Universität Siegen
Naturwissenschaftlich-Technische Fakultät
Department Elektrotechnik und Informatik
Lehrstuhl für Digital Integrated Systems

Masterarbeit

Evaluation model of a parametrizable ReRAM core in an FPGA architecture

by
Hrishikesh Balkrishna Karande
Matriculation number
1701995

Examiner
Prof. Dr.-Ing. Michael Wahl
2nd Examiner
Prof. Dr.-Ing. Bing Li

Date of delivery
30.10.2025

Contents

List of Figures

List of Tables

1 Abstract

This thesis presents the design, implementation and FPGA-based emulation of a Resistive Random Access Memory (ReRAM) model, targeting the Xilinx Zybo Z7-7020 development board. A behavioral SystemVerilog model of a single ReRAM cell is developed, incorporating key device characteristics such as Low Resistance State (LRS), High Resistance State (HRS), write delays, and endurance limits. The model is extended to support data storage through the addition of a data-in interface and is integrated into a scalable ReRAM core array for system-level evaluation. The design is synthesized and deployed on the FPGA, with an AXI4-Lite interface enabling communication with the on-chip ARM processing system for control and data exchange. This platform provides a foundation for exploring ReRAM-based accelerators, including the integration of convolutional neural network (CNN) workloads. The work emphasizes reproducibility, with all design files, testbenches, and documentation structured for seamless replication and further research. The complete project, including source code and comprehensive documentation, is available in this GitHub repository:`https://github.com/hrishikeshkarande/CNN-ReRAM_on_FPGA`.

2 Introduction

2.1 Background & Motivation

Resistive Random Access Memory (ReRAM) [1] has emerged as a promising non-volatile memory technology due to its scalability, low power consumption, and potential for in-memory computing. Unlike conventional charge-based memories, ReRAM stores information by switching between distinct resistance states—Low Resistance State (LRS) and High Resistance State (HRS)—through the application of electrical stimuli. This property enables not only high-density storage but also the possibility of embedding computation within the memory array, significantly reducing data movement overhead.

In recent years, the growing computational demands of data-intensive applications, particularly in artificial intelligence (AI) and machine learning, have intensified interest in hardware accelerators that leverage emerging memory technologies. ReRAM's ability to combine storage and computation in a single structure positions it as a candidate for efficient implementations of neural networks and other parallel processing workloads.

However, physical ReRAM devices are still in active development and not widely accessible for experimental research. FPGA-based emulation provides a practical solution, allowing researchers to model ReRAM behavior, test architectures, and validate algorithms before fabrication. By implementing a configurable, cycle-accurate behavioral model of ReRAM on the Xilinx Zybo Z7-7020 board, this work aims to bridge the gap between device-level models and system-level applications. Such an emulation platform supports reproducible experimentation, facilitates performance evaluation, and paves the way for integrating ReRAM into real-time AI accelerators.

2.2 Problem Statement

The rapid growth of data-intensive applications, particularly in artificial intelligence and machine learning, has created a pressing need for high-performance and energy-efficient computing architectures. Traditional von Neumann systems suffer from the "memory wall" bottleneck, where the cost of moving data between memory and processing units dominates execution time and power consumption.

Emerging non-volatile memory technologies such as Resistive Random Access Memory (ReRAM) offer the potential to overcome these limitations by enabling in-memory computation. However, physical ReRAM devices are not yet widely available for large-scale experimentation, and their fabrication costs can be prohibitive for academic research.

This creates a clear need for a flexible, FPGA-based emulation platform that can accurately model the behavior of ReRAM cells and arrays, including their timing characteristics, endurance limits, and data storage capabilities. Such a platform should support integration with processor systems, enabling real-world software-hardware co-design experiments and facilitating the evaluation of ReRAM-based accelerators for applications such as Convolutional Neural Networks (CNNs).

