An FPGA-Based RISC-V Instruction Set Extension and Memory Controller for Multi-Level Cell NVM

Mina Ibrahim*, Martel Shokry*, Lokesh Siddhu†, Lars Bauer, Hassan Nassar†, Jörg Henkel†

† *Karlsruhe Institute of Technology (KIT), Chair for Embedded Systems (CES), Germany*
† {lokesh.siddhu, hassan.nassar, henkel}@kit.edu

***Abstract*—Non-volatile memory (NVM) technologies, particularly Multi-Level Cell (MLC) NVMs, offer significant potential for increasing memory density. MLC NVMs provide a trade-off between write latency and retention time, where faster writes/stores result in lower retention and slower writes yield higher retention. However, limited work has been done to validate and prototype NVM-based systems in hardware, leveraging this tradeoff at the system level.**

In this paper, we present a novel memory controller architecture and a RISC-V instruction set extension to optimize MLC NVM write operations by balancing speed and retention time. Our custom NVM controller, built around a finite state machine with an AXI memory-mapped interface, efficiently manages read/write operations with enhanced burst transfers, minimizing latency. Additionally, we introduce a fast-store instruction in RISC-V to increasing write performance while addressing retention limitations. These enhancements are implemented in hardware on an FPGA platform. Experimental results show that our controller reduces hardware overhead by 30% compared to conventional designs, and the fast-store instruction improves performance by over 7% for streaming workloads with less than 0.08% hardware overhead.

***Index Terms*—NVM, FPGA, RISC-V**

I. Introduction

Advancements in manufacturing technologies have driven towards adopting non-volatile memories (NVM) that utilize resistive or magnetic properties, rather than charge-based DRAM, as main memories for embedded systems [1]. NVMs offer key advantages, such as high density, low power consumption, and non-volatility, making them strong candidates to replace DRAM. Many NVM types can be enhanced using Multi-Level Cell (MLC) technology [2], [3], which allows multiple bits to be stored in a single cell. For instance, in Phase Change Memory (PCM), bits are stored at different resistance levels. However, precise resistance (or any other physical property) control in MLC requires iterative write algorithms, which degrade write performance. One suggested approach to speed up the write process involves reducing the number of SET iterations, but this shortens data retention, leading to refreshing to maintain data integrity [4], [5]. Increased refreshes reduces memory lifespan and increases energy consumption [6]–[9].

* Both authors contributed equally. This work was partially funded by the by the German Research Foundation (DFG) as part of SPP 2377 project ARTS-NVM (grant: 502308721) and as part of the Research Grant (405422836) "Design and Optimization of Non-Volatile One-Memory Architecture (NVM-OMA)". The authors thank David Göbel for his help on implementing the memory controller.

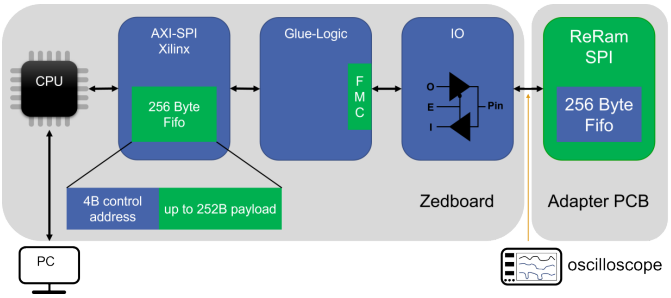

Fig. 1. NVM Memory controller and extension

Moreover, as policies for NVMs become more complex, prototyping and validating these techniques on real hardware is essential for achieving reliable, timely results—unlike simulation-based methods that may not fully capture hardware details and require long simulation runs. For hardware interfacing with NVMs, in terms of the processor selection, RISC-V [10] is an ideal choice due to its open-source nature and flexibility. Unlike proprietary instruction set architectures (ISAs), RISC-V allows developers to customize and extend the architecture to meet specific needs, enabling innovation in hardware design [11]. This open-source approach not only reduces licensing costs but also creates a collaborative ecosystem where academia, industry, and individual developers can contribute to and benefit from a shared pool of resources. Consequently, RISC-V is gaining popularity in embedded systems, IoT, and specialized processors, where adaptability and cost-efficiency are critical [12].

