoDefaultValSet(){
1052      int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_10);
1053      *p_control |= AUTO_DEFAULT_VAL_SET_MASK;
1054      delay();
1055      return;
1056  }
1057
1058  void DUT_clearAutoDefaultValSet(){
1059      int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_10);
1060      *p_control &= ~AUTO_DEFAULT_VAL_SET_MASK;
1061      delay();
1062      return;
1063  }
1064
1065
1066  ///Current Mode///
1067
1068  void DUT_setCurrentMode(){
1069      int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_3);
1070      *p_control |= CURRENT_MODE_MASK;
1071      delay();
1072      return;
1073  }
1074
1075  void DUT_resetCurrentMode(){
```

```
1076      int* p_control= (int*)(DUT_OFFSET+
              ↪ CONFIG_MEMRISTORCTRL_DIG_3);
1077      *p_control &= ~CURRENT_MODE_MASK;
1078      delay();
1079      return;
1080  }
1081
1082  void DUT_setChCurrentDefault(int Ch){
1083      switch(Ch)
1084      {
1085          case 11:
1086          case 12:
1087          {
1088              int* p_control= (int*)(DUT_OFFSET+
                      ↪ CONFIG_MEMRISTORCTRL_DIG_3);
1089              *p_control |= CH1_CURRENT_DEFAULT_MASK;
1090              delay();
1091              break;
1092          }
1093          case 21:
1094          case 22:
1095          {
1096              int* p_control= (int*)(DUT_OFFSET+
                      ↪ CONFIG_MEMRISTORCTRL_DIG_4);
1097              *p_control |= CH2_CURRENT_DEFAULT_MASK;
1098              delay();
1099              break;
1100          }
1101      }
1102      return;
1103  }
1104
1105  void DUT_resetChCurrentDefault(int Ch){
1106      switch(Ch)
1107      {
1108          case 11:
1109          case 12:
1110          {
1111              int* p_control= (int*)(DUT_OFFSET+
                      ↪ CONFIG_MEMRISTORCTRL_DIG_3);
1112              *p_control &= ~ CH1_CURRENT_DEFAULT_MASK;
1113              delay();
1114              break;
```

```
1115            }
1116            case 21:
1117            case 22:
1118            {
1119                int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_4);
1120                *p_control &= ~ CH2_CURRENT_DEFAULT_MASK;
1121                delay();
1122                break;
1123            }
1124        }
1125        return;
1126    }
1127
1128    void DUT_setChCurrentPulse(int Ch){
1129        switch(Ch)
1130        {
1131            case 11:
1132            case 12:
1133            {
1134                int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_3);
1135                *p_control |= CH1_CURRENT_PULSE_MASK;
1136                delay();
1137                break;
1138            }
1139            case 21:
1140            case 22:
1141            {
1142                int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_DIG_4);
1143                *p_control |= CH2_CURRENT_PULSE_MASK;
1144                delay();
1145                break;
1146            }
1147        }
1148        return;
1149    }
1150
1151    void DUT_setall_ChCurrentPulse(){
1152
1153        int* p_control_ch1= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_3);
```

```
1154      *p_control_ch1 |= CH1_CURRENT_PULSE_MASK;
1155      delay();
1156
1157      int* p_control_ch2= (int*)(DUT_OFFSET+
              ↪ CONFIG_MEMRISTORCTRL_DIG_4);
1158      *p_control_ch2 |= CH2_CURRENT_PULSE_MASK;
1159      delay();
1160
1161      return;
1162  }
1163
1164  void DUT_resetChCurrentPulse(int Ch){
1165      switch(Ch)
1166      {
1167          case 11:
1168          case 12:
1169          {
1170              int* p_control= (int*)(DUT_OFFSET+
                      ↪ CONFIG_MEMRISTORCTRL_DIG_3);
1171              *p_control &= ~ CH1_CURRENT_PULSE_MASK;
1172              delay();
1173              break;
1174          }
1175          case 21:
1176          case 22:
1177          {
1178              int* p_control= (int*)(DUT_OFFSET+
                      ↪ CONFIG_MEMRISTORCTRL_DIG_4);
1179              *p_control &= ~ CH2_CURRENT_PULSE_MASK;
1180              delay();
1181              break;
1182          }
1183      }
1184      return;
1185  }
1186
1187  void DUT_resetall_ChCurrentPulse(){
1188
1189      int* p_control_ch1= (int*)(DUT_OFFSET+
              ↪ CONFIG_MEMRISTORCTRL_DIG_3);
1190      *p_control_ch1 &= ~ CH1_CURRENT_PULSE_MASK;
1191      delay();
1192
```

```
1193     int* p_control_ch2= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_4);
1194     *p_control_ch2 &= ~ CH2_CURRENT_PULSE_MASK;
1195     delay();
1196
1197
1198     return;
1199 }
1200
1201 void DUT_set_CurrentPulse_var_Imode(){
1202
1203     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_7);
1204     *p_control |= VAR_CURRENT_PULSE_MASK;
1205     delay();
1206     return;
1207 }
1208
1209 void DUT_reset_CurrentPulse_var_Imode(){
1210
1211     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_7);
1212     *p_control &= ~ VAR_CURRENT_PULSE_MASK;
1213     delay();
1214     return;
1215 }
1216
1217 //CONFIG_MEMRISTORCTRL_DIG_1
1218 void DUT_setClkDivider(int ClkDivider){
1219     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_DIG_1);
1220     *p_control = ClkDivider & CLK_DIVIDER_MASK;
1221     delay();
1222     return;
1223 }
1224
1225
1226 bool should_remove(int value, int *values_to_remove, int
    ↪ num_values_to_remove) {
1227     for (int i = 0; i < num_values_to_remove; i++) {
1228         if (value == values_to_remove[i]) {
1229             return true;
1230         }
```

```
1231     }
1232     return false;
1233 }
1234 
1235 void remove_and_loop(int *array, int *size, int *
       ↪ values_to_remove, int num_values_to_remove) {
1236     int new_size = *size;
1237     int j = 0;
1238     for (int i = 0; i < *size; i++) {
1239         if (!should_remove(array[i], values_to_remove,
              ↪ num_values_to_remove)) {
1240             array[j++] = array[i];
1241         } else {
1242             new_size--;
1243         }
1244     }
1245     *size = new_size;
1246     // Loop through the remaining elements
1247     for (int i = 0; i < *size; i++) {
1248       DUT_resetChDefault1(array[i]);
1249       DUT_setChDefault2(array[i]);
1250     }
1251 }
1252 
1253 
1254 void config_init_default_single_ch(volatile memristor *m)
1255 {
1256     int array[] = {11, 12, 21, 22, 31};
1257     int size = sizeof(array) / sizeof(array[0]);
1258     int values_to_remove[] = {m->mem_pos};
1259     int num_values_to_remove = sizeof(values_to_remove) /
          ↪ sizeof(values_to_remove[0]);
1260 
1261     DUT_resetChDefault1(m->mem_pos);
1262     DUT_resetChDefault2(m->mem_pos);
1263 
1264     remove_and_loop(array, &size, values_to_remove,
          ↪ num_values_to_remove);
1265 
1266     return;
1267 }
1268 
1269 void config_init_default_multiple_ch(volatile memristor *
```

```
    ↪ m_1st, volatile memristor *m_2nd)
1270 {
1271     int array[] = {11, 12, 21, 22};
1272     int size = sizeof(array) / sizeof(array[0]);
1273     int values_to_remove[] = {m_1st->mem_pos,m_2nd->
           ↪ mem_pos};
1274     int num_values_to_remove = sizeof(values_to_remove) /
           ↪ sizeof(values_to_remove[0]);
1275
1276     DUT_resetChDefault1(m_1st->mem_pos);
1277     DUT_resetChDefault2(m_1st->mem_pos);
1278     DUT_resetChDefault1(m_2nd->mem_pos);
1279     DUT_resetChDefault2(m_2nd->mem_pos);
1280
1281     remove_and_loop(array, &size, values_to_remove,
           ↪ num_values_to_remove);
1282   return;
1283 }
1284
1285
1286 void pwm_generator(volatile memristor *m) {
1287     DUT_setChPulsePara(m->mem_pos, m->pulse_w, m->pulse_T,
           ↪  m->pulse_rep);
1288     return;
1289 }
1290
1291 // subfunctions
1292 void sub_pulse(volatile memristor *m) {
1293     DUT_setCh_subPulsePara(m->mem_pos, m->subpulse_delay,
           ↪ m->subpulse_w);
1294     return;
1295 }
1296
1297 void read_enable(volatile memristor *m) {
1298     DUT_setReadEnCh(m->mem_pos);
1299 }
1300
1301 void write_start(volatile memristor *m) {
1302     DUT_setChPulseStartEn(m->mem_pos);
1303     DUT_resetChPulseStartEn(m->mem_pos);
1304     return;
1305 }
1306
```

```
1307
1308 void write_digital_reset(volatile memristor *m) {
1309     pwm_generator(m);
1310     DUT_resetCh_Pulse(m->mem_pos);
1311     return;
1312 }
1313
1314 void read_digital(volatile memristor *m) {
1315     pwm_generator(m);
1316     DUT_resetCh_Pulse(m->mem_pos);
1317     sub_pulse(m);
1318     DUT_setChPulse2En(m->mem_pos);
1319     return;
1320 }
1321
1322 void read_digital_var_mode(volatile memristor *m) {
1323     pwm_generator(m);
1324     sub_pulse(m);
1325     DUT_setChPulse2En(m->mem_pos);
1326     return;
1327 }
1328
1329 void write_digital_set(volatile memristor *m) {
1330     pwm_generator(m);
1331     DUT_setCh_Pulse(m->mem_pos);
1332     return;
1333 }
1334
1335 void write_digital_reset_var_mode(volatile memristor *m) {
1336     DUT_resetCh_Pulse(m->mem_pos);
1337     return;
1338 }
1339
1340 void write_digital_set_var_mode(volatile memristor *m) {
1341     DUT_setCh_Pulse(m->mem_pos);
1342     return;
1343 }
1344
1345 void write_restore_default(volatile memristor *m) {
1346     m-> on = false;
1347     DUT_default_Ch_Pulse(m->mem_pos);
1348     return;
1349 }
```

```
1350 ////Initialization function////
1351 void memristor_init(volatile memristor *m, int mem_pos,
        ↪ bool ext_DAC, bool int_DAC, volatile bool on) {
1352     m->mem_pos = mem_pos;
1353     m->pulse_w = 3125;
1354     m->pulse_T = 6250;
1355     m->pulse_rep = 3;
1356     m->subpulse_w = 2000;
1357     m->subpulse_delay = 300;
1358     m->ext_DAC = ext_DAC;
1359     m->int_DAC = int_DAC;
1360     m->on = on; //selected memristor for read and write
1361     return;
1362 }
1363
1364 //simple error correction: change of pulse width
1365 void memristor_cor(volatile memristor *m, int pulse_w, int
        ↪  pulse_T, int pulse_rep) {
1366     m->pulse_w = pulse_w;
1367     m->pulse_T = pulse_T;
1368     m->pulse_rep = pulse_rep;
1369     pwm_generator(m);
1370     return;
1371 }
1372
1373 //I mode: change of pulse width
1374 void memristor_pulse_I_Var_mode(volatile memristor *m) {
1375     m->pulse_w = 3125;
1376     m->pulse_T = 6250;
1377     m->pulse_rep = 3;
1378     pwm_generator(m);
1379 }
1380
1381
1382 ////V test start////
1383
1384 void digital_voltage_write_set_single_memristor(volatile
        ↪ memristor *m)
1385 {
1386
1387       m-> on = true;
1388       config_init_default_single_ch(m);
1389       write_digital_set(m);
```

```
1390
1391        DUT_setChPulseStartEn(m->mem_pos);
1392        DUT_resetChPulseStartEn(m->mem_pos);
1393
1394        delay_wait_for_pwm();
1395        write_restore_default(m);
1396
1397        //delay_wait_for_pwm();
1398 }
1399
1400 void digital_voltage_write_reset_single_memristor(volatile
       ↪  memristor *m)
1401 {
1402
1403        m-> on = true;
1404        config_init_default_single_ch(m);
1405        write_digital_reset(m);
1406
1407        DUT_setChPulseStartEn(m->mem_pos);
1408        DUT_resetChPulseStartEn(m->mem_pos);
1409
1410        delay_wait_for_pwm();
1411        write_restore_default(m);
1412
1413
1414        //delay_wait_for_pwm();
1415 }
1416
1417 void digital_read_single_memristor(volatile memristor *m)
1418 {
1419
1420        m-> on = true;
1421        config_init_default_single_ch(m);
1422        read_digital(m);
1423
1424        DUT_setChPulseStartEn(m->mem_pos);
1425        DUT_resetChPulseStartEn(m->mem_pos);
1426
1427        delay_wait_for_pwm();
1428        write_restore_default(m);
1429        DUT_resetChPulse2En(m->mem_pos);
1430
1431        //delay_wait_for_pwm();
```

```
1432 }
1433
1434 void digital_voltage_write_set_multiple_memristor(volatile
     ↪  memristor *m_1st, volatile memristor *m_2nd)
1435 {
1436
1437         m_1st->on = true;
1438         m_2nd->on = true;
1439         config_init_default_multiple_ch(m_1st,m_2nd);
1440         write_digital_set(m_1st);
1441         write_digital_set(m_2nd);
1442
1443         DUT_set_allChPulseStartEn();
1444         DUT_reset_allChPulseStartEn();
1445
1446         delay_wait_for_pwm();
1447         write_restore_default(m_1st);
1448         write_restore_default(m_2nd);
1449
1450         //delay_wait_for_pwm();
1451
1452
1453
1454 }
1455
1456 void digital_voltage_write_reset_multiple_memristor(
     ↪ volatile memristor *m_1st, volatile memristor *m_2nd)
1457 {
1458
1459         m_1st->on = true;
1460         m_2nd->on = true;
1461         config_init_default_multiple_ch(m_1st,m_2nd);
1462         write_digital_reset(m_1st);
1463         write_digital_reset(m_2nd);
1464
1465         DUT_set_allChPulseStartEn();
1466         DUT_reset_allChPulseStartEn();
1467
1468         delay_wait_for_pwm();
1469         write_restore_default(m_1st);
1470         write_restore_default(m_2nd);
1471
1472         //delay_wait_for_pwm();
```

```
1473
1474 }
1475
1476 void digital_read_multiple_memristor(volatile memristor *
     ↪ m_1st, volatile memristor *m_2nd)
1477 {
1478
1479           m_1st->on = true;
1480           m_2nd->on = true;
1481           config_init_default_multiple_ch(m_1st,m_2nd);
1482           DUT_resetCh_Pulse(m_1st->mem_pos);
1483           DUT_resetCh_Pulse(m_2nd->mem_pos);
1484           sub_pulse(m_1st);
1485           sub_pulse(m_2nd);
1486
1487           DUT_set_allChPulse2En();
1488
1489           DUT_set_allChPulseStartEn();
1490           DUT_reset_allChPulseStartEn();
1491
1492           delay_wait_for_pwm();
1493           write_restore_default(m_1st);
1494           write_restore_default(m_2nd);
1495           DUT_reset_allChPulse2En();
1496           //delay_wait_for_pwm();
1497 }
1498 ////V test end////
1499
1500
1501
1502 ////current test start/////
1503
1504 void digital_current_write_single_memristor(volatile
     ↪ memristor *m)
1505 {
1506       DUT_setCurrentMode();
1507
1508       m-> on = true;
1509       config_init_default_single_ch(m);
1510       memristor_pulse_I_Var_mode(m);
1511       DUT_setChCurrentPulse(m->mem_pos);
1512
1513       DUT_setChPulseStartEn(m->mem_pos);
```

```
1514        DUT_resetChPulseStartEn(m->mem_pos);
1515
1516        delay_wait_for_pwm();
1517        write_restore_default(m);
1518        DUT_resetChCurrentPulse(m->mem_pos);
1519        //delay_wait_for_pwm();
1520
1521      DUT_resetCurrentMode();
1522 }
1523
1524 void digital_current_write_multiple_memristor(volatile
      ↪ memristor *m_1st, volatile memristor *m_2nd)
1525 {
1526      DUT_setCurrentMode();
1527
1528       m_1st->on = true;
1529       m_2nd->on = true;
1530
1531       config_init_default_multiple_ch(m_1st,m_2nd);
1532       memristor_pulse_I_Var_mode(m_1st);
1533       memristor_pulse_I_Var_mode(m_2nd);
1534       DUT_setall_ChCurrentPulse();
1535
1536       DUT_set_allChPulseStartEn();
1537       DUT_reset_allChPulseStartEn();
1538
1539       delay_wait_for_pwm();
1540       write_restore_default(m_1st);
1541       write_restore_default(m_2nd);
1542       DUT_resetall_ChCurrentPulse();
1543
1544       //delay_wait_for_pwm();
1545
1546      DUT_resetCurrentMode();
1547
1548 }
1549 ////current test end////
1550
1551
1552
1553 ////var test start////
1554
1555 void digital_variable_V_write_set(volatile memristor *m){
```

```
1556
1557        m-> on = true;
1558        config_init_default_single_ch(m);
1559        memristor_pulse_I_Var_mode(m);
1560        DUT_setCh_Pulse_var_mode_pwm1();
1561
1562        DUT_setChPulseStartEn(11);
1563        DUT_resetChPulseStartEn(11);
1564
1565        delay_wait_for_pwm();
1566        write_restore_default(m);
1567        DUT_default_Ch_Pulse_var_mode();
1568        //delay_wait_for_pwm();
1569 }
1570
1571 void digital_variable_V_write_reset(volatile memristor *m)
       ↪ {
1572
1573        //reset
1574        m-> on = true;
1575        config_init_default_single_ch(m);
1576        memristor_pulse_I_Var_mode(m);
1577        DUT_resetCh_Pulse_var_mode_pwm1();
1578
1579        DUT_setChPulseStartEn(11);
1580        DUT_resetChPulseStartEn(11);
1581
1582        delay_wait_for_pwm();
1583        write_restore_default(m);
1584        DUT_default_Ch_Pulse_var_mode();
1585        //delay_wait_for_pwm();
1586
1587 }
1588
1589
1590 void digital_variable_read(volatile memristor *m){
1591        //reset
1592        m-> on = true;
1593        config_init_default_single_ch(m);
1594        DUT_resetCh_Pulse_var_mode_pwm1();
1595        read_digital_var_mode(m);
1596
1597        DUT_setChPulseStartEn(11);
```

```
1598        DUT_resetChPulseStartEn(11);
1599
1600        delay_wait_for_pwm();
1601        write_restore_default(m);
1602        DUT_default_Ch_Pulse_var_mode();
1603        //delay_wait_for_pwm();
1604
1605  }
1606
1607
1608   void digital_variable_I_write_set(volatile memristor *m){
1609
1610        DUT_setCurrentMode();
1611        m-> on = true;
1612        config_init_default_single_ch(m);
1613        DUT_resetChDefault2_var_mode();
1614
1615        DUT_set_CurrentPulse_var_Imode();
1616
1617        DUT_setChPulseStartEn(11);
1618        DUT_resetChPulseStartEn(11);
1619
1620        delay_wait_for_pwm();
1621        write_restore_default(m);
1622        DUT_reset_CurrentPulse_var_Imode();
1623        DUT_default_Ch_Pulse_var_Imode();
1624        //delay_wait_for_pwm();
1625        DUT_resetCurrentMode();
1626   }
1627
1628  ////var test end////
1629
1630
1631  ////VMM test start/////
1632   void digital_VMM(){
1633      DUT_resetChDefault1(11);
1634      DUT_resetChDefault2(11);
1635      DUT_resetChDefault1(12);
1636      DUT_resetChDefault2(12);
1637      DUT_resetChDefault1(21);
1638      DUT_resetChDefault2(21);
1639      DUT_resetChDefault1(22);
1640      DUT_resetChDefault2(22);
```

```
1641      DUT_resetChDefault1(31);
1642      DUT_resetChDefault2(31);
1643      DUT_setVMM();
1644      DUT_set_allChPulseStartEn();
1645      DUT_reset_allChPulseStartEn();
1646      delay_wait_for_pwm();
1647      DUT_resetVMM();
1648  }
1649 ////VMM test end////
1650 ////END DIGITAL PART////
1651
1652 ////START ANALOG PART////
1653
1654 //CONFIG_MEMRISTORCTRL_ANALOG
1655 #define CONFIG_MEMRISTORCTRL_ANALOG_1   0   *4
1656 #define CONFIG_MEMRISTORCTRL_ANALOG_2   1   *4
1657 #define CONFIG_MEMRISTORCTRL_ANALOG_3   2   *4
1658 #define CONFIG_MEMRISTORCTRL_ANALOG_4   3   *4
1659 #define CONFIG_MEMRISTORCTRL_ANALOG_5   4   *4
1660 #define CONFIG_MEMRISTORCTRL_ANALOG_6   5   *4
1661 #define CONFIG_MEMRISTORCTRL_ANALOG_7   6   *4
1662 #define CONFIG_MEMRISTORCTRL_ANALOG_8   7   *4
1663 #define CONFIG_MEMRISTORCTRL_ANALOG_9   8   *4
1664 #define CONFIG_MEMRISTORCTRL_ANALOG_10  9   *4
1665 #define CONFIG_MEMRISTORCTRL_ANALOG_11  10  *4
1666 #define CONFIG_MEMRISTORCTRL_ANALOG_12  11  *4
1667 #define CONFIG_MEMRISTORCTRL_ANALOG_13  12  *4
1668 #define CONFIG_MEMRISTORCTRL_ANALOG_14  13  *4
1669 #define CONFIG_MEMRISTORCTRL_ANALOG_15  14  *4
1670 #define CONFIG_MEMRISTORCTRL_ANALOG_16  15  *4
1671
1672 //CONFIG_MEMRISTORCTRL_ANALOG_1/2/3/4/5
1673 //opamp selection for v_ctrl, i_ctrl and var_i_ctrl
1674     // EXTERNAL SOURCE on : row 0 and row 1
1675 #define EXT_OPA_ON_DIFF_M                   50
1676 #define EXT_OPA_OFF                         13
1677
1678 //CONFIG_MEMRISTORCTRL_ANALOG_6
1679 //opamp selection for var_v_ctrl mode
1680 //#define VAR_V_OPA_ON_DIFF_M   50
1681 #define VAR_V_OPA_OFF_VDD                   13
1682 #define VAR_V_OPA_ON                        178
1683 #define VAR_V_OPA_ON_RD                     114
```

```
1684 #define VAR_I_OPA_ON                          141
1685
1686 //CONFIG_MEMRISTORCTRL_ANALOG_7
1687 //WIDTH SEL FOR CC
1688 #define VAR_V_MODE                            14
1689 #define VAR_I_MODE                            6
1690 #define W_DEFAULT                             7
1691
1692 //CONFIG_MEMRISTORCTRL_ANALOG_8
1693 #define CC_VAR_V_I_MODE_DEFAULT               96
1694 #define CC_VAR_V_MODE_RD_HRS                  96
1695 #define CC_VAR_V_MODE_LRS_D1                  96
1696 #define CC_VAR_V_MODE_LRS_D2                  80
1697 #define CC_VAR_V_MODE_LRS_D3                  48
1698 #define CC_VAR_I_MODE_R1                      48
1699 #define CC_VAR_I_MODE_R2                      52
1700 #define CC_VAR_I_MODE_R3                      50
1701 #define CC_VAR_I_MODE_R4                      49
1702
1703
1704 //CONFIG_MEMRISTORCTRL_ANALOG_9
1705 #define VAR_MEM5_STATE_OFF                    12
1706 #define VAR_MEM5_STATE_HRS                    33
1707 #define VAR_MEM5_STATE_LRS                    29
1708 #define VAR_MEM5_STATE_RD                     38
1709 #define VAR_MEM5_STATE_I                      102
1710
1711
1712 //CONFIG_MEMRISTORCTRL_ANALOG_10
1713 #define MUX_CTRL_HRS                          15
1714 #define MUX_CTRL_LRS_C1                       13
1715 #define MUX_CTRL_LRS_C2                       7
1716 #define MUX_CTRL_LRS_C1_C2                    5
1717 #define MUX_CTRL_RD_HRS                       15
1718 #define MUX_CTRL_I_MODE_C1                    13
1719 #define MUX_CTRL_I_MODE_C2                    7
1720 #define MUX_CTRL_VMM                          15
1721 #define MUX_CTRL_VAR                          15
1722
1723
1724 //CONFIG_MEMRISTORCTRL_ANALOG_11/12/13/14
1725 #define OPM_MEM_HRS                           33
1726 #define OPM_MEM_LRS                           29
```

```
1727 #define OPM_MEM_RD                              38
1728 #define OPM_MEM1_2_I_MODE                      102
1729 #define OPM_MEM3_4_I_MODE                      157
1730 #define OPM_MEM_V_shorted_GND                  12
1731 #define OPM_MEM_DEFAULT                        0
1732
1733
1734 //CONFIG_MEMRISTORCTRL_ANALOG_15
1735 #define EXT_DAC_V_REF_R1                       193
1736 #define EXT_DAC_V_REF_R2                       200
1737 #define EXT_DAC_V_REF_R1_R2                    201
1738 #define EXT_DAC_I_REF_R1                       194
1739 #define EXT_DAC_I_REF_R2                       208
1740 #define EXT_DAC_I_REF_R1_R2                    210
1741 #define EXT_DAC_VAR_V_REF                      196
1742 #define EXT_DAC_VAR_I_REF                      224
1743
1744
1745 //CONFIG_MEMRISTORCTRL_ANALOG_16
1746 #define DUMMY                                  0
1747
1748
1749 //CONFIG_MEMRISTORCTRL_ANALOG_1
1750 void OPAMP_SET_R1_V()
1751 {
1752     int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_1);
1753     *p_control = EXT_OPA_ON_DIFF_M;
1754     delay();
1755
1756 }
1757
1758 void OPAMP_DEFAULT_R1_V()
1759 {
1760     int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_1);
1761     *p_control = EXT_OPA_OFF;
1762     delay();
1763
1764 }
1765
1766 //CONFIG_MEMRISTORCTRL_ANALOG_2
1767 void OPAMP_SET_R2_V()
```

```
1768 {
1769     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_2);
1770     *p_control = EXT_OPA_ON_DIFF_M;
1771     delay();
1772 
1773 }
1774 
1775 void OPAMP_DEFAULT_R2_V()
1776 {
1777     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_2);
1778     *p_control = EXT_OPA_OFF;
1779     delay();
1780 
1781 }
1782 
1783 
1784 //CONFIG_MEMRISTORCTRL_ANALOG_3
1785 void OPAMP_SET_R1_I()
1786 {
1787     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_3);
1788     *p_control = EXT_OPA_ON_DIFF_M;
1789     delay();
1790 
1791 }
1792 
1793 
1794 void OPAMP_DEFAULT_R1_I()
1795 {
1796     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_3);
1797     *p_control = EXT_OPA_OFF;
1798     delay();
1799 
1800 }
1801 
1802 
1803 //CONFIG_MEMRISTORCTRL_ANALOG_4
1804 void OPAMP_SET_R2_I()
1805 {
1806     int* p_control= (int*)(DUT_OFFSET+
```

```
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_4);
1807    *p_control = EXT_OPA_ON_DIFF_M;
1808    delay();
1809
1810}
1811
1812void OPAMP_DEFAULT_R2_I()
1813{
1814    int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_4);
1815    *p_control = EXT_OPA_OFF;
1816    delay();
1817
1818}
1819
1820
1821//CONFIG_MEMRISTORCTRL_ANALOG_5
1822void OPAMP_SET_Var_I()
1823{
1824    int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_5);
1825    *p_control = EXT_OPA_ON_DIFF_M;
1826    delay();
1827
1828}
1829
1830void OPAMP_DEFAULT_Var_I()
1831{
1832    int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_5);
1833    *p_control = EXT_OPA_OFF;
1834    delay();
1835
1836}
1837
1838void voltage_SEL_Opamp(int Ch){
1839    switch(Ch)
1840    {
1841        case 11:
1842        case 12:
1843        {
1844                OPAMP_SET_R1_V();
1845            break;
```

```
1846            }
1847            case 21:
1848            case 22:
1849            {
1850                    OPAMP_SET_R2_V();
1851                break;
1852            }
1853        }
1854        return;
1855    }
1856
1857    void current_SEL_Opamp(int Ch){
1858        switch(Ch)
1859        {
1860            case 11:
1861            case 12:
1862            {
1863                   OPAMP_SET_R1_I();
1864                break;
1865            }
1866            case 21:
1867            case 22:
1868            {
1869                    OPAMP_SET_R2_I();
1870                break;
1871            }
1872        }
1873        return;
1874    }
1875
1876    //CONFIG_MEMRISTORCTRL_ANALOG_6
1877    void OPAMP_READ_HRS_Var_V()
1878    {
1879        int* p_control= (int*)(DUT_OFFSET+
              ↪ CONFIG_MEMRISTORCTRL_ANALOG_6);
1880        *p_control = VAR_V_OPA_ON_RD;
1881        delay();
1882
1883    }
1884    void OPAMP_LRS_Var_V()
1885    {
1886        int* p_control= (int*)(DUT_OFFSET+
              ↪ CONFIG_MEMRISTORCTRL_ANALOG_6);
```

```
1887     *p_control = VAR_V_OPA_ON;
1888     delay();
1889
1890 }
1891 void OPAMP_Var_I()
1892 {
1893     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_6);
1894     *p_control = VAR_I_OPA_ON;
1895     delay();
1896
1897 }
1898 void OPAMP_DEFAULT_Var_V()
1899 {
1900     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_6);
1901     *p_control = VAR_V_OPA_OFF_VDD;
1902     delay();
1903
1904 }
1905
1906
1907 //CONFIG_MEMRISTORCTRL_ANALOG_7
1908 void OPAMP_Var_V_CC()
1909 {
1910     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_7);
1911     *p_control = VAR_V_MODE;
1912     delay();
1913
1914 }
1915
1916 void OPAMP_Var_I_CC()
1917 {
1918     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_7);
1919     *p_control = VAR_I_MODE;
1920     delay();
1921
1922 }
1923 void OPAMP_DEFAULT_Var_CC()
1924 {
1925     int* p_control= (int*)(DUT_OFFSET+
```

```
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_7);
1926    *p_control = W_DEFAULT;
1927    delay();
1928
1929}
1930
1931//CONFIG_MEMRISTORCTRL_ANALOG_8
1932void CC_SEL_VAR_DEFAULT(){
1933    int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_8);
1934    *p_control = CC_VAR_V_I_MODE_DEFAULT;
1935    return;
1936}
1937
1938void CC_SEL_VAR_V_RD_HRS(){
1939    int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_8);
1940    *p_control = CC_VAR_V_MODE_RD_HRS;
1941    return;
1942}
1943
1944void CC_SEL_VAR_V_LRS_DEVICE(int Device){
1945    switch(Device)
1946    {
1947        case 1:
1948        {
1949                int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_ANALOG_8);
1950                *p_control = CC_VAR_V_MODE_LRS_D1;
1951                delay();
1952            break;
1953        }
1954        case 2:
1955        {
1956                int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_ANALOG_8);
1957                *p_control = CC_VAR_V_MODE_LRS_D2;
1958                delay();
1959            break;
1960        }
1961        case 3:
1962        {
1963                 int* p_control= (int*)(DUT_OFFSET+
```

```
                    ↪ CONFIG_MEMRISTORCTRL_ANALOG_8);
1964                *p_control = CC_VAR_V_MODE_LRS_D3;
1965                delay();
1966            break;
1967        }
1968    }
1969    return;
1970 }
1971
1972
1973 void CC_SEL_VAR_I_RES(int Resistor){
1974    switch(Resistor)
1975    {
1976        case 1:
1977        {
1978                int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_ANALOG_8);
1979                *p_control = CC_VAR_I_MODE_R1;
1980                delay();
1981            break;
1982        }
1983        case 2:
1984        {
1985                int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_ANALOG_8);
1986                *p_control = CC_VAR_I_MODE_R2;
1987                delay();
1988            break;
1989        }
1990        case 3:
1991        {
1992                 int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_ANALOG_8);
1993                *p_control = CC_VAR_I_MODE_R3;
1994                delay();
1995            break;
1996        }
1997
1998        case 4:
1999        {
2000                int* p_control= (int*)(DUT_OFFSET+
                    ↪ CONFIG_MEMRISTORCTRL_ANALOG_8);
2001                *p_control = CC_VAR_I_MODE_R4;
```

```
2002                    delay();
2003                break;
2004            }
2005        }
2006        return;
2007    }
2008
2009    //CONFIG_MEMRISTORCTRL_ANALOG_9
2010    void MEM5_STATE_SEL_VAR_OFF(){
2011        int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_ANALOG_9);
2012        *p_control = VAR_MEM5_STATE_OFF;
2013        delay();
2014        return;
2015    }
2016
2017    void MEM5_STATE_SEL_VAR_RD(){
2018        int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_ANALOG_9);
2019        *p_control = VAR_MEM5_STATE_RD;
2020        delay();
2021        return;
2022    }
2023
2024    void MEM5_STATE_SEL_VAR_HRS(){
2025        int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_ANALOG_9);
2026        *p_control = VAR_MEM5_STATE_HRS;
2027        delay();
2028        return;
2029    }
2030
2031    void MEM5_STATE_SEL_VAR_LRS(){
2032        int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_ANALOG_9);
2033        *p_control = VAR_MEM5_STATE_LRS;
2034        delay();
2035        return;
2036    }
2037
2038    void MEM5_STATE_SEL_VAR_I(){
2039        int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_ANALOG_9);
```

```
2040     *p_control = VAR_MEM5_STATE_I;
2041     delay();
2042     return;
2043 }
2044
2045
2046
2047 //CONFIG_MEMRISTORCTRL_ANALOG_10
2048 void V_MUX_CTRL_SEl_RD_HRS(){
2049     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_10);
2050     *p_control = MUX_CTRL_RD_HRS;
2051     delay();
2052     return;
2053 }
2054
2055 void V_MUX_CTRL_SEl_LRS_C1(){
2056     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_10);
2057     *p_control = MUX_CTRL_LRS_C1;
2058     delay();
2059     return;
2060 }
2061
2062 void V_MUX_CTRL_SEl_LRS_C2(){
2063     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_10);
2064     *p_control = MUX_CTRL_LRS_C2;
2065     delay();
2066     return;
2067 }
2068
2069 void V_MUX_CTRL_SEl_M1M3(){
2070     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_10);
2071     *p_control = MUX_CTRL_LRS_C1;
2072     delay();
2073     return;
2074 }
2075
2076 void V_MUX_CTRL_SEl_M1M4_M2M3(){
2077     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_10);
```

```
2078     *p_control = MUX_CTRL_LRS_C1_C2;
2079     delay();
2080     return;
2081 }
2082
2083
2084 void V_MUX_CTRL_SEl_M2M4(){
2085     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_10);
2086     *p_control = MUX_CTRL_LRS_C2;
2087     delay();
2088     return;
2089 }
2090
2091 void V_I_MUX_CTRL_SEL_SET_SINGLE_MEM(int Ch){
2092     switch(Ch)
2093     {
2094         case 11:
2095         case 21:
2096         {
2097                 V_MUX_CTRL_SEl_LRS_C1();
2098             break;
2099         }
2100         case 12:
2101         case 22:
2102         {
2103                 V_MUX_CTRL_SEl_LRS_C2();
2104             break;
2105         }
2106     }
2107     return;
2108 }
2109
2110 void V_MUX_CTRL_SEL_SET_MULT_MEM(int Ch1, int Ch2){
2111     if (Ch1 == 11 && Ch2 == 21) {
2112
2113         V_MUX_CTRL_SEl_M1M3();
2114     }
2115     else if (Ch1 == 12 && Ch2 == 22) {
2116
2117         V_MUX_CTRL_SEl_M2M4();
2118     }
2119     else {
```

```
2120            V_MUX_CTRL_SEl_M1M4_M2M3();
2121        }
2122        return;
2123    }
2124
2125
2126    void I_MUX_CTRL_SEl_M1M3(){
2127        int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_ANALOG_10);
2128        *p_control = MUX_CTRL_LRS_C1_C2;
2129        delay();
2130        return;
2131    }
2132
2133    void I_MUX_CTRL_SEl_M1M4(){
2134        int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_ANALOG_10);
2135        *p_control = MUX_CTRL_LRS_C1_C2;
2136        delay();
2137        return;
2138    }
2139
2140    void I_MUX_CTRL_SEl_M2M3(){
2141        int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_ANALOG_10);
2142        *p_control = MUX_CTRL_LRS_C1_C2;
2143        delay();
2144        return;
2145    }
2146
2147    void I_MUX_CTRL_SEl_M2M4(){
2148        int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_ANALOG_10);
2149        *p_control = MUX_CTRL_LRS_C1_C2;
2150        delay();
2151        return;
2152    }
2153
2154    void I_MUX_CTRL_SEL_SET_MULT_MEM(){
2155        int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_ANALOG_10);
2156        *p_control = MUX_CTRL_LRS_C1_C2;
2157        delay();
```

```
2158     return;
2159 }
2160
2161
2162 void Var_MUX_CTRL_SEl(){
2163     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_10);
2164     *p_control = MUX_CTRL_VAR;
2165     delay();
2166     return;
2167 }
2168
2169 void MUX_CTRL_SEl_VMM(){
2170     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_10);
2171     *p_control = MUX_CTRL_VMM;
2172     delay();
2173     return;
2174 }
2175
2176 //CONFIG_MEMRISTORCTRL_ANALOG_11
2177 void MEM1_STATE_SEL_OFF(){
2178     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_11);
2179     *p_control = OPM_MEM_V_shorted_GND;
2180     delay();
2181     return;
2182 }
2183
2184 void MEM1_STATE_SEL_RD(){
2185     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_11);
2186     *p_control = OPM_MEM_RD;
2187     delay();
2188     return;
2189 }
2190
2191 void MEM1_STATE_SEL_HRS(){
2192     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_11);
2193     *p_control = OPM_MEM_HRS;
2194     delay();
2195     return;
```

```
2196 }
2197
2198 void MEM1_STATE_SEL_LRS(){
2199     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_11);
2200     *p_control = OPM_MEM_LRS;
2201     delay();
2202     return;
2203 }
2204
2205 void MEM1_STATE_SEL_I(){
2206     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_11);
2207     *p_control = OPM_MEM1_2_I_MODE;
2208     delay();
2209     return;
2210 }
2211
2212 //CONFIG_MEMRISTORCTRL_ANALOG_12
2213 void MEM2_STATE_SEL_OFF(){
2214     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_12);
2215     *p_control = OPM_MEM_V_shorted_GND;
2216     delay();
2217     return;
2218 }
2219
2220 void MEM2_STATE_SEL_RD(){
2221     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_12);
2222     *p_control = OPM_MEM_RD;
2223     delay();
2224     return;
2225 }
2226
2227 void MEM2_STATE_SEL_HRS(){
2228     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_12);
2229     *p_control = OPM_MEM_HRS;
2230     delay();
2231     return;
2232 }
2233
```

```
2234 void MEM2_STATE_SEL_LRS(){
2235     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_12);
2236     *p_control = OPM_MEM_LRS;
2237     delay();
2238     return;
2239 }
2240
2241 void MEM2_STATE_SEL_I(){
2242     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_12);
2243     *p_control = OPM_MEM1_2_I_MODE;
2244     delay();
2245     return;
2246 }
2247
2248 //CONFIG_MEMRISTORCTRL_ANALOG_13
2249 void MEM3_STATE_SEL_OFF(){
2250     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_13);
2251     *p_control = OPM_MEM_V_shorted_GND;
2252     delay();
2253     return;
2254 }
2255
2256 void MEM3_STATE_SEL_RD(){
2257     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_13);
2258     *p_control = OPM_MEM_RD;
2259     delay();
2260     return;
2261 }
2262
2263 void MEM3_STATE_SEL_HRS(){
2264     int* p_control= (int*)(DUT_OFFSET+
         ↪ CONFIG_MEMRISTORCTRL_ANALOG_13);
2265     *p_control = OPM_MEM_HRS;
2266     delay();
2267     return;
2268 }
2269
2270 void MEM3_STATE_SEL_LRS(){
2271     int* p_control= (int*)(DUT_OFFSET+
```

```
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_13);
2272    *p_control = OPM_MEM_LRS;
2273    delay();
2274    return;
2275 }
2276
2277 void MEM3_STATE_SEL_I(){
2278    int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_13);
2279    *p_control = OPM_MEM3_4_I_MODE;
2280    delay();
2281    return;
2282 }
2283
2284 //CONFIG_MEMRISTORCTRL_ANALOG_14
2285 void MEM4_STATE_SEL_OFF(){
2286    int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_14);
2287    *p_control = OPM_MEM_V_shorted_GND;
2288    delay();
2289    return;
2290 }
2291
2292 void MEM4_STATE_SEL_RD(){
2293    int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_14);
2294    *p_control = OPM_MEM_RD;
2295    delay();
2296    return;
2297 }
2298
2299 void MEM4_STATE_SEL_HRS(){
2300    int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_14);
2301    *p_control = OPM_MEM_HRS;
2302    delay();
2303    return;
2304 }
2305
2306 void MEM4_STATE_SEL_LRS(){
2307    int* p_control= (int*)(DUT_OFFSET+
        ↪ CONFIG_MEMRISTORCTRL_ANALOG_14);
2308    *p_control = OPM_MEM_LRS;
```

```
2309     delay();
2310     return;
2311 }
2312
2313 void MEM4_STATE_SEL_I(){
2314     int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_14);
2315     *p_control = OPM_MEM3_4_I_MODE;
2316     delay();
2317     return;
2318 }
2319
2320 void MEMS_STATE_SEL_RD(int Ch){
2321     switch(Ch)
2322     {
2323         case 11:
2324         {
2325                 MEM1_STATE_SEL_RD();
2326             break;
2327         }
2328         case 12:
2329         {
2330                 MEM2_STATE_SEL_RD();
2331             break;
2332         }
2333         case 21:
2334         {
2335                  MEM3_STATE_SEL_RD();
2336             break;
2337         }
2338         case 22:
2339         {
2340                  MEM4_STATE_SEL_RD();
2341             break;
2342         }
2343     }
2344     return;
2345 }
2346
2347 void MEMS_STATE_SEL_HRS(int Ch){
2348     switch(Ch)
2349     {
2350         case 11:
```

```
2351            {
2352                    MEM1_STATE_SEL_HRS();
2353                break;
2354            }
2355            case 12:
2356            {
2357                    MEM2_STATE_SEL_HRS();
2358                break;
2359            }
2360            case 21:
2361            {
2362                     MEM3_STATE_SEL_HRS();
2363                break;
2364            }
2365            case 22:
2366            {
2367                     MEM4_STATE_SEL_HRS();
2368                break;
2369            }
2370        }
2371        return;
2372    }
2373
2374    void MEMS_STATE_SEL_I(int Ch){
2375        switch(Ch)
2376        {
2377            case 11:
2378            {
2379                    MEM1_STATE_SEL_I();
2380                break;
2381            }
2382            case 12:
2383            {
2384                    MEM2_STATE_SEL_I();
2385                break;
2386            }
2387            case 21:
2388            {
2389                     MEM3_STATE_SEL_I();
2390                break;
2391            }
2392            case 22:
2393            {
```

```
2394                    MEM4_STATE_SEL_I();
2395                break;
2396            }
2397        }
2398        return;
2399    }
2400
2401    void MEMS_STATE_SEL_LRS(int Ch){
2402        switch(Ch)
2403        {
2404            case 11:
2405            {
2406                    MEM1_STATE_SEL_LRS();
2407                break;
2408            }
2409            case 12:
2410            {
2411                    MEM2_STATE_SEL_LRS();
2412                break;
2413            }
2414            case 21:
2415            {
2416                     MEM3_STATE_SEL_LRS();
2417                break;
2418            }
2419            case 22:
2420            {
2421                     MEM4_STATE_SEL_LRS();
2422                break;
2423            }
2424        }
2425        return;
2426    }
2427
2428    //CONFIG_MEMRISTORCTRL_ANALOG_15
2429    void V_DAC_SEL_R1(){
2430          int* p_control= (int*)(DUT_OFFSET+
                ↪ CONFIG_MEMRISTORCTRL_ANALOG_15);
2431        *p_control = EXT_DAC_V_REF_R1;
2432        delay();
2433        return;
2434    }
2435
```

```
2436 void V_DAC_SEL_R2(){
2437       int* p_control= (int*)(DUT_OFFSET+
           ↪ CONFIG_MEMRISTORCTRL_ANALOG_15);
2438     *p_control = EXT_DAC_V_REF_R2;
2439     delay();
2440     return;
2441 }
2442
2443 void V_DAC_SEL_R1_R2(){
2444       int* p_control= (int*)(DUT_OFFSET+
           ↪ CONFIG_MEMRISTORCTRL_ANALOG_15);
2445     *p_control = EXT_DAC_V_REF_R1_R2;
2446     delay();
2447     return;
2448 }
2449
2450 void V_EXT_DAC_SEL(int Ch){
2451     switch(Ch)
2452     {
2453         case 11:
2454         case 12:
2455         {
2456                 V_DAC_SEL_R1();
2457             break;
2458         }
2459         case 21:
2460         case 22:
2461         {
2462                 V_DAC_SEL_R2();
2463             break;
2464         }
2465     }
2466     return;
2467 }
2468
2469
2470 void I_DAC_SEL_R1(){
2471       int* p_control= (int*)(DUT_OFFSET+
           ↪ CONFIG_MEMRISTORCTRL_ANALOG_15);
2472     *p_control = EXT_DAC_I_REF_R1;
2473     delay();
2474     return;
2475 }
```

```
2476
2477 void I_DAC_SEL_R2(){
2478       int* p_control= (int*)(DUT_OFFSET+
              ↪ CONFIG_MEMRISTORCTRL_ANALOG_15);
2479     *p_control = EXT_DAC_I_REF_R2;
2480     delay();
2481     return;
2482 }
2483
2484 void I_EXT_DAC_SEL(int Ch){
2485     switch(Ch)
2486     {
2487         case 11:
2488         case 12:
2489         {
2490                 I_DAC_SEL_R1();
2491             break;
2492         }
2493         case 21:
2494         case 22:
2495         {
2496                 I_DAC_SEL_R2();
2497             break;
2498         }
2499     }
2500     return;
2501 }
2502
2503 void I_DAC_SEL_R1_R2(){
2504       int* p_control= (int*)(DUT_OFFSET+
              ↪ CONFIG_MEMRISTORCTRL_ANALOG_15);
2505     *p_control = EXT_DAC_I_REF_R1_R2;
2506     delay();
2507     return;
2508 }
2509
2510 void Var_V_DAC_SEL(){
2511       int* p_control= (int*)(DUT_OFFSET+
              ↪ CONFIG_MEMRISTORCTRL_ANALOG_15);
2512     *p_control = EXT_DAC_VAR_V_REF;
2513     delay();
2514     return;
2515 }
```

```
2516
2517 void Var_I_DAC_SEL(){
2518       int* p_control= (int*)(DUT_OFFSET+
            ↪ CONFIG_MEMRISTORCTRL_ANALOG_15);
2519     *p_control = EXT_DAC_VAR_I_REF;
2520     delay();
2521     return;
2522 }
2523
2524
2525
2526 //CONFIG_MEMRISTORCTRL_ANALOG_16
2527 void DUMMY_16(){
2528     int* p_control= (int*)(DUT_OFFSET+
          ↪ CONFIG_MEMRISTORCTRL_ANALOG_16);
2529     *p_control = DUMMY;
2530     delay();
2531     return;
2532 }
2533
2534
2535 void analog_default()
2536 {
2537   OPAMP_DEFAULT_R1_V();
2538   OPAMP_DEFAULT_R2_V();
2539   OPAMP_DEFAULT_R1_I();
2540   OPAMP_DEFAULT_R2_I();
2541   OPAMP_DEFAULT_Var_I();
2542   OPAMP_DEFAULT_Var_V();
2543   OPAMP_DEFAULT_Var_CC();
2544   CC_SEL_VAR_DEFAULT();
2545   MEM5_STATE_SEL_VAR_OFF();
2546   MEM1_STATE_SEL_OFF();
2547   MEM2_STATE_SEL_OFF();
2548   MEM3_STATE_SEL_OFF();
2549   MEM4_STATE_SEL_OFF();
2550   DUMMY_16();
2551 }
2552
2553 //
    ↪ ////////////////////////////////////////////////////////////////////////////
    ↪
2554 ////////////////////////////////////V test start
```