2.3 Research Objectives

The primary objective of this thesis is to design, implement, and evaluate a behavioral emulation of ReRAM on the Xilinx Zybo Z7-7020 FPGA platform. The specific objectives are as follows:

1. **ReRAM Cell Modeling:** Develop a SystemVerilog-based behavioral model of a single ReRAM

cell, incorporating resistive state switching, write delays, endurance limitations, and data storage capability.

2. **ReRAM Core Design:** Extend the single-cell model into a scalable ReRAM core array, supporting addressable read/write operations and enabling system-level integration.

3. **FPGA Integration:** Implement the ReRAM core on the Zybo Z7-7020 board and interface it with the on-chip ARM processing system using an AXI4-Lite communication protocol.

4. **Application Demonstration:** Integrate the ReRAM emulation platform into a proof-of-concept workload, such as a CNN inference pipeline, to evaluate functional correctness and performance.

5. **Reproducible Research:** Document the design methodology, simulation results, FPGA synthesis steps, and software integration to ensure that the work can be replicated and extended by other researchers.

2.4 Scope of Work

This thesis focuses on the FPGA-based behavioral emulation of Resistive Random Access Memory (ReRAM) for research and prototyping purposes. The work is limited to modeling the functional and timing characteristics of ReRAM at a behavioral level and does not include transistor-level device simulations or fabrication processes. The key aspects of the scope are as follows:

- **Behavioral Modeling:** Development of a cycle-accurate, high-level SystemVerilog model of a single ReRAM cell and its extension into a multi-cell core array.
- **FPGA Implementation:** Deployment of the ReRAM core on the Xilinx Zybo Z7-7020 platform, including synthesis, place-and-route, and hardware validation.
- **Processor Integration:** Implementation of an AXI4-Lite interface for communication between the FPGA fabric and the ARM Cortex-A9 processing system on the Zybo board.
- **Functional Demonstration:** Execution of representative workloads, such as small-scale Convolutional Neural Network (CNN) inference, to illustrate practical application scenarios.
- **Evaluation and Documentation:** Analysis of functional correctness, resource utilization, and performance, along with complete documentation to ensure reproducibility.

The scope explicitly excludes:

- Physical fabrication or testing of actual ReRAM devices.
- Detailed electrical or material-level modeling of resistive switching phenomena.
- Large-scale optimization of CNN architectures beyond proof-of-concept integration.

3 System Architecture

The proposed system implements a behavioral model of Resistive Random Access Memory (ReRAM) on the Xilinx Zybo Z7-7020 FPGA platform, enabling experimental evaluation of memory-in-compute architectures. The design is divided into hardware and software components, with an AXI4-Lite interface bridging the FPGA logic and the ARM Cortex-A9 Processing System (PS).

3.1 Hardware Architecture

The hardware design resides in the FPGA's Programmable Logic (PL) and includes:

- **ReRAM Cell Module (`ReRAM_cell`)**: A behavioral SystemVerilog module modeling a single ReRAM bit-cell, supporting two distinct resistance states — Low Resistance State (LRS) and High Resistance State (HRS). The cell responds to write commands by changing its state according to the input data and simulates write delays and endurance limits.
- **ReRAM Core Module (`ReRAM_core`)**: An array of `ReRAM_cell` instances organized to form a memory block. The core supports addressable read/write operations, decoding incoming addresses, and managing data flow to and from individual cells.
- **AXI4-Lite Slave Interface**: Handles communication between the ARM processor and the ReRAM core, decoding AXI transactions into control and data signals for the memory array.

3.2 Software Architecture

The software runs on the ARM Cortex-A9 PS and performs:

- Initialization and configuration of the AXI interface.
- Sending read/write commands to the ReRAM core.
- Executing application-level workloads, such as small-scale Convolutional Neural Network (CNN) inference, to demonstrate the integration of ReRAM as a computational memory.
- Collecting and logging performance metrics for evaluation.