FPGAs are an ideal candidate to implement RISC-V [11]. As RISC-V supports instruction set extensions, FPGAs can be used to implement hardware accelerators for the new instructions [13]. In this paper, we target implementing a RISC-V on FPGA customized for NVM. We aim to implement an extension instruction for RISC-V to handle an extra write mode of MLC for NVM. Moreover, as Figure 1 shows, we target to build our own NVM-specific memory controller on FPGA to be able to interface several commercial NVM chips. In summary our contributions are as follows:

- We create a memory controller to connect NVMs with an FPGA via a custom FMC adapter. Using an FSM and an AXI memory-mapped interface, it efficiently manages read/write operations with burst transfers, allowing precise NVM performance.
- We propose utilizing the trade-off between write latency

and retention time in NVMs by extending the RISC-V instruction set to include a new fast-store instruction.

II. Background

Non-volatile memory (NVM) technology, has evolved to offer increased storage capacity by enabling the storage of multiple digital bits within a single cell, known as Multi-Level Cell (MLC). However, MLC faces challenges related to retention time due to resistance drift, which can lead to errors in reading data and necessitate periodic refresh operations. Figure 2, shows as an example the resistance drift in Phase Change Memory (PCM) [1] which is a key issue for NVM. Research highlights a trade-off between write latency and retention time (Table I): write operations with more SET iterations and lower currents extend retention time but increase write latency, while fewer SET iterations improve write performance but reduce retention time, leading to more frequent refreshes. This trade-off also negatively affects the longevity of MLC NVM [8]. MLC offers greater storage capacity compared to Single-Level Cell (SLC), but it has a notable limitation: a shorter retention time, often lasting only a few hours due to resistance drift. As a result, MLC requires more frequent refresh operations than DRAM. Additionally, the precision of write operations directly affects retention time. Higher precision writes, involving more SET iterations, extend retention time but increase write latency. This creates a trade-off between write latency and retention in MLC. Continuing with the PCM example, a write operation with 7 SET iterations achieves a retention time of 3054.9 seconds, while one with 3 SET iterations results in a much shorter retention time of just 2.01 seconds as shown in Table I. In addition to PCM, other NVM technologies exist such as ReRAM [14], FRAM, MRAM [15], and STTRAM [2].

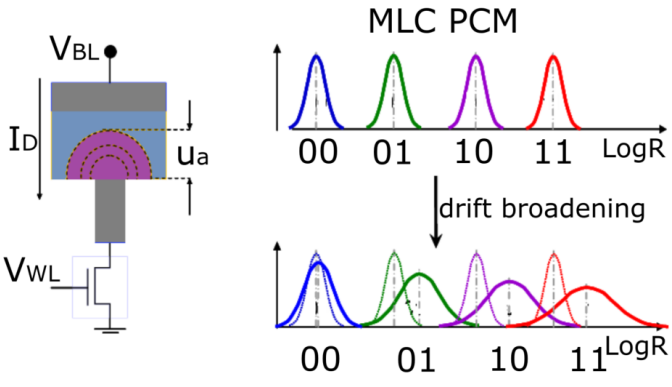

Fig. 2. Drift resistance problem

TABLE I
Trade-off between Write Latency and Retention Time of MLC PCM, based on [4].

Write Type	Latency	Retention	Current	Norm. Energy
3-SETs-Write	550 ns	2.01 s	42 μA	0.84
4-SETs-Write	700 ns	24.05 s	37 μA	0.869
5-SETs-Write	850 ns	104.4 s	35 μA	0.972
6-SETs-Write	1000 ns	991.4 s	32 μA	0.975
7-SETs-Write	1150 ns	3054.9 s	30 μA	1.00