```
    ↪ ////////////////////////////////
2555
2556 void analog_voltage_write_set_single_memristor(volatile
    ↪ memristor *m)
2557 {
2558       analog_default();
2559       voltage_SEL_Opamp(m->mem_pos);
2560       V_I_MUX_CTRL_SEL_SET_SINGLE_MEM(m->mem_pos);
2561       MEMS_STATE_SEL_LRS(m->mem_pos);
2562       V_EXT_DAC_SEL(m->mem_pos);
2563 }
2564
2565 void analog_voltage_write_reset_single_memristor(volatile
    ↪ memristor *m)
2566 {
2567
2568       analog_default();
2569       voltage_SEL_Opamp(m->mem_pos);
2570       V_MUX_CTRL_SEl_RD_HRS();
2571       MEMS_STATE_SEL_HRS(m->mem_pos);
2572       V_EXT_DAC_SEL(m->mem_pos);
2573 }
2574
2575 void analog_read_single_memristor(volatile memristor *m)
2576 {
2577
2578       analog_default();
2579       voltage_SEL_Opamp(m->mem_pos);
2580       V_MUX_CTRL_SEl_RD_HRS();
2581       MEMS_STATE_SEL_RD(m->mem_pos);
2582       V_EXT_DAC_SEL(m->mem_pos);
2583 }
2584
2585 void analog_voltage_write_set_multiple_memristor(volatile
    ↪ memristor *m_1st, volatile memristor *m_2nd)
2586 {
2587
2588       analog_default();
2589       voltage_SEL_Opamp(m_1st->mem_pos);
2590       voltage_SEL_Opamp(m_2nd->mem_pos);
2591       V_MUX_CTRL_SEL_SET_MULT_MEM(m_1st->mem_pos,m_2nd->
          ↪ mem_pos);
2592       MEMS_STATE_SEL_LRS(m_1st->mem_pos);
```

```
2593        MEMS_STATE_SEL_LRS(m_2nd->mem_pos);
2594        V_DAC_SEL_R1_R2();
2595
2596 }
2597
2598 void analog_voltage_write_reset_multiple_memristor(
       ↪ volatile memristor *m_1st, volatile memristor *m_2nd)
2599 {
2600
2601        analog_default();
2602        voltage_SEL_Opamp(m_1st->mem_pos);
2603        voltage_SEL_Opamp(m_2nd->mem_pos);
2604        V_MUX_CTRL_SEl_RD_HRS();
2605        MEMS_STATE_SEL_HRS(m_1st->mem_pos);
2606        MEMS_STATE_SEL_HRS(m_2nd->mem_pos);
2607        V_DAC_SEL_R1_R2();
2608
2609 }
2610
2611 void analog_read_multiple_memristor(volatile memristor *
       ↪ m_1st, volatile memristor *m_2nd)
2612 {
2613
2614        analog_default();
2615        voltage_SEL_Opamp(m_1st->mem_pos);
2616        voltage_SEL_Opamp(m_2nd->mem_pos);
2617        V_MUX_CTRL_SEl_RD_HRS();
2618        MEMS_STATE_SEL_RD(m_1st->mem_pos);
2619        MEMS_STATE_SEL_RD(m_2nd->mem_pos);
2620        V_DAC_SEL_R1_R2();
2621 }
2622 ////V test end/////
2623
2624
2625
2626 ////current test start////
2627
2628 void analog_current_write_single_memristor(volatile
       ↪ memristor *m)
2629 {
2630        analog_default();
2631        current_SEL_Opamp(m->mem_pos);
2632        V_I_MUX_CTRL_SEL_SET_SINGLE_MEM(m->mem_pos);
```

```
2633        MEMS_STATE_SEL_I(m->mem_pos);
2634        I_EXT_DAC_SEL(m->mem_pos);
2635  }
2636
2637  void analog_current_write_multiple_memristor(volatile
        ↪ memristor *m_1st, volatile memristor *m_2nd)
2638  {
2639        analog_default();
2640        current_SEL_Opamp(m_1st->mem_pos);
2641        current_SEL_Opamp(m_2nd->mem_pos);
2642        I_MUX_CTRL_SEL_SET_MULT_MEM();
2643        MEMS_STATE_SEL_I(m_1st->mem_pos);
2644        MEMS_STATE_SEL_I(m_2nd->mem_pos);
2645        I_DAC_SEL_R1_R2();
2646
2647  }
2648  ////current test end/////
2649
2650
2651
2652  ////var test start////
2653
2654  void analog_variable_V_write_set(int Device){
2655
2656        analog_default();
2657        OPAMP_LRS_Var_V();
2658        OPAMP_Var_V_CC();
2659        CC_SEL_VAR_V_LRS_DEVICE(Device);
2660        MEM5_STATE_SEL_VAR_LRS();
2661        Var_MUX_CTRL_SEl();
2662        Var_V_DAC_SEL();
2663
2664  }
2665
2666  void analog_variable_V_write_reset(){
2667
2668        analog_default();
2669        OPAMP_READ_HRS_Var_V();
2670        OPAMP_Var_V_CC();
2671        CC_SEL_VAR_V_RD_HRS();
2672        MEM5_STATE_SEL_VAR_HRS();
2673        Var_MUX_CTRL_SEl();
2674        Var_V_DAC_SEL();
```

```
2675
2676 }
2677
2678 void analog_variable_read(){
2679
2680       analog_default();
2681       OPAMP_READ_HRS_Var_V();
2682       OPAMP_Var_V_CC();
2683       CC_SEL_VAR_V_RD_HRS();
2684       MEM5_STATE_SEL_VAR_RD();
2685       Var_MUX_CTRL_SEl();
2686       Var_V_DAC_SEL();
2687
2688 }
2689
2690  void analog_variable_I_write_set(int Resistor){
2691
2692       analog_default();
2693       OPAMP_Var_I();
2694       OPAMP_Var_I_CC();
2695       CC_SEL_VAR_I_RES(Resistor);
2696       MEM5_STATE_SEL_VAR_I();
2697       Var_MUX_CTRL_SEl();
2698       Var_I_DAC_SEL();
2699
2700  }
2701
2702 /////var test end////
2703
2704 ////VMM test start////
2705  void analog_VMM(){
2706
2707       analog_default();
2708       voltage_SEL_Opamp(11);
2709       voltage_SEL_Opamp(21);
2710       MUX_CTRL_SEl_VMM();
2711       MEMS_STATE_SEL_HRS(11);
2712       MEMS_STATE_SEL_HRS(12);
2713       MEMS_STATE_SEL_HRS(21);
2714       MEMS_STATE_SEL_HRS(22);
2715       V_DAC_SEL_R1_R2();
2716  }
2717 ////VMM test end////
```