3.3 ReRAM_cell Module

The `ReRAM_cell` is designed as a parameterized SystemVerilog module with the following features:

- Two resistance states: *LRS* (logic '1') and *HRS* (logic '0').
- Programmable write delay to emulate slower resistive switching compared to SRAM/DRAM.
- Endurance counter to simulate cell degradation after a predefined number of write cycles.
- Data input/output ports for storing and retrieving binary data.

3.4 ReRAM_core Module

The `ReRAM_core` module instantiates multiple `ReRAM_cell` modules to create an $M \times N$ memory array. It includes:

- Address decoding logic for selecting individual cells.
- Row and column selection signals for scalable memory organization.
- Aggregated data buses for read and write operations.
- Control logic to synchronize cell operations with system clock and AXI transactions.

3.5 AXI4-Lite Interface

The AXI4-Lite interface provides a low-complexity, memory-mapped protocol to connect the PS and PL:

- **Write Transactions:** Used to send data and control commands to the `ReRAM_core`.
- **Read Transactions:** Used to retrieve stored data from the ReRAM array.
- **Register Mapping:** The interface maps ReRAM core control signals and memory addresses to predefined registers accessible from the PS.

3.6 System Block Diagram

Figure 1 shows the overall system architecture. The ARM Cortex-A9 PS communicates with the ReRAM core through the AXI4-Lite interface. The ReRAM core consists of an address decoder, control logic, and an $M \times N$ array of `ReRAM_cell` modules. The design is clocked synchronously and validated through both simulation and hardware testing.

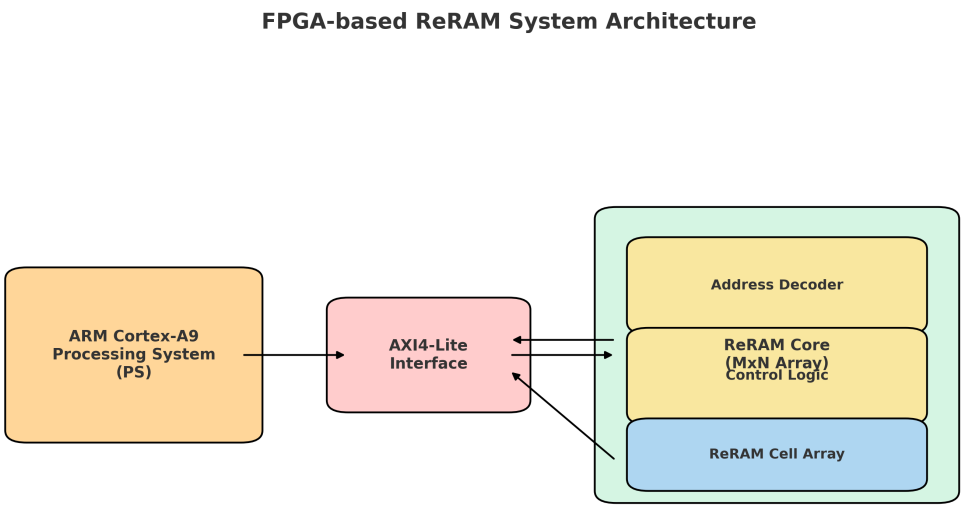

Figure 1: Overall System Architecture

4 Implementation

4.1 ReRAM Cell Behavioral Model

This module models the basic electrical and endurance characteristics of a single ReRAM cell. It's a behavioral model, meaning it describes what the cell does, not how it's implemented at the transistor level.