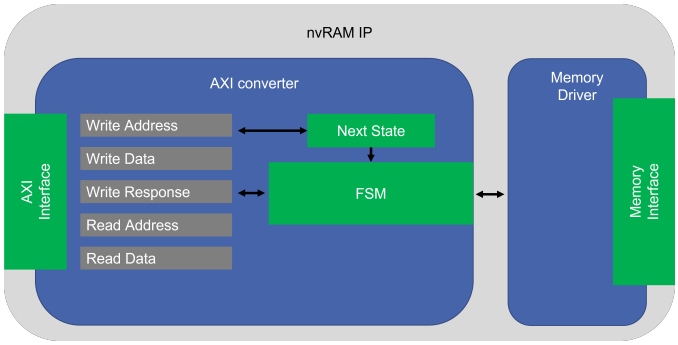

Fig. 3. Design of memory controller with AXI interface

III. Related Work

Many efforts have addressed performance in Multi-Level Cell (MLC) Phase Change Memory (PCM) by utilizing trade-offs between write latency and retention time. The Quick-and-Dirty (QnD) architecture [4] issues fast, low-retention writes during high-demand periods and refreshes data in idle times, boosting system performance while maintaining reasonable memory lifespan, with minimal changes to the memory controller. Some approaches add fast and slow write instructions. Li et al. [16] propose a compiler-directed dual-write (CDDW) scheme to optimize write modes for performance, while Qiu et al. [17] present a write mode-aware loop tiling approach to maximize fast writes by reducing retention requirements. Swift-CNN [5] optimizes PCM for CNNs by adjusting write modes based on each layer's retention needs, reducing inference and training times while lowering energy consumption.

These works rely on simulation, which is slow and susceptible to modeling inaccuracies. In contrast, we develop a hardware/FPGA-based platform for reliable and faster evaluation/emulation.

IV. Extending RISC-V and Implementing our Memory Controller

Using NVM effectively necessitates the development of an NVM memory controller for a RISC-V platform, including the extension of the instruction set for MLC operations. The memory controller involves a custom FSM for NVM interfacing. While the new store instructions improve memory operations, they required changes to hardware (memory controller and cache) and software tools (compiler and assembler).

A. Memory Controller for NVM

The memory controller designed in this work is designed to be able to accomodate different commercial NVM chips from different suppliers. Our NVM memory controller consists mainly of two parts as Figure 3 shows. First, a component to communicate with the processor via an AXI interface. Second, a memory driver to interface with our developed PCB containing the commercial chips. The component communicating with the processor uses an FSM to regulate the communication between the processor and the different NVM chips.

The FSM is designed to control how the memory controller interacts with the non-volatile memory (NVM). It follows a structured approach where each memory access operation