```
2718 ////END ANALOG PART////
2719 #define NO_HOSTEDIO
2720 #ifdef NO_HOSTEDIO
2721 #define PRINTF(...)
2722 #else
2723 #define PRINTF(...) { printf(__VA_ARGS__); }
2724 #endif
2725 ////START GO!////
2726 int main() {
2727
2728     DUT_setClkDivider(CLK_DIV);
2729
2730     // 2*2 memristor and m5 for variable mode grid
        ↪ declaration and initialization
2731     volatile memristor *m1;
2732     volatile memristor m1_struct;
2733     m1 = &m1_struct;
2734     volatile memristor *m2;
2735     volatile memristor m2_struct;
2736     m2 = &m2_struct;
2737     volatile memristor *m3;
2738     volatile memristor m3_struct;
2739     m3 = &m3_struct;
2740     volatile memristor *m4;
2741     volatile memristor m4_struct;
2742     m4 = &m4_struct;
2743     volatile memristor *m5;
2744     volatile memristor m5_struct;
2745     m5 = &m5_struct;
2746
2747     //PRINTF("The addr of m1 is 0x%x\n", &m1);
2748     //PRINTF("The data of m1 is 0x%x\n", m1);
2749     //PRINTF("The addr of m2 is 0x%x\n", &m2);
2750     //PRINTF("The data of m2 is 0x%x\n", m2);
2751     //PRINTF("The addr of m3 is 0x%x\n", &m3);
2752     //PRINTF("The data of m3 is 0x%x\n", m3);
2753     //PRINTF("The addr of m4 is 0x%x\n", &m4);
2754     //PRINTF("The data of m4 is 0x%x\n", m4);
2755
2756     //PRINTF("The addr of m1->mem_pos is 0x%x\n", &m1->
        ↪ mem_pos);
2757
2758     // memristor position initialization
```

```
2759     memristor_init(m1, 11, true, false, false);
2760     memristor_init(m2, 12, true, false, false);
2761     memristor_init(m3, 21, true, false, false);
2762     memristor_init(m4, 22, true, false, false);
2763     memristor_init(m5, 31, true, false, false);
2764 
2765     //Enable PWM block
2766     DUT_set_allChPulseGenEn();
2767     DUT_set_allChPulseRepetitionEn();
2768 
2769 ////Programming of memristor array start here////
2770 
2771     analog_read_single_memristor(m1);
2772     digital_read_single_memristor(m1);
2773 
2774     analog_read_single_memristor(m2);
2775     digital_read_single_memristor(m2);
2776 
2777     analog_read_single_memristor(m3);
2778     digital_read_single_memristor(m3);
2779 
2780     analog_read_single_memristor(m4);
2781     digital_read_single_memristor(m4);
2782 
2783     analog_read_multiple_memristor(m1,m4);
2784     digital_read_multiple_memristor(m1,m4);
2785 
2786     analog_read_multiple_memristor(m1,m3);
2787     digital_read_multiple_memristor(m1,m3);
2788 
2789     analog_read_multiple_memristor(m2,m4);
2790     digital_read_multiple_memristor(m2,m4);
2791 
2792     analog_read_multiple_memristor(m2,m3);
2793     digital_read_multiple_memristor(m2,m3);
2794 
2795 
2796     analog_voltage_write_set_single_memristor(m1);
2797     digital_voltage_write_set_single_memristor(m1);
2798 
2799     analog_voltage_write_set_single_memristor(m2);
2800     digital_voltage_write_set_single_memristor(m2);
2801 
```

```
2802      analog_voltage_write_set_single_memristor(m3);
2803      digital_voltage_write_set_single_memristor(m3);
2804
2805      analog_voltage_write_set_single_memristor(m4);
2806      digital_voltage_write_set_single_memristor(m4);
2807
2808
2809      analog_voltage_write_set_multiple_memristor(m1,m4);
2810      digital_voltage_write_set_multiple_memristor(m1,m4);
2811
2812      analog_voltage_write_set_multiple_memristor(m1,m3);
2813      digital_voltage_write_set_multiple_memristor(m1,m3);
2814
2815      analog_voltage_write_set_multiple_memristor(m2,m4);
2816      digital_voltage_write_set_multiple_memristor(m2,m4);
2817
2818      analog_voltage_write_set_multiple_memristor(m2,m3);
2819      digital_voltage_write_set_multiple_memristor(m2,m3);
2820
2821      analog_voltage_write_reset_single_memristor(m1);
2822      digital_voltage_write_reset_single_memristor(m1);
2823
2824      analog_voltage_write_reset_single_memristor(m2);
2825      digital_voltage_write_reset_single_memristor(m2);
2826
2827      analog_voltage_write_reset_single_memristor(m3);
2828      digital_voltage_write_reset_single_memristor(m3);
2829
2830      analog_voltage_write_reset_single_memristor(m4);
2831      digital_voltage_write_reset_single_memristor(m4);
2832
2833      analog_voltage_write_reset_multiple_memristor(m1,m4);
2834      digital_voltage_write_reset_multiple_memristor(m1,m4);
2835
2836      analog_voltage_write_reset_multiple_memristor(m1,m3);
2837      digital_voltage_write_reset_multiple_memristor(m1,m3);
2838
2839      analog_voltage_write_reset_multiple_memristor(m2,m4);
2840      digital_voltage_write_reset_multiple_memristor(m2,m4);
2841
2842      analog_voltage_write_reset_multiple_memristor(m2,m3);
2843      digital_voltage_write_reset_multiple_memristor(m2,m3);
2844
```

```
2845
2846      analog_current_write_single_memristor(m1);
2847      digital_current_write_single_memristor(m1);
2848
2849      analog_current_write_single_memristor(m2);
2850      digital_current_write_single_memristor(m2);
2851
2852      analog_current_write_single_memristor(m3);
2853      digital_current_write_single_memristor(m3);
2854
2855      analog_current_write_single_memristor(m4);
2856      digital_current_write_single_memristor(m4);
2857
2858      analog_current_write_multiple_memristor(m1,m4);
2859      digital_current_write_multiple_memristor(m1,m4);
2860
2861      analog_current_write_multiple_memristor(m1,m3);
2862      digital_current_write_multiple_memristor(m1,m3);
2863
2864      analog_current_write_multiple_memristor(m2,m4);
2865      digital_current_write_multiple_memristor(m2,m4);
2866
2867      analog_current_write_multiple_memristor(m2,m3);
2868      digital_current_write_multiple_memristor(m2,m3);
2869
2870
2871      analog_variable_read();
2872      digital_variable_read(m5);
2873
2874      analog_variable_V_write_reset();
2875      digital_variable_V_write_reset(m5);
2876
2877      analog_variable_V_write_set(1);
2878      digital_variable_V_write_set(m5);
2879
2880      analog_variable_V_write_set(2);
2881      digital_variable_V_write_set(m5);
2882
2883      analog_variable_V_write_set(3);
2884      digital_variable_V_write_set(m5);
2885
2886      analog_variable_I_write_set(1);
2887      digital_variable_I_write_set(m5);
```

```
2888
2889        analog_variable_I_write_set(2);
2890        digital_variable_I_write_set(m5);
2891
2892        analog_variable_I_write_set(3);
2893        digital_variable_I_write_set(m5);
2894
2895        analog_variable_I_write_set(4);
2896        digital_variable_I_write_set(m5);
2897
2898
2899        analog_VMM();
2900        digital_VMM();
2901    }
```

Listing 8.1: Memristor Struct Definition

Declaration in lieu of oath

I declare in lieu of oath by my signature below

- that I myself have elaborated this thesis without other help, with exception of guidance by my supervisors,
- that I have marked as quotation all text pieces which have been literally or nearly literally adopted from external sources, and
- that I have used only the mentioned sources (literature, internet pages, other means), and
- that all my statements have been made to the best of my knowledge, that they represent the truth and that I did not conceal anything.

I know that a false declaration in lieu of oath will be punished according to §156 and §161 (1) of the German penal code with fine or imprisonment.

place / date	Wei Zhao

Eidesstattliche Erklärung

Ich versichere an Eides Statt durch meine untenstehende Unterschrift,

- dass ich die vorliegende Arbeit - mit Ausnahme der Anleitung durch die Betreuer - selbstständig und ohne fremde Hilfe angefertigt habe und
- dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus fremden Quellen entnommen sind, entsprechend als Zitate gekennzeichnet habe und
- dass ich ausschließlich die angegebenen Quellen (Literatur, Internetseiten, sonstige Hilfsmittel) verwendet habe und
- dass ich alle entsprechenden Angaben nach bestem Wissen und Gewissen vorgenommen habe, dass sie der Wahrheit entsprechen und dass ich nichts verschwiegen habe.

Mir ist bekannt, dass eine falsche Versicherung an Eides Statt nach §156 und nach §161 Abs. 1 des Strafgesetzbuches mit Freiheitsstrafe oder Geldstrafe bestraft wird.

Ort, Datum	Wei Zhao