4.1.1 Module Declaration and Parameters

Listing 1: Example of a ReRAM Cell Module

```
1  `timescale 1ns / 1ps
2  module reram_cell #( /*Parameters*/
3  parameter LRS_STATE = 1'b0, // Low Resistance State (e.g., programmed '0')
4  parameter HRS_STATE = 1'b1, // High Resistance State (e.g., erased '1') - This is our
       default
5  state
6  parameter SET_DELAY_CYCLES = 10, // Requires 10 clock cycles for SET operation (
       changing to LRS)
7  parameter RESET_DELAY_CYCLES = 10, // Requires 10 clock cycles for RESET operation (
       changing to
8  HRS)
9  parameter ENDURANCE_LIMIT = 1000 // Can work for 1000 write cycles
10 ) ( /*Port Defs*/
11 input wire clk,
12 input wire rst_n, // Active low reset
13 input wire write_en, // Asserts to initiate a write operation
14 input wire target_state, // We tell the cell the state that we have to change to (
       LRS_STATE
15 or HRS_STATE)
16 output wire read_data, // Tells us the current state of the cell
17 output wire busy, // Tells us that the cell is undergoing a write operation
18 output wire failed // Tells us that the cell has reached the endurance limit
19 );
```

`timescale 1ns / 1ps : Defines the simulation time unit (1ns) and precision (1ps).

module reram_cell #(...) (...) : Standard Verilog/SystemVerilog module definition with parameters and ports.

Parameters (#(...)): These allow you to customize the cell's behavior without changing the code directly.

- **LRS_STATE** and **HRS_STATE**: Define the logical values representing the Low Resistance State (LRS) and High Resistance State (HRS). Here, LRS is '0' and HRS is '1'. HRS is also the default reset state.
- **SET_DELAY_CYCLES / RESET_DELAY_CYCLES**: Number of clock cycles required for a write operation (SET or RESET). This models the time it takes for the resistive switching to occur.

- **ENDURANCE_LIMIT**: The maximum number of write cycles (SET or RESET) the cell can reliably perform before it "fails."

Ports (`(...)`): The inputs and outputs of the module.

- **clk**: Clock signal, synchronizes all sequential logic.
- **rst_n**: Active-low reset.
- **write_en**: When asserted, it initiates a write.
- **target_state**: Specifies whether to SET (to LRS) or RESET (to HRS) the cell.
- **read_data**: Outputs the current stored state of the ReRAM cell.
- **busy**: Indicates if a write operation is in progress.
- **failed**: Indicates if the cell has exceeded its **ENDURANCE_LIMIT**.

4.1.2 Internal Signals

Listing 2: Internal Signals and Logic for the ReRAM Cell Module

```
/*Internal Signals*/
//Internal state of the ReRAM cell (LRS_STATE or HRS_STATE)
reg current_state;
//Assignment from internal state to output, This is combinational assignment
assign read_data = current_state;
//Counter for SET/RESET delays
reg [31:0] delay_counter;
//State machine for write operations (IDLE, SETTING, RESETTING)
localparam IDLE = 2'b00;
localparam SETTING = 2'b01;
localparam RESETTING = 2'b10;
reg [1:0] write_fsm_state;
//Endurance Counter
reg [31:0] endurance_counter;
reg cell_failed_flag;
assign busy = (write_fsm_state != IDLE);
assign failed = cell_failed_flag;
```

`reg current_state;:` A register holding the actual data/state of the ReRAM cell. This is the "memory" element.

`assign read_data = current_state;:` A continuous assignment. The `read_data` output simply reflects the `current_state` combinatorially (immediately).

`reg [31:0] delay_counter;:` A counter to track the progress of the SET/RESET delay cycles. 32 bits is a generous size for typical delays.

`localparam IDLE = 2'b00; ...:` Defines the states for the write operation's Finite State Machine (FSM). `localparam` means these are local to the module and cannot be overridden by external parameters.

- **IDLE**: The cell is not performing a write operation.

- **SETTING**: The cell is currently undergoing a SET operation.
- **RESETTING**: The cell is currently undergoing a RESET operation.

`reg [1:0] write_fsm_state;:` The register that holds the current state of the write FSM. It's 2 bits because there are 3 states, requiring at least 2 bits ($2^2 = 4$ possible states).

`reg [31:0] endurance_counter;:` Counts the number of successful write operations.

`reg cell_failed_flag;:` A flag that becomes '1' once the `endurance_counter` reaches `ENDURANCE_LIMIT`.

`assign busy = (write_fsm_state != IDLE);:` Continuous assignment. The `busy` output is '1' whenever the FSM is not in the `IDLE` state (i.e., it's `SETTING` or `RESETTING`).

`assign failed = cell_failed_flag;:` Continuous assignment. The `failed` output simply reflects the `cell_failed_flag`.

4.1.3 Behavioral Modeling (The always Block)

Listing 3: Behavioral Model of the ReRAM Cell

```
/*Behavior Modelling*/
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin // Asynchronous Active-Low Reset
            current_state <= HRS_STATE;
            delay_counter <= 0;
            write_fsm_state <= IDLE;
            endurance_counter <= 0;
            cell_failed_flag <= 1'b0;
            $display("Time %0t: Cell RESET! Initial State: %b, Endurance: %0d", $time
                , HRS_STATE, 0);
        end else begin
            // Check for cell failure due to endurance. This check happens every
                cycle.
            // If already failed, remain failed. If not failed and limit reached, set
                 flag.
            if (!cell_failed_flag && endurance_counter >= ENDURANCE_LIMIT) begin
                cell_failed_flag <= 1'b1;
                $display("Time %0t: Cell FAILED! Endurance Limit (%0d) reached.
                    Current endurance: %0d", $time, ENDURANCE_LIMIT,
                    endurance_counter);
            end

            // Main Write FSM
            case (write_fsm_state)
                IDLE: begin
                    // Only start a write if write_en is asserted AND cell has not
                        failed
                    if (write_en && !cell_failed_flag) begin
                        if (target_state == LRS_STATE) begin
                            write_fsm_state <= SETTING;
                            delay_counter <= 0;
```

```
                            $display("Time %0t: Cell (IDLE->SETTING) for target %b. Current: %b", $time, target_state, current_state);
                        end else if (target_state == HRS_STATE) begin
                            write_fsm_state <= RESETTING;
                            delay_counter <= 0;
                            $display("Time %0t: Cell (IDLE->RESETTING) for target %b. Current: %b", $time, target_state, current_state);
                        end
                    end else if (write_en && cell_failed_flag) begin
                        $display("Time %0t: Cell (IDLE) Write attempt to FAILED cell. Current: %b, Endurance: %0d", $time, current_state, endurance_counter);
                        // Cell remains in IDLE, state does not change, endurance_counter does not increment.
                    end
                end // end IDLE

                SETTING: begin
                    if (delay_counter < SET_DELAY_CYCLES - 1) begin // Counting up to DELAY-1, so DELAY cycles in total
                        delay_counter <= delay_counter + 1;
                        $display("Time %0t: Cell (SETTING) Delay: %0d/%0d. Current State: %b", $time, delay_counter, SET_DELAY_CYCLES, current_state);
                    end else begin // Delay is complete
                        current_state <= LRS_STATE; // State change happens here
                        endurance_counter <= endurance_counter + 1; // Increment on successful write operation
                        write_fsm_state <= IDLE; // Changing state back to IDLE
                        $display("Time %0t: Cell (SETTING->IDLE) COMPLETE. New State: %b, Endurance: %0d", $time, LRS_STATE, endurance_counter + 1);
                    end
                end // end SETTING

                RESETTING: begin
                    if (delay_counter < RESET_DELAY_CYCLES - 1) begin // Counting up to DELAY-1
                        delay_counter <= delay_counter + 1;
                        $display("Time %0t: Cell (RESETTING) Delay: %0d/%0d. Current State: %b", $time, delay_counter, RESET_DELAY_CYCLES, current_state);
                    end else begin // Delay is complete
                        current_state <= HRS_STATE; // State change happens here
                        endurance_counter <= endurance_counter + 1; // Increment on successful write operation
                        write_fsm_state <= IDLE; // Changing state back to IDLE
                        $display("Time %0t: Cell (RESETTING->IDLE) COMPLETE. New State: %b, Endurance: %0d", $time, HRS_STATE, endurance_counter + 1);
                    end
                end // end RESETTING
```

```
61
62                 default: begin // This state should never be reached, if ever reached
                       then take the machine to IDLE
63                     write_fsm_state <= IDLE;
64                 end
65             endcase
66         end // end else (not reset)
67     end // end always
```

`always @(posedge clk or negedge rst_n) begin ... end`: This is a sequential block, meaning its contents are executed on the positive edge of `clk` or the negative edge of `rst_n`. All assignments within this block use non-blocking assignments (`<=`), which is crucial for synchronous logic.

Reset Logic (`if (!rst_n)`): When `rst_n` is low (active-low reset), all internal registers are initialized to their default values.

- `current_state` is set to `HRS_STATE`.
- Counters and FSM state are cleared.
- `cell_failed_flag` is reset to 0.
- A `$display` message confirms the reset.

Endurance Check (`else if (!cell_failed_flag && endurance_counter >= ENDURANCE_LIMIT)`): This logic runs on every positive clock edge (when not in reset).

- It continuously checks if the `endurance_counter` has reached or exceeded the `ENDURANCE_LIMIT` and if the `cell_failed_flag` isn't already set.
- If the limit is reached, `cell_failed_flag` is set to `1'b1`, and a `$display` message announces the cell failure.
- Crucially, once `cell_failed_flag` is set, it stays set (`!cell_failed_flag` becomes false, so this `if` condition won't trigger again for the same cell).

Main Write FSM (`case (write_fsm_state)`): This case statement defines the behavior of the cell based on its current `write_fsm_state`.

IDLE State:

- **Condition to start write**: Checks if `(write_en && !cell_failed_flag)`. A write can only start if `write_en` is asserted and the cell has not failed.
- **Determine target**:
  - If `target_state` is `LRS_STATE`, the FSM transitions to `SETTING`. `delay_counter` is reset.
  - If `target_state` is `HRS_STATE`, the FSM transitions to `RESETTING`. `delay_counter` is reset.
- **Attempt to write failed cell**: `else if (write_en && cell_failed_flag)`: If `write_en` is asserted but the cell has already failed, a message is displayed, but the cell remains in `IDLE`. No state change occurs, and `endurance_counter` is not incremented, modeling a failed write attempt.

SETTING State:
- **Delay countdown**: `if (delay_counter < SET_DELAY_CYCLES - 1)`: The FSM stays in `SETTING` for `SET_DELAY_CYCLES`. The `delay_counter` increments each cycle. The comparison `SET_DELAY_CYCLES - 1` ensures that it counts from 0 up to `SET_DELAY_CYCLES - 1`, covering exactly `SET_DELAY_CYCLES` clock cycles (e.g., for 10 cycles, it counts 0, 1, ..., 9).
- **Completion**: `else`: When the delay is complete (`delay_counter` is `SET_DELAY_CYCLES - 1` and the next clock edge arrives):
  - `current_state <= LRS_STATE;`: The actual ReRAM cell state is updated to LRS.
  - `endurance_counter <= endurance_counter + 1;`: The endurance counter is incremented, as a successful write occurred.
  - `write_fsm_state <= IDLE;`: The FSM returns to the `IDLE` state.
  - A `$display` message confirms the completion.

RESETTING State: Similar to `SETTING`, but the delay is governed by `RESET_DELAY_CYCLES`. Upon completion, `current_state <= HRS_STATE;`, `endurance_counter` increments, and the FSM returns to `IDLE`.

default State: A safety net. If the FSM somehow enters an undefined state, it immediately transitions back to `IDLE` to prevent unpredictable behavior.

5 Conclusion

Insert Text Here

References

[1] P. Carrasco and I. Vourkas, “Fpga implementation of an elementary reram memory control unit”, in *2024 20th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, 2024, pp. 1–4.