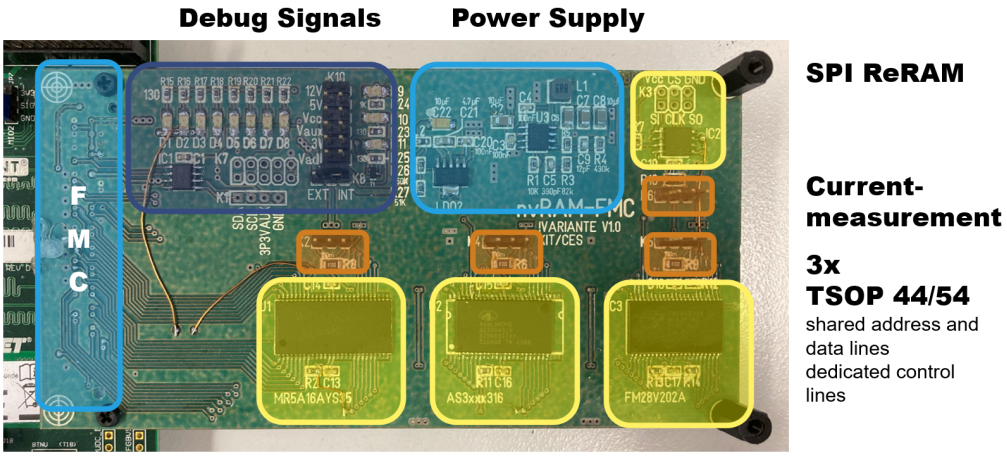

Fig. 4. FMC Adapter board in the 3.3V version with color-coded sections

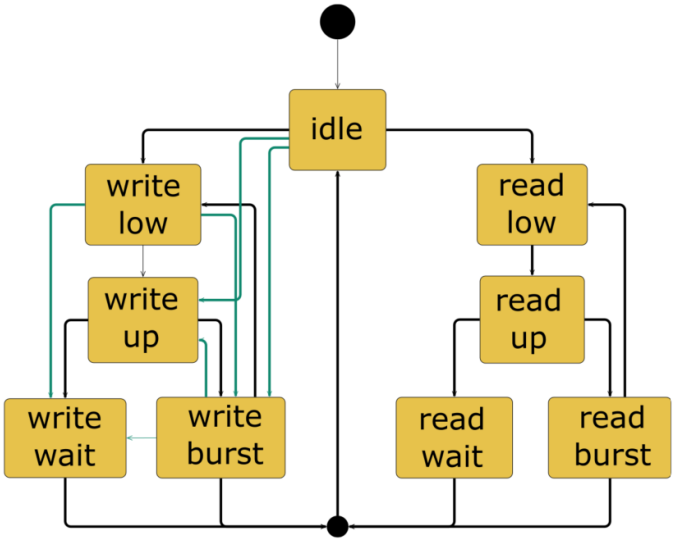

Fig. 5. State Diagram For AXI FSM

(read or write) is broken into smaller, timed phases. For instance:

- Read Cycle: The FSM goes through the following phases—enable (assert memory chip enable), wait (wait for data to be valid), read (sample data), and end (finalize the operation).
- Write Cycle: The FSM includes phases for enable, write (put data on the bus), and recovery (allow memory to stabilize).

These phases ensure proper coordination between the FPGA and the NVM by accounting for delays and ensuring signals are stable before the next phase begins. The clock frequency is set at 100 MHz, so each FSM step corresponds to a 10 ns period.

The FSM also manages the AXI transactions. Since the non-volatile memories work at a 16-bit width and the AXI bus operates at a 32-bit width, the FSM splits the AXI data into two parts: the lower 16 bits and the upper 16 bits. Each part is processed sequentially in separate states (low, up) for both read and write operations. State Diagram: The FSM starts in an idle state, then moves through states for reading and writing the lower 16 bits (read low, write low), and then for the upper 16 bits (read up, write up). The FSM returns to idle once the full 32-bit word is processed as shown in Figure 5.

The AXI interface was further optimized by introducing burst transfers, which allow multiple sequential memory accesses to be performed with a single address transmission. This reduces overhead and speeds up large data transfers, such as those involved in direct memory access (DMA) operations.

The FSM for AXI is designed to handle both the lower and upper 16 bits of a memory word sequentially. However, additional logic is introduced to skip unnecessary FSM states when, for example, only one part (either lower or upper) needs to be processed. This ensures that operations involving only partial data do not waste cycles. The AXI bus protocol automatically manages the synchronization between the CPU and the memory controller, removing the need for manual synchronization as in the earlier GPIO-based approach. This simplifies the software interaction with the memory controller and eliminates potential data corruption or mismatches which can be shown using the block diagram in Figure 3.

1) PCB Layout

The memory controller interfaces with the NVM memories laid out on a PCB board. The PCB provides connections for current measurements are highlighted in orange in Figure4, which shows the completed board. Current sense resistors are located near them. Yellow highlights mark the footprints and mounted nvRAMs, with the TSOP memory chips placed on the bottom half of the board. Although address and data lines are numbered differently, they share the same pin locations. Larger memory chips with more address lines are arranged to accommodate smaller chips on the same footprint. The connections follow a bus topology horizontally, routed vertically via vias. Signals are grouped into 8-bit blocks to facilitate future integration of level shifters between the FMC connector (blue, bottom) and the first memory chip.

All three nvRAMs share control signals: *ChipSelect* (CS), *OutputEnable* (OE), and *WriteEnable* (WE), with identical supply pin locations. The differences lie in the pull-up resistor configurations during operation and startup. FRAM has an additional *Sleep* pin, absent in newer MRAM revisions, where it remains unconnected.

The power supply section (light blue) is positioned op-

posite the nvRAMs to isolate high-frequency signals from the switching regulator. Between the power supply and FMC connector are debug LEDs and pin headers for spare signals. Control LEDs for 12 V, 5 V, and 3.3 V supplies are also placed here. Generated and FMC-provided voltages are accessible via pin headers for debugging. A jumper (K8 EXIT-INT) allows switching between board-generated and external 3.3 V power from the FMC connector, enabling external power use in case of failure or minimal assembly.

An I2C EEPROM near the FMC connector supports potential IPMI integration, storing board voltages or a unique identifier. It can also help distinguish between board versions during setup, such as identifying defects like the "FMC card is inserted" signal issue.

B. RISC-V Extension for NVM Write Mode

After implementing our memory controller we had to extend RISC-V to support MLC with a new instruction. In order to integrate the new instruction into the compiler toolchain, available opcodes were identified from the RISC-V instruction set manual [18]. Four opcodes were selected to accommodate varying store granularities, ranging from byte to double-word operations. These opcodes were located under the *S-type* main opcode, with minor opcodes ranging from four to seven, where opcode four corresponds to the fast store byte instruction, and opcode seven corresponds to the fast store double-word instruction.

Four new C macros were implemented to support all data granularities. Store byte fast instruction is shown in figure 6 to illustrate the used format. These macros utilize inline assembly to directly pass the instructions to the assembler. The function parameters consist of the data to be stored and the target address. The *volatile* qualifier is employed to ensure that the assembler does not eliminate the instructions due to their lack of output. Additionally, the *'memory'* clobber is used to prevent the compiler from reordering instructions around new instructions. The instruction format follows the S-type structure, similar to the standard store format, as it requires the same parameters. It needs a data register, a register containing the address, and another for the immediate offset, which is used to compute the final address. Additionally, the output field in the assembly instruction is left blank, as store instructions do not have any outputs.

```
#define store_fast_byte(data, addr) \
 asm volatile("sbf%0,%1(%2)\n"::"r"(data),"i"(0),"r"(addr):"memory");
```

Fig. 6. Instruction Format

The assembler files, particularly the opcode table, were modified to support the new inline assembly instructions. These table modifications follow the same structure as the entries for standard store instructions, with the primary difference being the match field. This adjustment ensures that the match field sets the appropriate minor opcode bits, which differ from the standard store instructions, in the machine code.

After ensuring the machine code for the new instructions was correctly set, changes were introduced in the HDL to decode and implement the functionality of the new instructions. In the decode stage, four hot bits, covering all fast store granularities, were added to the instruction vector. Once decoded, the new instructions follow a similar path as ordinary store operations in terms of setting critical flags, such as the store/read flag and memory size indicator. To propagate fast store status from execute stage to cache stage, a fast store flag was added to a propagating package.

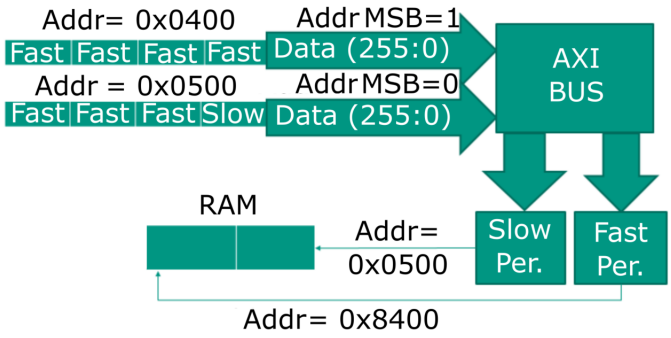

Fig. 7. Cache behavior for slow and fast writes. If all words written in a cache line can use fast write, it is written back to memory using fast write mode. Otherwise, it uses slow write mode. Slow mode and fast mode each has a different address range.

1) Cache Stage Modification in VHDL

To be able to use the new memory instruction effectively, modification of Cache is needed. The cache uses a write-back policy with each line consisting of four double words and follows an incremental burst strategy during write-back or flushing. It includes standard flags like dirty, valid, and shared. Four additional fast store flags were added to these flags, where each one indicates fast write status for each double word in the cache line. If all fast store flags are set, the entire cache line is forwarded to the fast store peripheral by setting the address MSB to 1. Otherwise, the cache line is sent to the slow store peripheral by leaving the address unchanged. An illustration of the typical cache behavior for fast and slow writes is provided in Figure 7.

C. Experimental Setup

After designing the memory controller and extending RISC-V with new instructions, we built an experimental setup to measure access latency for various NVM technologies and verify the RISC-V extension.

1) Memory Controller Testing

We implemented the system in VHDL and tested our memory controller, focusing on access latency for different NVMs, including ReRAM, FRAM, and MRAM. ReRAM used a serial SPI interface, while FRAM and MRAM required an 8-bit parallel interface. We used interfaces provided by the commercial NVM chips (Table II). Figure 8 shows the general setup. ReRAM was mounted on an adapter board connected to a Zedboard via FMC pins. The processor executed C test programs implemented as console applications initiated by

TABLE II
SELECTED AND ORDERED NVRAMS

Manufacturer	Technology	Name	Size	Interface
Avalanche	STT MRAM	AS3004316	4 Mb	Parallel x16
Infineon	FRAM	FM28V202A-TG	2 Mb	Parallel x16
Fujitsu	ReRAM	MB85AS8MT	2 Mb	SPI (Serial)

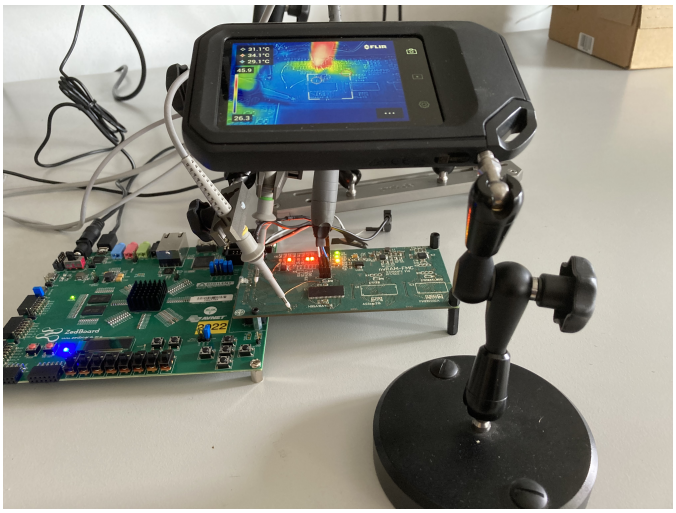

Fig. 8. NVM performance measurement setup. The developped PCB is connected to a Zedboard via FMC. Oscilloscope probes are connected to collect the results and a thermal camera is used to ensure that no overheating occurs.

user input. Most tests ran automatically, generating target addresses and test data. A byte stream, composed of a control command, 3 address bytes, and 252 data bytes, was sent to the AXI_to_SPI IP block. Data passed through a 256-byte FIFO before transmission. Glue logic mapped SPI signals to the FMC pins, enabling data transfer from the FPGA to ReRAM, which also had a 256-byte FIFO for data. Power consumption was measured with an oscilloscope connected directly to the ReRAM.

For speed measurements, after a write command was sent, the 'write in progress' bit of the status register was queried to confirm when the transfer from FIFO to NVM was complete. Performance varied by packet size and pattern, and a test program wrote data in different sizes and patterns to measure this. The CPU timed the write and read operations to calculate performance.

2) Emulating the New RISC-V Instruction

As the NVM chips did not provide access to different MLC write modes, the RISC-V write mode extension was emulated using RAM with added delays. To support our new instruction, we used the GCC toolchain compatible with RISC-V to compile, assemble, and link the program. *Startup.s* and the provided RISC-V source codes were used in the process. The compiled executable was converted into two HEX files (upper and lower 32 bits) using *elf2rawx*. These were uploaded to the RISC-V processor via HDL code, and the ROM was initialized with the program. Output was observed via UART serial communication. Figure 9 illustrates the compilation process.

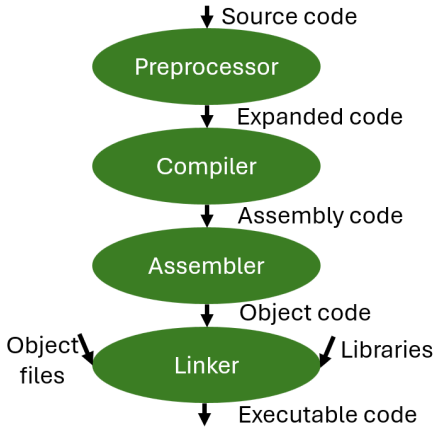

Fig. 9. Compilation process

V. RESULTS

Our evaluation focuses on the hardware overhead and timing performance of the custom memory controller and new RISC-V instructions. We compare the custom memory controller with Xilinx's controller, assess the minimal impact of the new instructions, and review read/write latency for ReRAM, FRAM, and MRAM.

A. Hardware Overhead

We first analyze the hardware overhead of using our memory controller and adding the new instruction. Adding the custom memory controller has saved considerable hardware cost as shown in Table III. We have used Xilinx's auto-generated memory controller for reference. It introduced 339 additional flip flops and 429 lookup tables compared to 450 flip flops and 563 lookup tables of the built-in Xilinx controller, yielding a 30% reduction in hardware cost (by developing a dedicated NVM memory specific controller).

TABLE III
HARDWARE UTILIZATION OF THE MEMORY CONTROLLER

Utilization of	**LUTs**	**Registers**
Xilinx Built-in Memory Controller	450	563
Custom Memory Controller	339	429

Moreover, the hardware utilization results of adding the new instruction is shown in table IV, where the LUTs have increased by 0.03% and registers are approximately the same.

TABLE IV
HARDWARE UTILIZATION OF THE INSTRUCTION SET EXTENSION

Utilization of RISC-V in %	**LUTs**	**Registers**
Initially	40.59	10.16
After adding new instruction	40.56	10.19

B. Timing Analysis

Next, we investigate the delay of using the different NVM commercial chips. Regarding ReRAM speed performance, Figure 10 shows how packet size affects the write and read

times. Time per byte is decreasing with the increasing number of bytes in a packet due to the huge overhead of the SPI commands with additional commands for activation and polling. After reaching the FIFO limit of 252 bytes, the packets need to split as no improvements are possible.

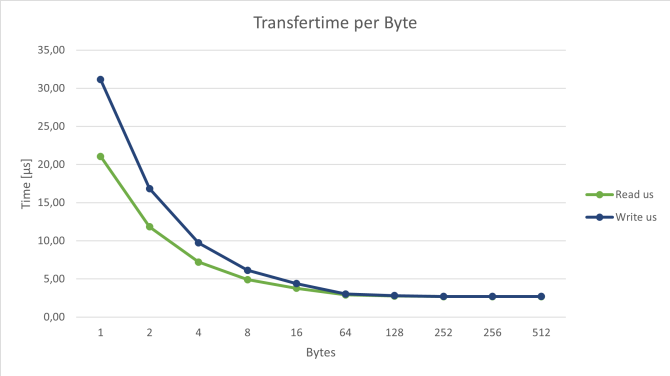

Fig. 10. Transfer Time for Different Packet Sizes

For brevity we do not show the curves of the other NVM chips but rather the ReRAM performance results are compared to FRAM and MRAM, as shown in Table V. The table presents the access time per byte for 64-byte transactions.

TABLE V
ACCESS TIME PER BYTE

Type Of Memory	FRAM	MRAM	ReRAM
Read Time	61 ns	41 ns	1.6 us
Write Time	86 ns	66 ns	15 us

As discussed in Experimental setup, a configurable delay is implemented in the SRAM to emulate the MLC behavior. Figure 11 and Figure 12 demonstrate this behavior. Further, we excecuted a streaming workload with both slow and fast store variants. Cache hits were observed for accesses within the cache line width, while the remaining accesses were serviced via main memory. The fast-store workload outperformed the slow-store workload by 7.7%. As mentioned earlier, we emulated the slow write with a 5-cycle delay. This cycle difference is configurable, and users can adjust the delay to suit specific memory types.

VI. CONCLUSION

This paper presented an innovative approach to improving MLC NVM performance through a custom NVM memory controller and RISC-V instruction set extension. Our controller, utilizing a finite state machine and AXI interface, enables efficient memory access with burst transfer capabilities, while minimizing hardware overhead. The proposed fast-store instruction reduces write latency by addressing the trade-off between write speed and retention time in MLC NVMs. FPGA-based experiments with multiple NVMs confirmed the design's performance improvements in speed and resource usage. These contributions enhance MLC NVM efficiency in embedded systems and memory-intensive applications, laying the groundwork for future optimization of NVM architectures.

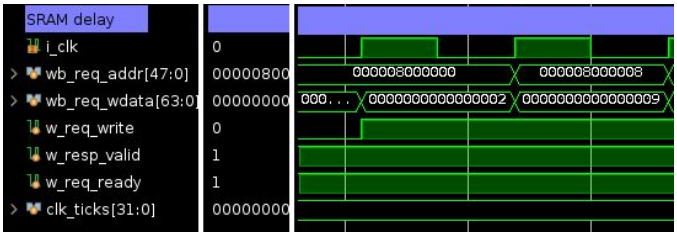

Fig. 11. SRAM signals with no delay

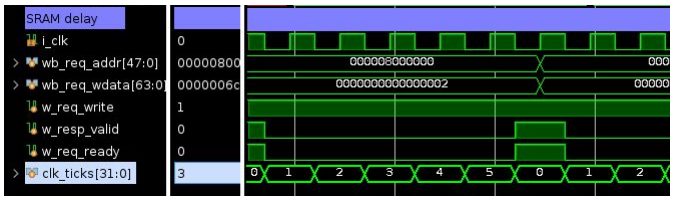

Fig. 12. SRAM signals with delay equal to 5 clock cycles which can be observed from the clk_ticks signal which means that the write response valid and write request ready are delayed by 5 clock cycles to delay the next write instruction.

VII. ACKNOWLEDGMENTS

This work is supported by German Research Foundation (DFG): ARTS-NVM (part of SPP2377) and NVM-OMA (Project Number: 405422836) projects.

REFERENCES

[1] G. W. Burr *et al.*, "Phase change memory technology", *Journal of Vacuum Science & Technology B*, 2010.
[2] C. Pan *et al.*, "Exploiting multiple write modes of nonvolatile main memory in embedded systems", *ACM TECS*, 2017.
[3] H. Nassar *et al.*, "ANV-PUF: Machine-Learning-Resilient NVM-Based Arbiter PUF", *ACM TECS*, 2023.
[4] M. Zhang *et al.*, "Quick-and-dirty: An architecture for high-performance temporary short writes in mlc pcm", *IEEE TC*, 2019.
[5] L. Siddhu *et al.*, "Swift-cnn: Leveraging pcm memory's fast write mode to accelerate cnns", *IEEE ESL*, 2023.
[6] M. K. Qureshi *et al.*, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling", in *MICRO*, 2009.
[7] N. Hölscher *et al.*, "Memory Carousel: LLVM-Based Bitwise Wear-Leveling for Non-Volatile Main Memory", *IEEE TCAD*, 2023.
[8] J. J. Kan *et al.*, "A study on practically unlimited endurance of stt-mram", *IEEE Transactions on Electron Devices*, 2017.
[9] H.-C. Yu *et al.*, "Cycling endurance optimization scheme for 1mb stt-mram in 40nm technology", in *International Solid-State Circuits Conference Digest of Technical Papers*.
[10] P. Kanjarbhat *et al.*, "Implementation of pwm using risc-v processor", in *AMATHE*, 2024.
[11] D.-T. Nguyen-Hoang *et al.*, "Implementation of a 32-bit risc-v processor with cryptography accelerators on fpga and asic", in *IEEE ICCE*, 2022.
[12] A. P *et al.*, "Implementation of risc-v instruction set architecture for edge iot computing platform", in *ICAECT*, 2024.
[13] H. Nassar *et al.*, "Supporting dynamic control-flow execution for runtime reconfigurable processors", in *ICM*, 2023.
[14] Y.-C. Chen *et al.*, "Dynamic conductance characteristics in hfox-based resistive random access memory", *RSC Adv.*, 2017.
[15] J. F. Scott, *A Comparison of Magnetic Random Access Memories (MRAMs) and Ferroelectric RandomAccess Memories (FRAMs)*. Springer Berlin Heidelberg, 2007.
[16] Q. Li *et al.*, "Compiler directed write-mode selection for high performance low power volatile PCM", in *LCTES*, 2013.
[17] K. Qiu *et al.*, "Write mode aware loop tiling for high performance low power volatile pcm in embedded systems", *TC*, 2015.
[18] A. Waterman *et al.*, "The RISC-V instruction set manual", May 2017. [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf