AN EFFICIENT FLASH TRANSLATION LAYER IMPLEMENTATION FOR NOR FLASH MEMORY

by

Fatih Çam

B.S., Electrical & Electronics Engineering, Middle East Technical University, 2019

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electrical & Electronics Engineering
Boğaziçi University

2023

ACKNOWLEDGEMENTS

I want to express my gratitude to Assistant Professor Faik Başkaya for his help and direction during the thesis period. His knowledge enabled me to retrace my steps and create a professional thesis that complies with all academic requirements.

This thesis is the culmination of everything I've learned during my undergraduate and graduate studies. Therefore, I also would like to thank the Middle East Technical University and Boğaziçi University faculty members who teach in the electrical and electronics engineering departments.

My company, TUBITAK BILGEM, gave me access to all the resources I required and provided me with the training to produce my thesis in a more professional manner. I thank my coworkers from YITAL, whose friendship I value greatly. My colleague Ercan Doğan deserves special appreciation.

ABSTRACT

AN EFFICIENT FLASH TRANSLATION LAYER IMPLEMENTATION FOR NOR FLASH MEMORY

As technology develops day by day, the amount of data that electronic systems need to manage is increasing in direct proportion. Especially in embedded systems, the necessity to use space efficiently does not allow flash memories used as storage units to exceed a certain size. The fact that the technology on which flash memories are built has a certain lifetime has made it necessary to use these technologies in the most effective way. The algorithm developed as a solution to this problem is the flash translation layer (FTL). Different FTL algorithms have been proposed over the years, and these algorithms have led to some disadvantages along with many advantages. The aim of this thesis is to present a much simpler and implementable FTL algorithm, which is different from the complex algorithms proposed before, and which also shows high success. For read and write operations, the target page address is found directly from the mapping table and the operation is performed. In the write operation, the target page address is determined by the value from the random number generator. If data has been written to this flash address before, the data at this address is moved to the next free flash address and the data to be written is written to the page address from the random number generator. With this method, page writing is carried out quickly and effectively. The algorithm was implemented on the FPGA and tested over UART with test software designed on the computer. The results of the test application showed that the designed algorithm used flash memory pages with a regular distribution and the success rate was 98%. In addition, only one flash memory page sacrifice was sufficient to use this algorithm. Thus, this study did not cause space loss in flash memory, unlike other algorithms.

ÖZET

NOR FLASH BELLEKLER İÇİN VERİMLİ BİR FLASH ÇEVİRİ KATMANI UYGULAMASI

Teknoloji her geçen gün geliştikçe elektronik sistemlerin yönetmesi gereken veri miktarı da doğru orantılı bir şekilde artmaktadır. Özellikle gömülü sistemlerde, alanı verimli kullanma zorunluluğu depolama birimi olarak kullanılan flash belleklerin belli bir boyutu geçmesine izin vermemektedir. Flash belleklerin üzerine inşa edildiği teknolojinin belli bir kullanım ömrünün olması bu teknolojilerin en efektif bir şekilde kullanımını zorunlu hale getirmiştir. Bu probleme çözüm olarak geliştirilen algoritma flash çeviri katmanıdır (FTL). Yıllar boyunca farklı FTL algoritmaları önerilmiş ve bu algoritmalar birçok avantajla beraber bazı dezavantajlara da yol açmıştır. Bu tezin amacı daha önce önerilen kompleks algoritmalardan farklı olarak çok daha basit ve implement edilebilir aynı zamanda yüksek başarı gösteren bir FTL algoritması sunmaktır. Okuma ve yazma işlemleri için hedef sayfa adresi direkt olarak eşleme tablosundan bulunmakta ve operasyon gerçekleştirilir. Yazma operasyonunda ise hedef sayfa adresi rastgele sayı üreticisinden gelen değer ile belirlenir. Eğer bu flah adresine daha önceden veri yazılmışsa bu adresteki veri bir sonraki boş flash adresine taşınır ve yazılacak veri rastegele sayı üreticinden gelen sayfa adresine yazılır. Bu yöntemle sayfa yazma işlemi hızlı ve efektif olarak gerçekleştirilmektedir. Algoritma FPGA üzerinde implement edilmiş ve bilgisayar üzerinde tasarlanan bir test yazılımı ile UART üzerinden test edilmiştir. Test uygulamasının sonuçları göstermiştir ki tasarlanan algoritma flash bellek sayfalarını düzenli bir dağılımla kullanmış ve başarı oranı %98 olmuştur. Ayrıca bu algoritmanın kullanılması için sadece bir flash bellek sayfasının ayrılması yetmiştir. Böylelikle bu çalışma diğer algoritmaların aksine flash bellekte alan kaybına sebep olmamıştır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . viii
LIST OF SYMBOLS . . . xi
LIST OF ACRONYMS/ABBREVIATIONS . . . xii
1. INTRODUCTION . . . 1
2. BACKGROUND . . . 4
2.1. Memory Storage Architectures . . . 4
2.1.1. SRAM Technology . . . 5
2.1.2. EEPROM . . . 8
2.1.3. Flash Memory . . . 11
2.1.3.1. Read Operation . . . 13
2.1.3.2. Program Operation . . . 14
2.1.3.3. Erase Operation . . . 16
2.1.3.4. Flash Memory Structure . . . 19
2.1.3.5. Restrictions of Flash Memory Structure . . . 22
3. FLASH TRANSLATION LAYERS . . . 24
3.1. FTL Classifications . . . 25
3.1.1. Page Level Translation Layers . . . 25
3.1.2. Block Level Translation Layers . . . 29
3.1.3. Hybrid Translation Layers . . . 30
3.2. FTL Related Works . . . 32
3.2.1. DFTL: Demand Based Flash Translation Layer . . . 32
3.2.2. CFTL: Convertable Flash Translation Layer . . . 33
4. RNG BASED FLASH TRANSLATION LAYER (RFTL) . . . 35
4.1. Read Operation . . . 36
4.2. Erase Operation . . . 37

4.3. Write Operation 38
4.3.1. Case 1 38
4.3.2. Case 2 39
4.3.3. Case 3 41
4.3.4. Case 4 43
5. RTL DESIGN OF THE ALGORITHM 45
5.1. Structure of the RFTL Algorithm 49
5.1.1. Flash Memory Block 49
5.1.2. Mapping Table 51
5.1.3. RNG Block 52
5.1.4. Controller Finite State Machine 53
6. TEST RESULTS AND COMPARISON 54
6.1. Test Setup 54
6.2. Test Results 57
6.3. Comparison 60
7. CONCLUSION 63
REFERENCES 65

LIST OF FIGURES

Figure 2.1. The Structure of a MOSFET. 6

Figure 2.2. Conduction Channel Formation of N-Type MOSFET. 6

Figure 2.3. SRAM Memory Cell. 7

Figure 2.4. Floating-Gate Transistor Configurations. 9

Figure 2.5. Impact of Charge on a Floating-Gate Transistor. 10

Figure 2.6. NOR and NAND Flash Cell Allignment. 12

Figure 2.7. NAND Flash Memory Program Operation. 15

Figure 2.8. NOR Flash Memory Program Operation. 16

Figure 2.9. Flash Memory Erase Operation. 17

Figure 2.10. Flash Memory Functional Block Diagram. 19

Figure 2.11. NAND Flash Memory Architecture. 20

Figure 2.12. Dual Buffer Flash Memory Architecture. 21

Figure 3.1. Page Mapping Scheme of FTL. 26

Figure 3.2. Block Mapping Scheme of FTL. 28

Figure 3.3. Hybrid Mapping Scheme of FTL. 31

Figure 4.1. RFTL Architecture. 35

Figure 4.2. Read Operation in RFTL Algorithm. 36

Figure 4.3. Erase Operation in RFTL Algorithm. 37

Figure 4.4. Case 1 of Write Operation in RFTL Algorithm. 38

Figure 4.5. Case 1 Resulting Scheme of Write Operation in RFTL Algorithm. 39

Figure 4.6. Case 2 of Write Operation in RFTL Algorithm. 40

Figure 4.7. Case 2 Resulting Scheme of Write Operation in RFTL Algorithm. 41

Figure 4.8. Case 3 of Write Operation in RFTL Algorithm. 42

Figure 4.9. Case 3 Resulting Scheme of Write Operation in RFTL Algorithm. 43

Figure 4.10. Case 4 Resulting Scheme of Write Operation in RFTL Algorithm. 44

Figure 5.1. Basic SoC Architecture. 45

Figure 5.2. SoC Architecture with FTL Addition. 47

Figure 5.3. The Design Structure of RFTL Algorithm. 49

Figure 5.4. The Design Structure of Flash Memory. 50

Figure 5.5. The Design Structure of the Mapping Table. 51

Figure 5.6. The Schematic of the LFSR. 52

Figure 5.7. The State Diagram of the Controller Logic. 53

Figure 6.1. The Test Setup of the RFTL Algorithm. 54

Figure 6.2. The User Interface of the Test Software. 56

Figure 6.3. The Usage Map of the Test Interface. 58

Figure 6.4. The Heat Map of the Test Interface. 58

Figure 6.5. The Distribution of the RNG Outputs. 60

Figure 6.6. The Comparison of RFTL Algorithm with Other FTLs. 61

LIST OF SYMBOLS

x^n	Nth degree of x
$\div$	Division operation
mod	Modulus operation

LIST OF ACRONYMS/ABBREVIATIONS

ADC	Analog to Digital Converter
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
AXI	Advanced Extensible Interface
BAST	Block Associative Sector Translation
BIOS	Basic Input/Output System
BL	Bitline
CAN	Controller Area Network
CBMT	Cached Block Mapping Table
CF	Compact Flash
CFTL	Convertable Flash Translation Layer
CHE	Channel Hot Electron
CLB	Configurable Logic Blocks
CMT	Cached Mapping Table
CPMT	Cached Page Mapping Table
CPU	Central Processing Unit
DAC	Digital to Analog Converter
DFTL	Demand-based Flash Translation Layer
DRAM	Dynamic Random Access Memory
ECC	Error Correction Code
EEPROM	Electronically Erasable Programmable Read-Only Memory
FG	Floating Gate
FMM	Flash Memory Manager
FN	Fowler Nordheim
FPGA	Field Programmable Gate Array
FSM	Finite State Machine

FTL	Flash Translation Layer
GB	Giga Byte
GPIO	General Purpose Input Output
I/O	Input / Output
LBN	Logical Block Number
LFSR	Linear-Feedback Shift Register
LPN	Logical Page Number
LPV	Logical Page Valid
LRU	Least Recently Used
MLC	Multi-Level-Cell
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
NVRAM	Non-Volatile Random Access Memory
OOB	Out of Bound
OTP	One-Time Programmable
PBN	Physical Page Number
PC	Personal Computer
PCIE	Peripheral Component Interconnect Express
POR	Power Off Recovery
PPN	Physical Page Number
PPP	Partial-Page Programming
PPV	Physical Page Valid
PWM	Pulse Width Modulation
RAM	Random Access Memory
RNG	Random Number Generator
ROM	Read-Only Memory
RTC	Real Time Clock
RTL	Register Transfer Level
SLC	Single-Level-Cell
SOC	System-on-a-Chip
SPI	Serial Peripheral Interface

SRAM	Static Random Access Memory
UART	Universal Asynchronous Receive Transmit
USB	Universal Serial Bus
WL	Wear Levelling
WL	Wordline

1. INTRODUCTION

Flash memory comes in two types, NOR and NAND. NOR flash is an excellent substitute for conventional read-only memory (ROM), such as the BIOS chip of computers, because it has independent address and data buses, allowing random access to any memory position. However, NAND can only be accessed in pages since address and data share the same I/O interface, despite having a better density and a lower cost per bit than NOR. NAND is typically utilized as a backup storage option. A certain number of blocks serve as the fundamental building blocks for erase operations on each flash chip. Additionally, each block comprises a fixed number of pages, which serve as the fundamental units for reading and writing. For out-of-band (OOB) data like the error correction code (ECC), the logical page number, and the page status flag, most flash memory additionally includes a spare region for each page.

Inherently superior to traditional hard disks, flash memory offers non-volatility, quick access, stress tolerance, and low power consumption. As a result, it has been widely used in embedded applications such as mobile devices, CF card memory, USB flash memory, and so on. Flash memory-based programs, however, require particular processes while reading from (writing to) flash memory due to their hardware features. Erase-before-write architecture is one of the fundamental hardware features of flash memory. This means that before new data can be written to a spot on flash memory, the location must first be erased. Additionally, the erase unit (block) is larger than the read or write unit (sector), which significantly reduces the performance of the entire flash memory system.

FTL (Flash Translation Layer) system software [1] proposes the following scheme: using the logical to the physical address mapping table, input data is written to an empty physical location to which no data have ever been written before, and the mapping table is updated as a result of the newly changed logical/physical address mapping. This works even if a physical address location mapped to a logical address

has already been written. As a result, one block is shielded against erase for each overwrite.

Existing FTL systems can be categorized into three groups based on the level of the mapping unit: page-level, block-level, and hybrid mappings. A logical page number (LPN) can be converted into a physical page number in page-level FTLs, as the name suggests (PPN). In other words, page-level FTLs have a significantly larger mapping table than any other type since they must keep a mapping entry for each logical page. In contrast, a logical block number (LBN) and an in-block offset are first separated from an LPN in block-level FTLs. The target page is then located using a search algorithm once the LBN has been converted to a physical block number (PBN). It is clear that block-level FTL mapping tables are fairly compact and can be conveniently stored in SRAM. By separating the flash memory into a data block area (DBA) and a log block area, hybrid mapping strategies attempt to achieve the flexibility of page-level FTLs while keeping the mapping table reasonably short and comparable to the block-level solutions (LBA).

Researchers have so far put out a number of implementation strategies for FTL. However, there are a variety of problems with each of these designs. The first attempt to move the traditional NOR-based FTL to NAND-type flash storage, eliminating the replacement page part, is made by DFTL (Demand-based FTL) [2], a page-level FTL technique. Although effective, this approach has a severe reliability issue since if the system fails, all updated data in the SRAM will be destroyed. Until the system returns to a consistent condition, spare portions of all data pages must be scanned. Therefore, DFTL is not appropriate in situations where flash memory is viewed as a durable and trustworthy storage option. The first hybrid FTL technique is called BAST (Block-Associative Sector Translation), and it was proposed in [3]. BAST restricts the overall number of replacement blocks to keep this table short enough to exist in the SRAM (also known as log blocks). Given that SRAM is far faster than flash memory, it is obvious that BAST has good read performance. But BAST struggles with random overwrite patterns, which could lead to a block thrashing issue.

Although the approaches mentioned above maximize the usage capacities of the devices, the challenges they produce substantially impede the development of effective implementations. The proposed algorithms both cause flash overhead and use very complex techniques, especially for write operations. In this thesis, we present a powerful technique called RFTL based on a random number generator. In the event that the previous page has already been written, the concept is based on using a random number generator to determine the new physical page number.

The contributions of this thesis are as follows:

- By using a random number generator-based technique instead of complex techniques such as least recently used or hot-cold data detection in write operations, the same goal was achieved with an uncomplicated algorithm.
- Unlike other algorithms, it was sufficient to allocate one flash memory page to run the entire algorithm.

The thesis is constructed as follows. Chapter 2 summarizes the background of the thesis topic. Chapter 3 gives general information about flash translation layers. Chapter 4 describes the approach of the RFTL algorithm for different operations of flash memory. Chapter 5 describes the RTL blocks of the design. Test setup and the test results of the design are explained in Chapter 6. Chapter 7 concludes by giving a summary of what has been achieved.

2. BACKGROUND

In this chapter, some important concepts related to this thesis are explained as a background for the next chapters.

2.1. Memory Storage Architectures

Considering storage options, although solid-state technologies, which were proposed in the past, such as EEPROM, are convenient from the point of size and data rates, they have limitations in the sense of capacity, speed, and power. The massive increase in the required capacity and device lifetime resulted in the failure of EEPROM devices as long-term storage devices. Despite being convenient in terms of capacity, rotating magnetic media is unable to give a straight performance in the sense of physical robustness and power consumption. In 1980, upon the invention of a new solid-state device, which is called Flash Memory, Toshiba offered new opportunities to improve data persistence for storage devices thanks to the characteristics of this new device, such as larger storage capacity and long-term storage stability. Flash memory also provided acceptable bandwidth and latency. There were two types of flash memories in the market, which were NOR flash and NAND flash. NOR flash was quickly able to replace EEPROM memory in embedded systems due to the possibility to be written at the byte level [4]. Although NAND flash has been applied in countless applications such as mobile handsets and primary storage solutions, NOR flash is still preferred widely in embedded systems [5]. This is because NOR flash provides lower read latency and higher data integrity.

Due to a number of problems, including programming mistakes and power problems, embedded systems don't always function properly. Devices need persistent memory storage in order to overcome these problems. Rotating magnetic storage is not a convenient solution for embedded systems because they are limited in terms of bandwidth, robustness and physical size [6]. Therefore, embedded system designers have

been in search of alternative solutions for re-writable and non-volatile memory technologies. In the following subsections, these alternatives are examined.

2.1.1. SRAM Technology

SRAM is based on the metal-oxide-semiconductor field effect transistor, which is utilized in practically every modern computing platform. The MOSFET is a field effect transistor device where the conductivity of the device is determined by a control signal. The devices are classified into two types Depletion mode MOSFET and Enhancement type MOSFET. Memory cells are built on the basis of Enhancement type MOSFET, which does not allow current to flow through the device due to a very high resistance caused by the low control signal. This operation mode is observed when the device is in the off state [7]. The characteristic construction of an n-type MOSFET is shown in Figure 2.1. An n-type MOSFET is composed of an n-type semiconductor substrate, which is also called bulk (B), and two p-doped regions named source (S) and drain (D). A current flow occurs from the source to drain in the presence of a positive voltage at the gate (G). Between the gate and the substrate, a high dielectric insulating layer is placed. As shown in Figure 2.2a, when there is no voltage applied to the gate of the device, current does not flow through the substrate. On the other hand, the application of a voltage to the gate results in an electric field between the substrate and the gate layer as shown in Figure 2.2b. Due to the electric field, a depletion region is created under the gate and insulating layer. The bias voltage at the gate and the size of the depletion region are interrelated. When the voltage difference between the gate and the source reaches a certain threshold value, the depletion region creates a conduction channel between the source and the drain. After the creation of the conduction channel, an increase in the applied voltage will allow more current to pass through the channel. Despite the analog characteristics of the MOSFET that comes from the controlling of the current between the source and the drain with the bias voltage applied to the gate, the device can also be used as an electronically controlled switch because of the features such as high efficiency and the ability to handle a large amount of current.

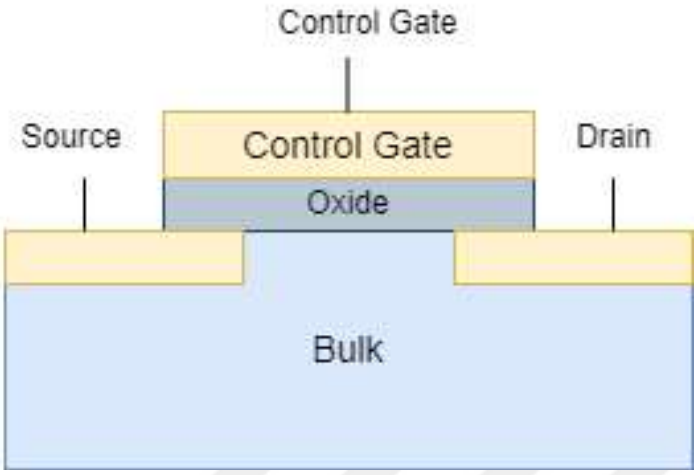

Figure 2.1. The Structure of a MOSFET.

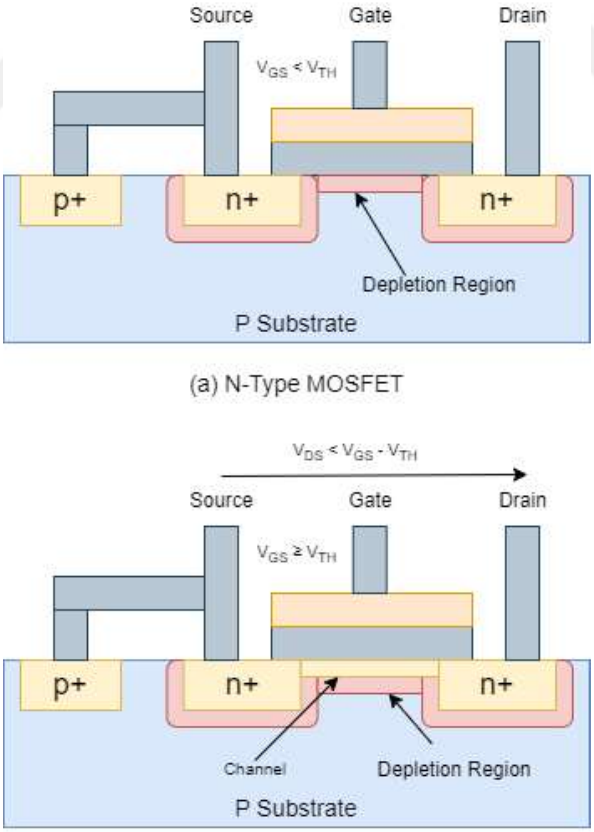

Figure 2.2. Conduction Channel Formation of N-Type MOSFET.

Static random access memory (SRAM) is a type of memory that uses a flip-flop memory cell to store a bit of data, and its structure is shown in Figure 2.3. The SRAM cell is composed of a pair of cross-coupled inverters that keeps the state of the cell, and a pair of access transistors that are responsible for read and write operations. In order to carry out the write operation, firstly the desired value and its complement are given to bitline (BL) and bitline-not (BL-not), correspondingly. Then, the word line (WL) is raised, and the cross-coupled inverters hold the new data. For reading the state of the cell, both bitlines are driven to high (logic 1), and then the wordline is raised, so that the access transistors M5 and M6 are enabled. This results in a slight drop in one bitline BL voltage. As a result, there will be a small voltage difference between BL and BL-not lines. A sense amplifier is utilized to recognize, which line has the higher voltage in order to determine the stored logic value (0 or 1) [8]. Although it has a power-efficient structure, SRAM is a volatile memory, which means that when the power is removed, data is lost. Therefore, researchers have turned their attention to non-volatile RAM (NVRAM) that has the ability to keep the data even if the power is removed.

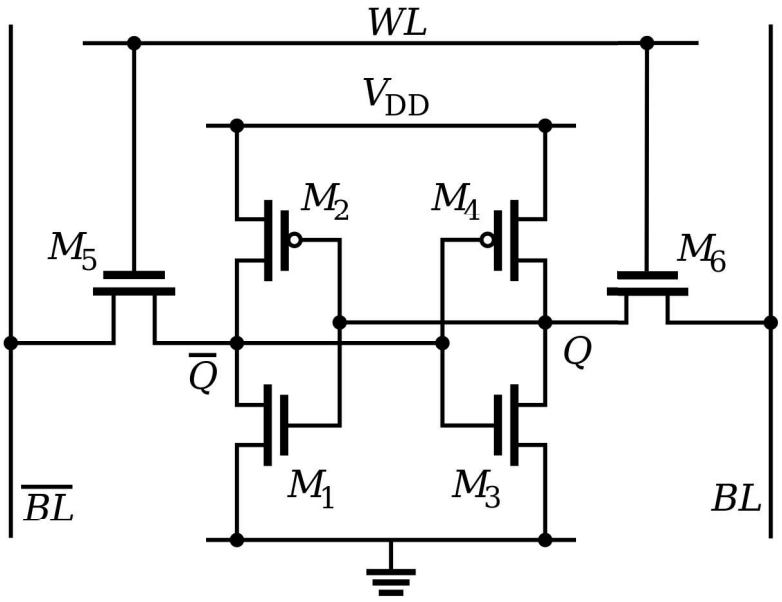

Figure 2.3. SRAM Memory Cell.

2.1.2. EEPROM

Non-volatile random-access-memory (NVRAM) is a memory device whose read and write characteristics show similarity to other RAM devices. However, NVRAM devices exhibit different data persistence characteristics, which means that the data that is held in the device is not immediately lost even after the power is turned off. Electrically erasable programmable read-only memory (EEPROM) is the first true non-volatile, readable, and programmable memory storage.

Arrays of floating-gate transistors make up EEPROMs. In 1967, Kahng and Sze originally suggested the floating-gate transistor as a component appropriate for creating a memory cell [9]. The device's design is very similar to that of a MOSFET, but it adds a fourth gate, known as the Floating Gate (FG) to the structure, which is placed between the control gate and the channel substrate as shown in Figure 2.4a. It serves as a barrier between the control gate and the channel substrate and is electrically separated from every other component of the circuit. Figure 2.4b shows the difference between the schematics of an ordinary MOSFET and a floating-gate transistor, which is an additional line attached to the image between the control gate and substrate. The floating-gate transistor has the unusual capacity to keep the charge for lengthy periods of time since it is electrically separated from other components of the device but will eventually lose the charge and the data contained in the device [10]. The electric field from the control gate, which is typically capacitively connected to the channel substrate, will be blocked by a charge when it is existing on the floating gate, inhibiting the development of the conduction channel. The floating gate's charge level modifies the threshold voltage that must be induced on the control gate in order to form a conduction channel, hence it is a non-binary effect, which is unfortunate. This trait is utilized to create the fundamental component of an NVRAM memory cell. Considering Figure 2.4c, the bit line will not detect current flow, which is regarded as a logic "0" if the charge is maintained in the floating gate as depicted in Figure 2.5a when the wordline is enabled to read from the cell.

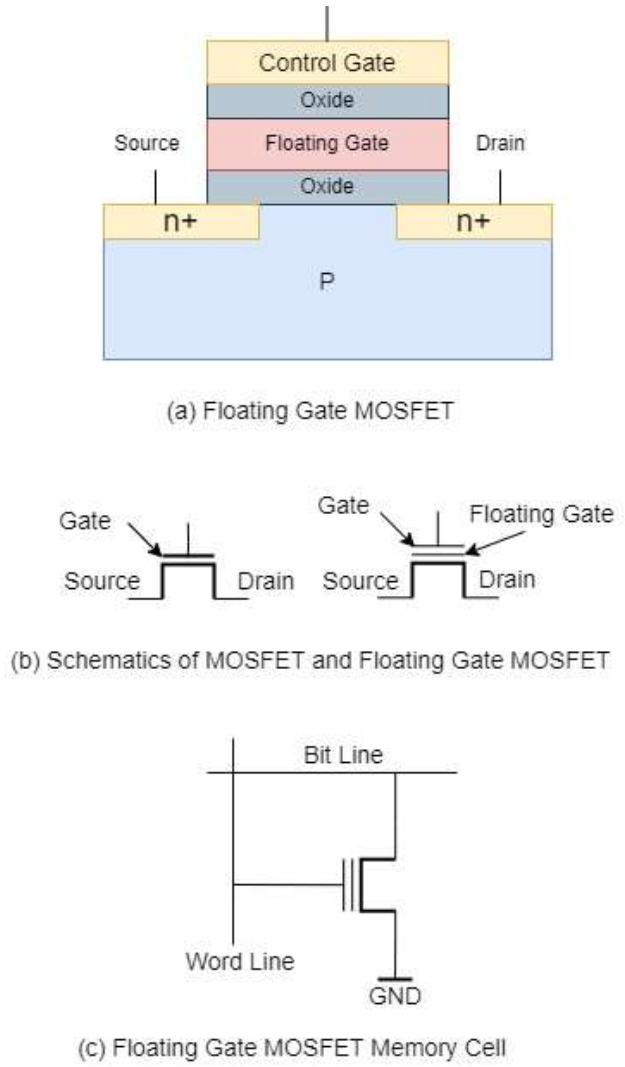

Figure 2.4. Floating-Gate Transistor Configurations.

This is because the floating gate is shielding the electric field from the control gate to the channel substrate. A conduction channel will develop and allow current to flow, which is interpreted as a logic "1" if there is no charge existing as depicted in Figure 2.5b when a voltage is created at the control gate [11]. There are technical difficulties with write and erase activities on the device, because charges must be added to or eliminated from the electrically isolated floating gate, which is the key distinction between non-volatile technologies that use a floating-gate transistor and other technologies.

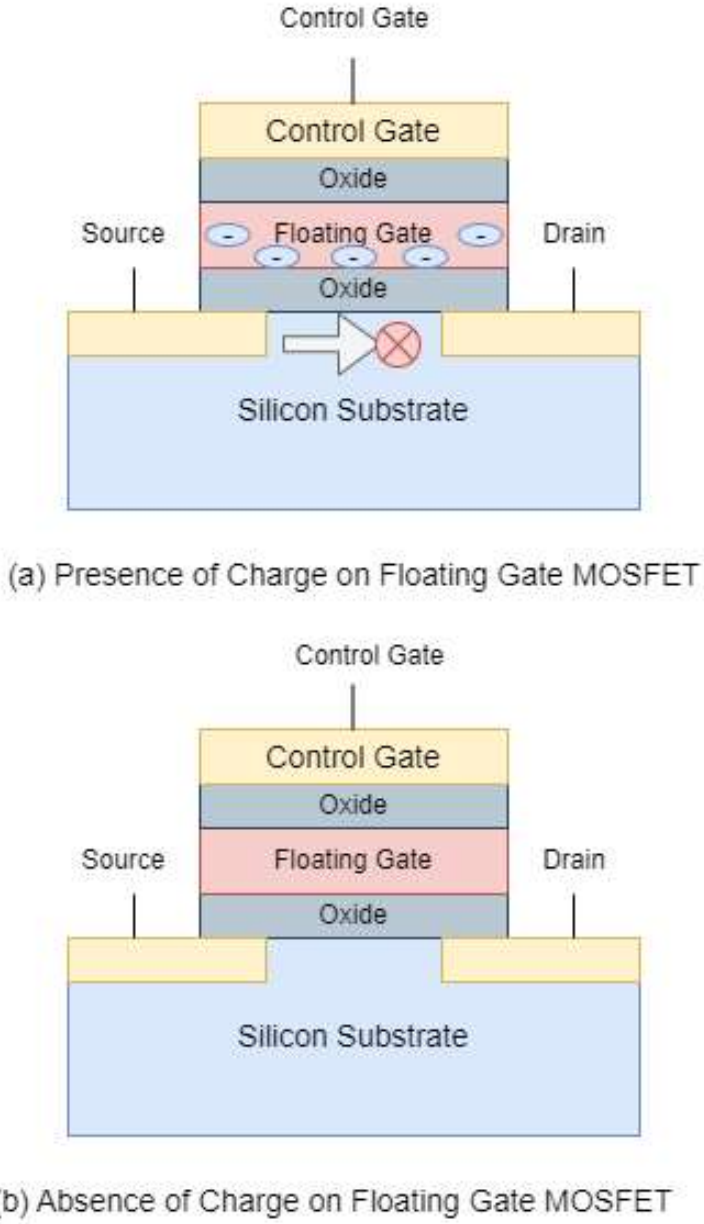

Figure 2.5. Impact of Charge on a Floating-Gate Transistor.

Fowler Nordheim (FN) tunneling, which involves applying a high electric field between the source and the control gate, is used in the floating-gate transistor structure to erase data [11]. As a result, a quantum mechanical tunnel is created through the oxide layer, allowing the removal of the confined electrons in the floating gate. This

technique has the benefit of allowing for huge tunneling currents while maintaining the insulator's dielectric characteristics. The required induced voltage differential is of the order of 18 V and cannot often be produced by the low supply voltages typically employed with memory devices. A considerable source of power is consumed in the overall functioning of any floating-gate transistor device when the voltage levels are produced by on-chip charge pumps. The buildup of charge over time in the electrically isolated floating gate is a serious drawback for all floating-gate transistor devices. Charge accumulates on the floating gate due to repeated erase cycles over time, which affects the forward threshold voltage needed at the gate and, ultimately, the device's capacity to conduct current. The device will ultimately get caught in a non-conducting status as a consequence.

The floating-gate transistor is the cornerstone of EEPROM technology. It was a significant advancement in data persistence technology, even if its structure and high voltage requirements for erasing meant that it still had capacity restrictions. In EEPROMS, to modify data in memory, a write operation must be followed by an erase operation in two steps. All floating-gate transistor-based memory requires the target region to be erased first before it can be written.

2.1.3. Flash Memory

Dr. Fuji Masuoka created flash memory in 1980, which is a sort of electronically erasable and reprogrammable memory. In comparison to existing EEPROM technologies, it is claimed to use less power, be shock-resistant, have a small footprint, and have data persistence periods longer than 10 years. When compared to other forms of solid-state devices, it offers greater capacity and speed as well as lower energy consumption, making it especially appealing for energy-constrained devices like embedded microprocessor systems and wireless sensor nodes. It makes use of the floating-gate transistor as the core component of a memory cell, much like EEPROM technologies.

NOR and NAND configurations are the two different types of flash that are accessible. Although they both have similar traits, each type's physical implementation is extremely different and has a particular performance benefit. NOR flash was first available with one-byte read-and-write units when it was first marketed by Intel in 1988. Because of the extra circuitry needed to read a cell's status, NOR flash memory cells are 2.5 times bigger than NAND ones [4]. NOR was initially rejected as a viable option for read/write storage while being regarded as acceptable and well-built for applications that demand random access to data because of its larger cell size. The improvement in NOR flash technology made it a more suitable memory storage choice. With millions of embedded devices using it, NOR flash is the most widely used format in embedded systems. The most prevalent format is NAND flash, which is distinguished by quick access for sequential byte access as well as high density and endurance. However, because of the physical cell interconnects and architecture, which increases operational complexity, it shows high startup latency and can be vulnerable to bit mistakes. Only pages are available for accessing data, which restricts the kinds of read and write operations. Typically, NAND storage is read in parallel, necessitating a high pin count dedication from the host processor, making it inappropriate for tiny devices with few pins.

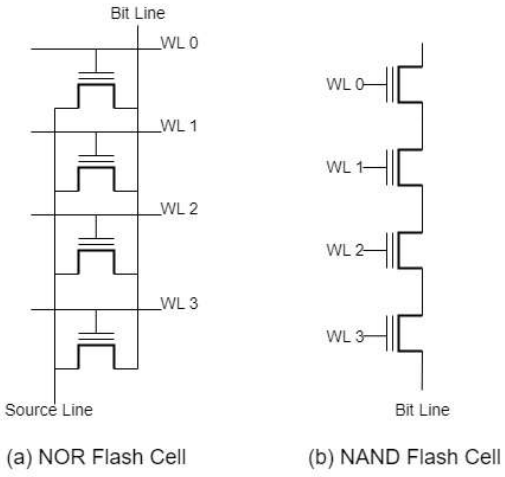

Figure 2.6. NOR and NAND Flash Cell Allignment.

At the most basic level, flash memory is made up of memory units called cells that will encode one or more bits of data depending on the fabrication method employed. A cell that has been erased will have the logic state of a "1" because of the electrical structure of flash memory. The cell value will be a logic "0" when it is written [12]. Data can be encoded as a byte thanks to the matrix's wording line for addressing and bit line for sensing connections between the cells. The most notable distinction between the NOR and NAND architectures is how a cell's state is decided, as seen in Figure 2.6. Each matrix component in the NOR (Figure 2.6a) design has a bit line connected to the drain and a control gate attached to the word line. Because of this, the matrix can target a single storage array element without disrupting any other parts. With concerns to the word line utilized to enable the control gate of the element or elements being read, NAND memory has a similar layout. In contrast to NOR memory, NAND flash has memory cells with source and drain connections that are linked in a domino chain (Figure 2.6b), with the source of one gate connected to the drain of the following [13].

2.1.3.1. Read Operation. The process used to determine a NOR cell's state is considerably different from the one used to find a NAND cell's state, which results in a significant size difference between the two topologies. In the NOR architecture, the relevant word line for the cell being investigated is activated in order to read the cell. A current in the order of amps will flow if the floating gate is not charged. A current-to-voltage converter transforms the bit line's output into the corresponding voltage by converting the flow of current. This leads in a system that can use a system that can utilize a well-understood voltage comparator circuit known as a sense amplifier, which is done to simplify measuring methodologies [14]. The value is compared to the reference cell's output, which will use a differential comparison method to evaluate the two voltage levels and produce a voltage that corresponds to the data that is maintained in the cell. This method is reliable and quick, with read times on current NOR flash memories being in the range of 10 to 20 nanoseconds.

Unlike NOR flash, which only allows one specific cell to be targeted, NAND architecture connects cells in series. It is necessary to distinguish the value of the cell being read from the values of the other cells in the chain. Each collection of cells includes a switching device that will link the group to the bit line being examined in addition to appropriately choosing the word line for a cell. In order for the target cell to generate a sense current, the remaining cells that aren't being read must be biased. To achieve this, the remaining cells in the group's word lines are driven over the conduction threshold voltage to the point where each cell will conduct regardless of the charge level on the floating gate, essentially converting the devices into pass-through transistors [15].

Both approaches have their limitations. Although the NAND flash read architecture has a lower physical dimension, it has some timing disadvantages. The bit line pre-charge normally lasts 2 to 6 microseconds, while the evaluation phase typically lasts between 5 and 10 microseconds in an effort to be mindful of peak current capacity.

2.1.3.2. Program Operation. NAND and NOR flash are programmed using separate methods but share the same core memory cell, which accounts for an essentially distinct mode of operation. Fowler-Nordheim tunneling, a quantum-effect electron tunnel produced in the presence of a strong electric field, is used to program NAND memory. When an electric field is produced between the substrate and control gate to get through the insulating oxide's potential barrier, a channel tunnel is formed. As shown in Figure 2.7, this makes it possible for electrons to enter the floating gate [16]. The strength of the generated electric field directly relates to the amount of charge that can be produced onto the floating gate. In order to provide the strong electric field needed to improve programming performance, a high voltage source is therefore required, which presents a challenge in terms of available power. The approach has certain drawbacks in that it takes a lot longer to program than NOR flash, but overall, because of the parallel architecture used, it achieves a greater programming rate per second. Additionally, NAND flash memory will experience high voltage-induced tunnel

oxide deterioration, which reduces the device's overall endurance. Also on the positive side, the method is preferable when programming a large number of cells because it requires incredibly low currents.

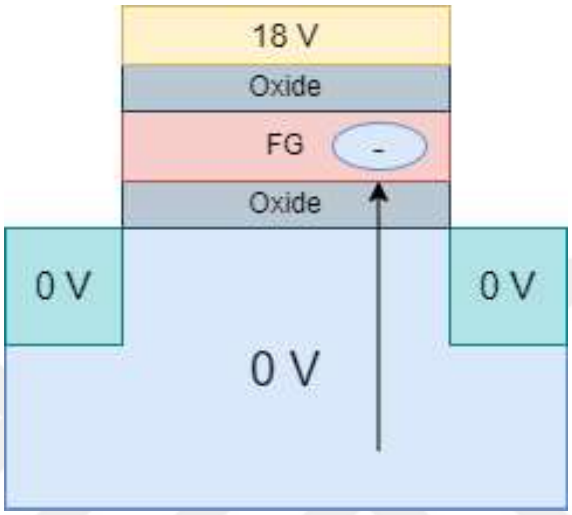

Figure 2.7. NAND Flash Memory Program Operation.

NOR flash is programmed by Channel Hot Electron (CHE) injection, which generates an electric field by a differential voltage between the source and drain, as opposed to NAND flash. Through a collision mechanism, the electric field gives electrons sufficient energy so that they can easily travel through the oxide layer. As shown in Figure 2.8, the control gate is then given a voltage to create a traversal electric field, which enables electrons to move to the floating gate [14]. As charge builds up in the floating gate, affecting its electric potential, the new charge will be less attracted, and programming will finally come to an end. This is one benefit of the channel hot injection programming method. Channel hot injection is much faster than per-cell programming in NAND storage, but it uses a lot more current. When programming several cells simultaneously, it can put a strain on the system's capabilities. The total current that is available and, consequently, the total number of cells that can be programmed in parallel, may be constrained by design constraints [17].

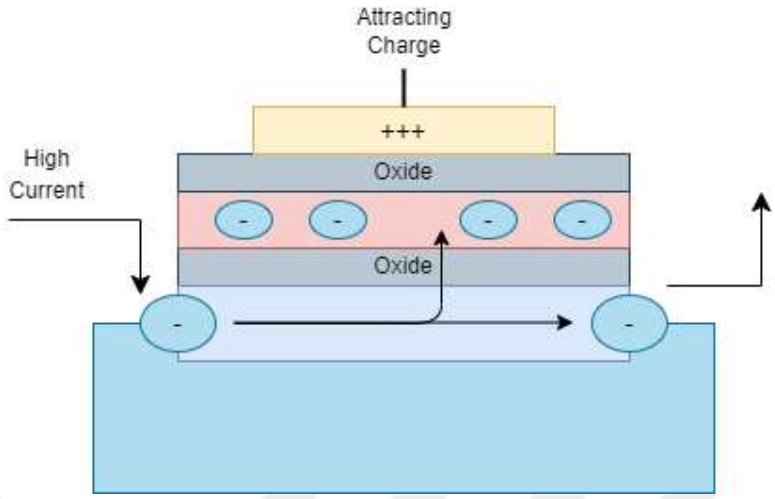

Figure 2.8. NOR Flash Memory Program Operation.

Although NAND and NOR flash is programmed in different ways, both architectures use a process known as a program and verify. It is used to confirm that the proper charge has been induced onto the floating gate, producing a particular threshold voltage that allows a cell to be thought of as programming. The target cells will be reprogrammed successively until they reach the required threshold voltages. The system will produce an error message indicating that the programming procedure has failed if the required threshold voltage cannot be obtained after a predetermined number of tries.

2.1.3.3. Erase Operation. The same method is used to erase NAND and NOR flash. Flash memory cells, unlike other floating-gate transistor memory systems, must be erased in blocks because of the structural links between them. Space conservation and reduced production costs were the main motivations behind this. Using Fowler-Nordheim tunneling, where a high voltage from the substrate to the control gate is applied across the oxide layer, the erase process is carried out as shown in Figure 2.9. A high voltage erase pulse is applied to the entire block to achieve this [14]. As a result, a Fowler-Nordheim tunnel can form between the gate and source, enabling the

removal of electrons from the floating gate. Channel construction is now possible when the control gate is activated thanks to the elimination of the charge accumulation in the floating gate.

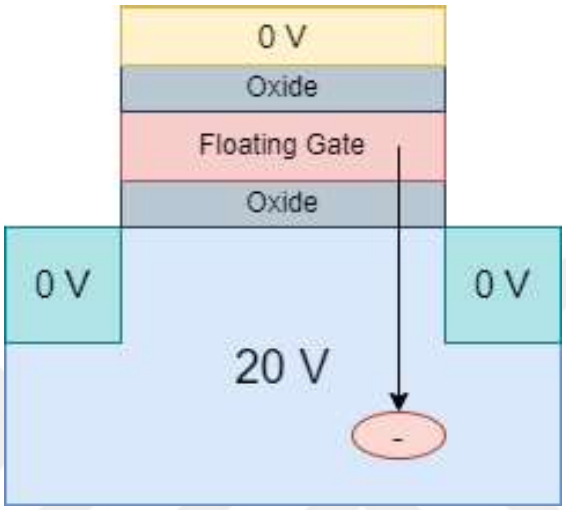

Figure 2.9. Flash Memory Erase Operation.

The erase processes are responsible for one of flash memory's primary constraints. The floating gate will gradually accumulate residual charge from the tunnel oxide's breakdown, which will confine the memory cell in a single state over time [18]. Another issue caused by the charge accumulation is that as the device ages, the erase time per cell lengthens, decreasing performance over the course of the device's lifetime. Memory devices actually use sophisticated erase algorithms to lessen this impact. In order to confirm that a block has been properly erased, both NAND and NOR memories need an erase verification algorithm. However, due to the architectural distinctions between NAND and NOR memories, the erase method uses different procedures. Because of the increased complexity and number of phases in the erase algorithm, NOR flash erase timings are three orders of magnitude longer than those of NAND memories, resulting in a significant variation in erase times between the two types of memory. This erase time discrepancy may restrict the kind of apps that can use a particular memory.

Due to the techniques employed for programming and erasing, flash memory presents basic difficulties that restrict the device's functionality. Long-term data retention may be impacted by oxidative stress, excessive erasure, and programming. As charge accumulates in the oxide layer of the cell over time, the required erase and program voltages increase along with the frequency of device cycling. As a result, tunnel oxide degradation occurs [19]. The voltage window between the erased and program states gets smaller in NOR flash due to oxide deterioration, which is known as the "erase threshold window closer". In reality, the internal controller makes allowances for this, which causes the programming and erase times to lengthen as the device gets older.

A single cell can be utilized to encode more than one state of data in an attempt to enhance the storage capacity per unit area of flash memory cells without the necessity to reduce the size of the memory cell. A cell is referred to as a Single-Level-Cell (SLC) if it only encodes one bit of data, and a Multi-Level-Cell if it contains many bits of data (MLC). By changing the charge level on the floating gate to show different bit states, an MLC enhances the density of a cell. The read circuitry becomes far more sophisticated as a result, as the system must now interpret the value in order to ascertain the value encoded by the cell, in addition to determining whether a current flow is present or not. A cell may now store up to 8 distinct states using this approach, allowing for the storage of 3-bits of data. Although MLC flash can now have a better density and cheaper cost per bit thanks to this method, there are still costs involved. Due to both increased read times and oxide deterioration, MLC devices have much lower lifetimes. SLC NAND flash has a significantly lower bit error rate than MLC NAND flash and can read and write data up to three times quicker than MLC NAND flash. Because of this, SLC technology devices last longer, are more reliable, and operate faster than MLC devices, but their cost per density is higher.

Small microprocessors may experience memory management issues due to the fact that erase tasks are slower than other processes on the device and must take place in physical blocks or sectors. Before the device starts to malfunction, each page in the

device has a finite amount of erase cycles (usually 100,000). To increase the device's number of write operations, a wear leveling algorithm can be employed to spread the cost of the write/erase cycle across all pages [20]. Other devices have adjacent block erase/write constraints in place, which reduces the amount of time between erasures and necessitates frequent consecutive block erasures.

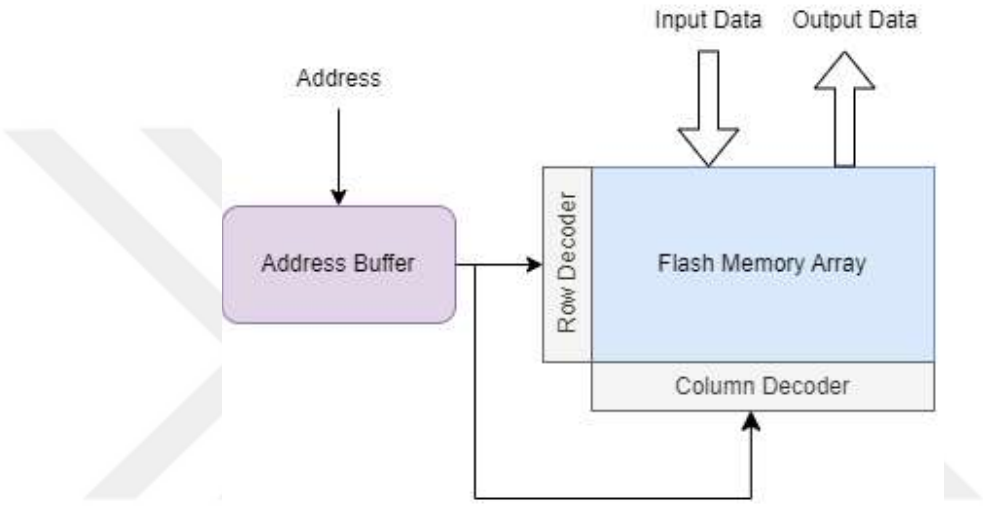

Figure 2.10. Flash Memory Functional Block Diagram.

2.1.3.4. Flash Memory Structure. Flash memory organizes cells into pages, which are the smallest read/write units. The minimum unit of data that may be stored or read in a flash system is a page, which is a physical collection of bytes in memory that have the same page address [21]. The general organizational structure of flash memory is depicted in Figure 2.10. Because certain manufacturers use the term "sector" differently, confusion is frequently caused when authors also refer to the smallest unit of readable or writable data as a sector. File systems also use the term "sector" to describe the smallest unit of transmutable data on rotating magnetic disks, which causes additional misunderstanding. In this study, the smallest transmutable unit of data in flash memory will be referred to as a page. A page has limited supplementary memory that can be utilized to hold meta-data or error correction codes (ECC) in addition to the

primary memory space, which uses a 2^n base addressing scheme. The out-of-bounds area (OOB) is what this region is known as, and data is normally not stored there. When the OOB region is divided up into smaller subsections, many NAND flash chips' OOB area can enable partial rewrites. This permits rewriting the OOB section of a single page a set number of times before needing to erase it. This feature is known as partial-page programming (PPP) [22].

A single NAND flash page can support a finite range of write operations to separate sub-divisions before the page must be deleted. This is possible thanks to partial-page programming. While portions of a page can be written separately, partial-page programming prevents previously written areas from being changed. There are only certain sections, and not all NAND flash devices have this feature.

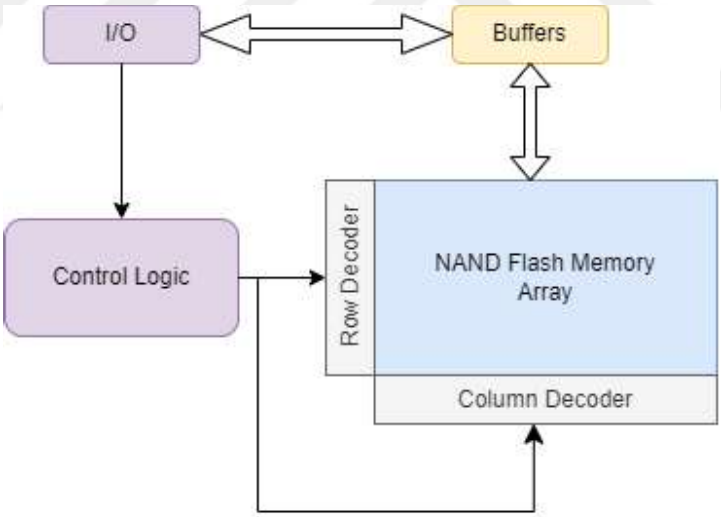

Figure 2.11. NAND Flash Memory Architecture.

Blocks are collections of pages. A block can be identified by its physical block number and is the smallest erasable unit. This means that if you want to delete the contents of a certain single page, you must also delete the block that the page is a part of. A block can contain several pages based on the particular implementation, and the

number of pages that make up a block is determined by the producer's standards for the device. As a consequence, the block's size varies depending on the situation. This rule does have several notable exceptions. A device can also be deleted at the page level, but doing so comes at a significantly higher energy and time cost.

Under usual conditions, it is impossible to read or write data directly from flash memory. An onboard SRAM buffer that shadows a page of flash is used for every transmitting data as shown in Figure 2.11. Because of the page alignment configuration, the host needs the flash memory to relocate the requested page into the page buffer in order to access data. The host might ask for the data to be transmitted from the device after the page has been correctly put into the buffer. In contrast, because of the erase-before-write requirement, when a page has to be changed the entire page must be loaded into the buffer, changed, and then written back into an erased page.

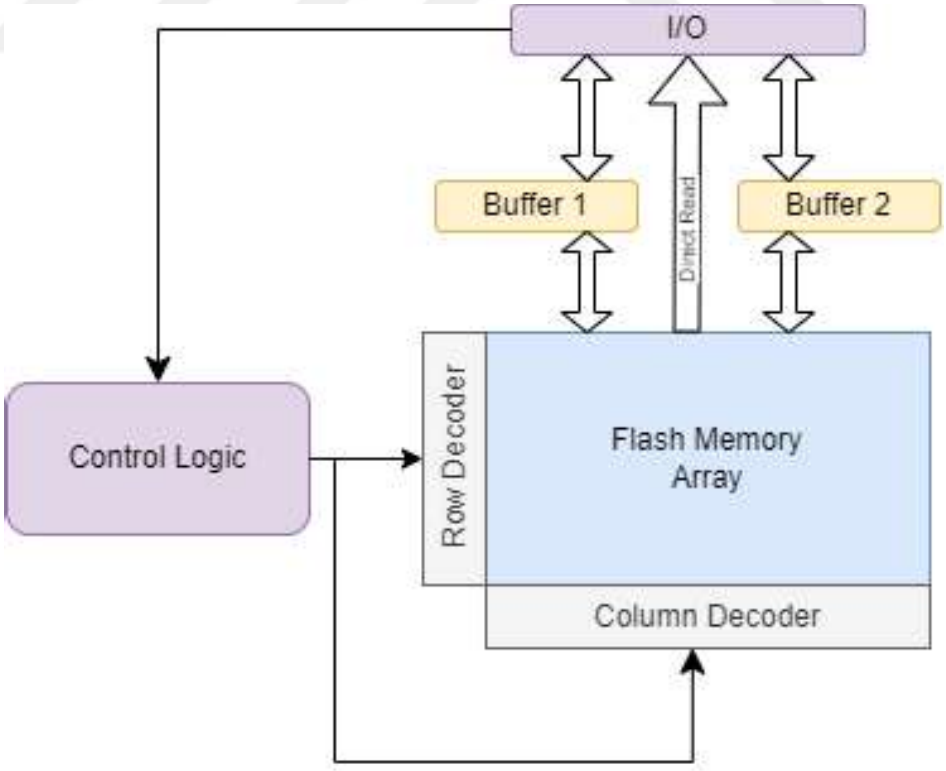

Figure 2.12. Dual Buffer Flash Memory Architecture.

Read and write operations are complicated and slowed down by the restriction on data transfer into and out of the buffer. It was proposed to use dual buffer NOR-based flash memory, which adds a second buffer to the design as shown in Figure 2.12. By enabling concurrent, non-overlapping functions, the delay is reduced. One of the buffers, for instance, could receive a page from the main memory while the host reads real-time data from the other. Reading information from the second buffer while simultaneously writing pages in one buffer is possible. The NOR arrangement of the architecture additionally adds a direct read option. This unusual procedure gives a significant speed advantage for this memory by allowing a data stream to be immediately read from the flash memory without disrupting the data stored in the buffers.

2.1.3.5. Restrictions of Flash Memory Structure. Flash memory has a limited number of erase/write operations and does not provide in-place upgrades while being inexpensive and generally reliable. In an attempt to evenly distribute erases and writes throughout the entire device, these problems are attempted to be mitigated using write normalization and wear leveling. Flash memory usage without wear leveling or additional erase/write normalizing techniques can hasten device failure. Although there are many flash management systems that offer cutting-edge and effective ways to store data, their practical use is limited by the fact that they all rely heavily on the RAM or EEPROM of the sensor processor. Moreover, because the methods are designed for parallel NAND memory, serial NOR flash will not benefit from them from the point of performance. Cost, performance, and usability trade-offs exist with all technologies. Flash memory pages gradually lose their ability to securely encode data since they deteriorate with repeated erases and have a limited lifetime. The host system is responsible for monitoring data consistency or erase cycles to prepare for ultimate failure. To ensure consistency, certain devices provide read-back after write verification. It is tried to write and delete pages evenly across the device so that a device degrades consistently. The wear leveling approach generally guarantees that data writes are distributed uniformly over the device [23].

Flash memory suffers from uneven read and write costs as well as wearability. Write times will usually be several orders of magnitude longer than read times. This is distinct from other storage media from the standpoint of implementation, such as rotating media, which often has symmetric performance. The fact that a unit of data cannot be permuted in place is another significant distinction from rotating media. The destination cell must be erased first before data can be stored to flash memory. The term "erase-before-write limitation" is used to describe this. The erase operation is more costly (both time and energy) than other functions. Moreover, the majority of architectures call for data to be deleted in blocks, which increases organizational complexity.

3. FLASH TRANSLATION LAYERS

Flash translation layers are in charge of supplying a mapping strategy between logical data units in an application and physical data units on a device. A direct mapping between the logical and physical storage units may or may not exist, depending on the FTL's design. In reality, the FTL is made up of three parts. The FTL must also provide wear leveling in addition to the logical to physical mapping in order to guarantee uniform wear throughout the device, which will increase the number of write operations due to erase failure. The FTL also needs to handle Power Off Recovery (POR), as this is necessary for embedded systems because a sudden loss of power can corrupt the FTL [24]. The FTL must recover and guarantee that the data on the device is consistent and accessible upon restart. Most earlier efforts concentrated solely on more efficient storage by reducing erase operations and lengthening the number of write operations of the device. The volume of information sent between hosts as well as the energy used during operations have not been taken into account; both of these factors are crucial to take into account for devices with limited memory, pins, and power.

There are a few numbers of definitions that are commonly used in FTL-based systems. They are logical pages, free pages, invalid pages, and valid pages. A page that has been deleted and is empty is referred to as a free page. A page that has data on it but is no longer associated with an application, service, or active logical mapping is said to be invalid. A page that is actively mapped and logically connected to an application or service is referred to as a valid page. Old pages are not erased or used as they are moved through rewrites or wear leveling. The device will ultimately run out of free or erased pages if left unattended. During this procedure, pages that have been neglected are invalid. Data-containing pages are regarded as valid.

The FTL must keep track of each page's usage status (free, valid, or invalid) directly or indirectly in order to address the problem of growth in invalid pages through-

out the operation. A tracking technique marks the original page as invalid when data is relocated. The FTL will start the garbage collection procedure once it reaches a certain threshold for the number of invalid pages on the device. As a result, invalid pages will be deleted, becoming free pages once more. When garbage collecting is carried out, it's possible that valid pages and invalid pages are coexisting in the same block. The valid pages are relocated outside of the targeted block for erasure, a process known as compaction, in order to prevent data loss during erasure. The block erasure can continue after being transferred.

3.1. FTL Classifications

Prior efforts have mostly concentrated on three different types of FTLs for NAND flash memory and how to optimize these algorithms for that configuration [25]. Few viable FTL algorithms for NOR flash have been studied, and those that have failed to address important issues for limited devices. FTL designs share common design characteristics with either a completely associative cache, n-way associative cache, or a directed mapped cache. They are closely connected to memory cache methods. With the help of these models, FTLs can be broadly divided into three types: block level (direct mapped) mapping techniques, which have a lower degree of granularity than page level (fully associative) mapping techniques, and hybrid (n-way) mapping level techniques, which combine techniques from both page and block mapping schemes. The fully associative or page-level approaches, block-level mapping, and hybrid mapping strategies will all be covered in the following sections. Each method has trade-offs in terms of SRAM footprint, effectiveness, and durability. Performance is measured by minimizing erase operations and increasing block utilization prior to an erase. The cost of power and data transfer is not taken into account in strategies.

3.1.1. Page Level Translation Layers

A page level or fully associative translation layer, first introduced by Ban in 1995, is regarded as a simple algorithm [26]. Page mapping systems typically perform

better than alternative taxonomies in earlier works, but they have the biggest SRAM footprint because a mapping must be saved for each page. Nevertheless, it provides the highest device utilization performance. The mapping between logical and physical pages is one-to-one. Usually, a 2-tuple is used to hold this, with the logical page number serving as the primary key as it is shown in Figure 3.1.

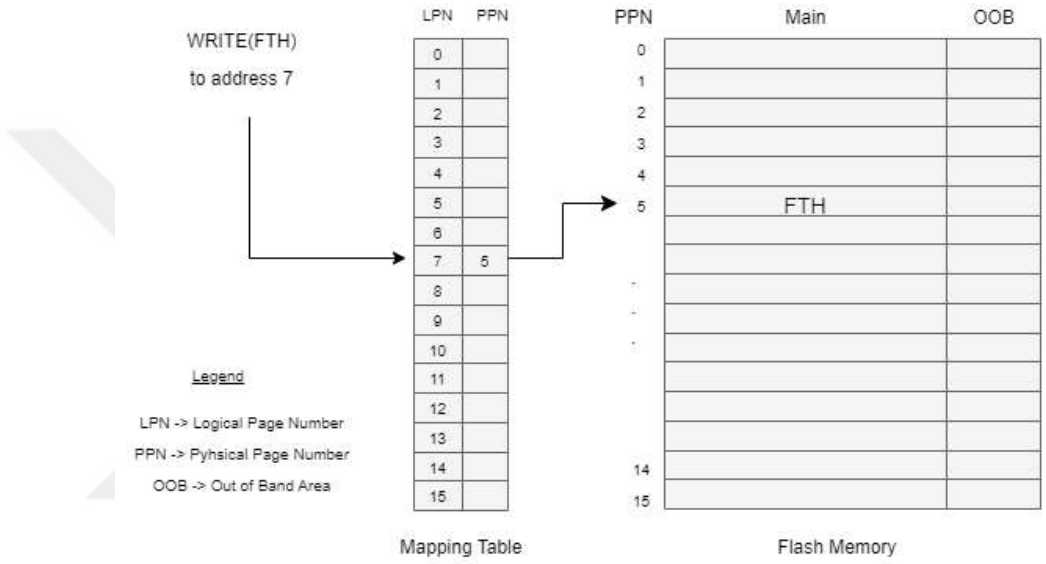

Figure 3.1. Page Mapping Scheme of FTL.

In this illustration, flash memory is made up of 16 pages, four pages of which are in each block. The blocks are divided into two categories: update (U) blocks and data (D) blocks. FTL uses data blocks to hold application data, whereas U blocks are utilized by the FTL for management [27]. Three D blocks and one U block are included in this illustration. The FTL will try to write A to logical page number 7 when a write instruction is given to it, such as write (7, A). To do this, the FTL will search the logical page 7 mapping record and determine that it corresponds to physical page 5. If physical page 5 is available, the FTL will write A to it. The FTL will choose a free page to write the data to if the physical page is invalid, update the logical to physical page mapping, and then flag physical page 5 as invalid.

The FTL must choose a victim block to erase if there are no available free pages in the system, which results in a management expense. The block may be chosen using some other victim block metric or the victim block with the most faulty pages [24]. The active pages will then be transferred to the U block by the FTL, together with the page that is now being written. Following that, the mapping table will be modified to represent the changes. The victim block will subsequently be deleted by the FTL, becoming the new U block.

Page-mapped FTLs must make sure that the mapping table is consistent after a power cut in order to provide recoverability (either planned or unplanned). The mapping table can be flushed to flash memory on a regular basis or each page's logical page numbers can be encoded in the out-of-band area. The FTL must sweep to find or recreate the table after a restart before it can be used.

N tuples of SRAM are required for flash memory with n pages. Both general-purpose computers and systems with limited memory may find this to be a challenging approach. Take the following scenario into consideration for a flash memory device with a 1 MB capacity and 512-byte pages. There are 2048 pages for this device that must be mapped into SRAM. This address space can be encoded using a short (2-byte) int. This results in a mapping table of 8,192 bytes using a 2-tuple of 2 short ints (one to encode the logical page and one to encode the physical page). Despite the fact that this table appears to be very small, the lack of SRAM on the majority of 8-bit microprocessors prevents it from being encoded.

The implementation of the page-mapped FTL must take into account two important factors. Since each physical page should have one mapping tuple in the translation table, the size of the mapping table might be a significant problem, especially for devices with limited memory. The table must be persistent, which necessitates its storage in non-volatile memory, adding to the difficulty of the situation. Because just a tiny portion of the table may potentially be in SRAM at any given moment, based on the nature of the access patterns, the memory limits can be eased. To achieve this,

caching techniques can be used to lessen address translation thrashing, which could affect system performance as a whole.

The paged mapped approach performs better than the block level and hybrid techniques. This is partly because page-mapped FTLs can postpone erases for as long as possible because of their flexible and unrestricted page insertion procedures. The plan makes sure an erase block is used up completely before being selected for erase. Furthermore, it is not necessary for data to be clustered into block areas that reduce the FTL's overhead; as long as pages are free and over the specified threshold, the system can function without having to call the garbage collector. Due to the negative connotations associated with performance limitations with SRAM-based mapping tables, little effort has been put toward page-level systems for NAND and NOR flash. These issues are covered by two works: DFTL [2] and LazyFTL [28]. Both compare the performance of page-mapped schemes to that of hybrid and block-level mapping strategies, underlining the fact that they perform better in terms of flash utilization, erasing, and wear.

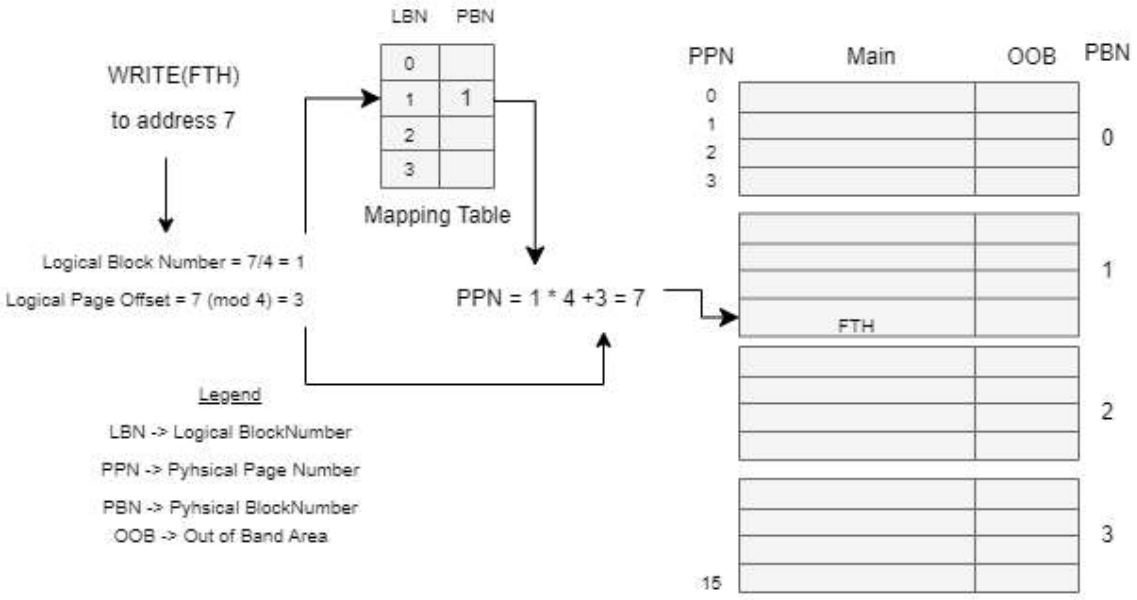

Figure 3.2. Block Mapping Scheme of FTL.

3.1.2. Block Level Translation Layers

The performance and footprint of the translation table are traded off in a block-mapped translation approach. Block-mapped strategies, which are by far the most widely used technique, organize pages into blocks that collimate with erase units on the physical flash memory device, in contrast to page-mapped strategies, which have a high level of granularity.

Pages are still accessible at the logical level with a block-level FTL, but the system groups them into logical blocks for write operations. Based on its offset within the block, each logical page in the logical block will translate to only one physical page in the physical block. Block-level translations are known as in-place systems because of the spatial page limits on where a page may live in a block. Block-mapped methods have a significant constraint as a result of flash's erase-before-write feature. When an individual page is modified using a block mapping approach, the entire block must first be deleted. The block (including valid and invalid pages), excluding the target page, will be copied to an erased block instead of being removed entirely, which can result in significant management costs.

Block mapping techniques, in contrast to the preceding approach, present m logical blocks in the mapping table, each of which can hold one to n logical pages. The logical block number serves as the main key for a 2-tuple mapping between the logical block number and the physical block number that makes up the record. Figure 3.2 depicts a block mapping FTL's general layout. A hash function whose most basic form specifies what logical block a page is located in is calculated as

$$\text{LBN} = [\text{LPN} \div \text{m}] \tag{3.1}$$

where m is the number of free blocks, LPN is the logical page number of the page to be read, and LBN is the logical block number to which a logical page will belong. The FTL mapping table stores the mapping between the logical block to the physical block number. The logical page's physical page number is calculated as

$$\text{PPN}_{offset} = \text{LPN} \pmod{\text{n}} \tag{3.2}$$

where n is the block size in pages and PPN is the physical page number within a specific block as an offset from the block's first page.

Take the scenario in Figure 3.2, where the write command write (7, A) is sent and the flash memory is divided into 4 blocks, each of which has 4 pages. A will be written to logical page 7 in an attempt by the operation. This can be done by first computing the logical block using Equation (3.1)

$$\text{LBN} = [\text{LPN} \div \text{m}] = [7 \div 4] = 1 \tag{3.3}$$

where logical block 1 membership for the logical page will be present. The FTL will use this value to identify the physical block in which the page will be placed. After that, the physical page offset will be determined utilizing Equation (3.2)

$$\text{PPN}_{offset} = \text{LPN} \pmod{\text{n}} = 7 \pmod{4} = 3 \tag{3.4}$$

where the block's logical page's offset is 3. The FTL will obtain the physical block number from the mapping table once the logical block number and physical page offset have been determined.

Block mapping can shrink the footprint of the mapping table in SRAM, but there are performance drawbacks if the system sends consecutive write commands to pages belonging to the same logical block. This will cause a block to thrash because more rewrites will be necessary. As a result, the system's overall performance would suffer because it will have to manage a copy and erase for each operation. The log-based block mapping techniques, which are outside the scope of this study, have been developed to address this performance issue. These schemes can be characterized as performance-boosting algorithms because their primary design goals include lowering thrashing and the erase cost overhead.

3.1.3. Hybrid Translation Layers

Both the page-level and block-level mapping strategies have drawbacks related to the size of the SRAM translation table or the restrictions of the erase-before-write algorithm. A hybrid translation technique that benefits from the performance advantages

of both block-level mapping and page-level mapping has been proposed as a solution to these problems. Similar to block mapping approaches, hybrid mapping locates a physical block that provides a desirable SRAM footprint for the translation table by using a logical to physical block mapping [3].

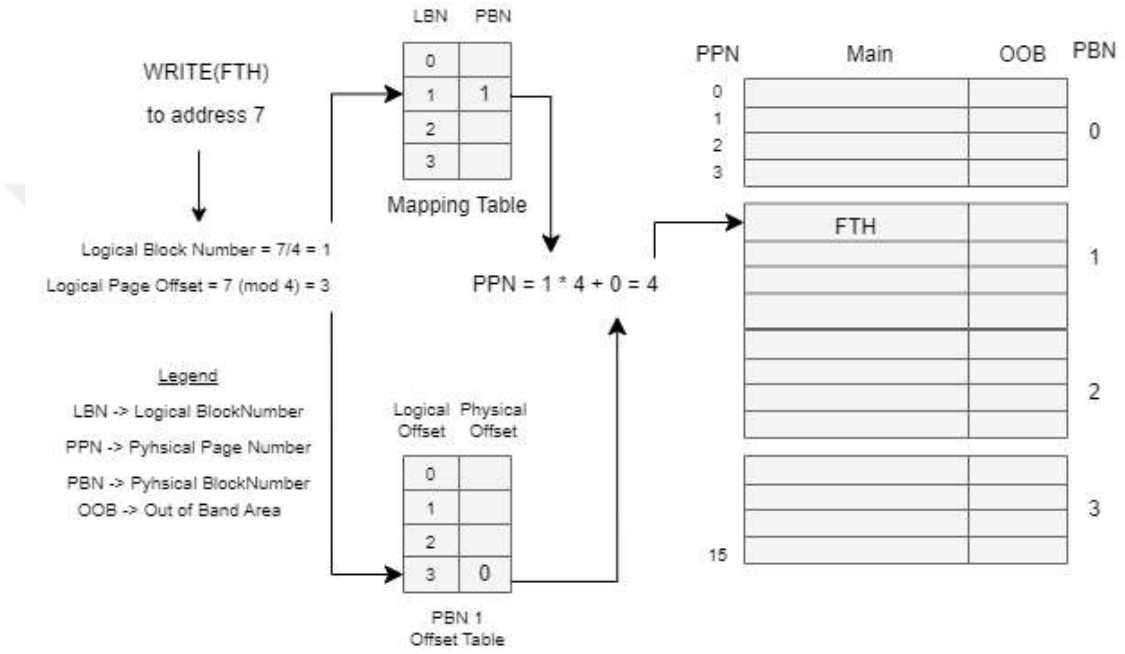

Figure 3.3. Hybrid Mapping Scheme of FTL.

The amount of pages in a block is determined by the manufacturer and can differ between devices, but is normally eight pages minimum. A block is made up of consecutive sets of pages that create a minimal-erase unit. n contrast to block mapping, hybrid mapping techniques do not require direct page mapping for a block. With hybrid schemes, any free page inside the physical block can be used to insert an intra-block page. A single page may exist multiple times in a given block, especially under large sequential rewrites, hence more complicated intra-block page mappings are needed. This helps to handle the erase-before-write-driven thrashing that block-mapped systems experience. Because page mappings must be maintained on a block

level, the technique adds complexity. In its most basic configuration, the FTL will use data stored in a page's OOB section to do a linear scan to identify the most recent version of a given page. This may work well for small block sizes, but as block sizes grow, it may become impossible to maintain.

3.2. FTL Related Works

Note that the algorithm that is proposed in this study is a page-level-based algorithm. Therefore, related FTL works, which are covered in this section are also page-level FTL algorithms. This is aimed so that a good comparison opportunity is provided.

3.2.1. DFTL: Demand Based Flash Translation Layer

Despite being an effective design, Gupta, Kim, and Urgaonkar [2] state that the page-level flash translation layer can produce a huge translation table that must be stored in SRAM. The mapping table for a 16 GB memory device will be roughly 32 MB in size. Their work is just a flash mapping interface and focuses on enterprise-level databases using NAND flash-based SSD technology. These levels of SRAM will not be available on highly memory-restricted systems. They offer a solution to this problem that involves selectively caching page-level address mappings in order to reduce the amount of data saved in SRAM. Although the approach significantly improves performance for enterprise-level systems using NAND flash, it is an impractical solution for devices with limited memory and bandwidth. They raise a valid point from the perspective of the memory-constrained method when they say that block-level mappings may cause high-level internal fragmentation in devices as well as performance deterioration from excessive garbage collection overheads. They contend that the proposed cache-based technique may work well for devices with limited memory because data is temporal in nature. Sadly, this aspect alone does not ensure that it is appropriate for serial NOR flash devices. This technique would result in severe page-mapping trashing over the serial data bus since more time would be spent on mapping table transfers, as

was previously mentioned with constrained SRAM sizes. Performance would suffer as a result of the page-mapping movement. They continue by claiming that because they are a page-level map scheme, they decrease operations like complete and partial block-level merges, which are troublesome sources of performance deterioration in block-level schemes, making their technique excessively advantageous. Although this may be the case, the system does not take power on recoverability, data and page level mapping consistency, or consistency into account. Although it might be an ideal method for enterprise-level systems, it is impractical to utilize in systems with limited resources.

3.2.2. CFTL: Convertable Flash Translation Layer

CFTL is a hybrid block mapping method that dynamically switches between write and read-optimized schemes while managing FTL mapping pages with an effective caching mechanism [29]. It functions primarily as a page-mapped FTL that is kept in flash memory. Eight cache blocks and page map tables based on the temporal and spatial locality of the data are stored in SRAM. A master mapping table that keeps track of all other mapping tables is likewise kept in SRAM. The authors contend that this is significant for the locality in both space and time, but there are no variable latency costs with flash memory; the only penalty is the cost of looking up translations in the FTL. Therefore, the overhead of transferring the lookup tables from flash memory will be the main factor in the FTL lookup. This causes a significant bottleneck for a device with limited bandwidth and ought to be avoided. With the CFTL architecture, each page also has a write counter that is kept in SRAM and used to distinguish between hot and cold pages. This significantly raises the overhead for both page re-allocation and data storage structure. The page-level mapping allows changes to be inserted on any page, which the authors claim leads to some slight gains. However, they neglect to mention how this affects operations for wear leveling and garbage collection. Additionally, they miss out on calculating the costs of page conversions when transitioning from page to block-level mapping. They have also neglected to explain how block-level access works, how they deal with consistency problems between flash memory tables and SRAM caches, problems with record integrity in general, and power-on recovery. Per-

formance comparisons to DFTL [2] are done and significant improvements are claimed, however, no testing information is given. While it might quicken FTL translations and lookups, it doesn't address any other fundamental problems with the FTL design.

4. RNG BASED FLASH TRANSLATION LAYER (RFTL)

RFTL is a page-level flash translation layer algorithm that is based on using a random number to handle address translation. The RFTL algorithm utilizes a mapping table in read and erase operations as other FTL algorithms. The architecture of the RFTL algorithm is shown in Figure 4.1.

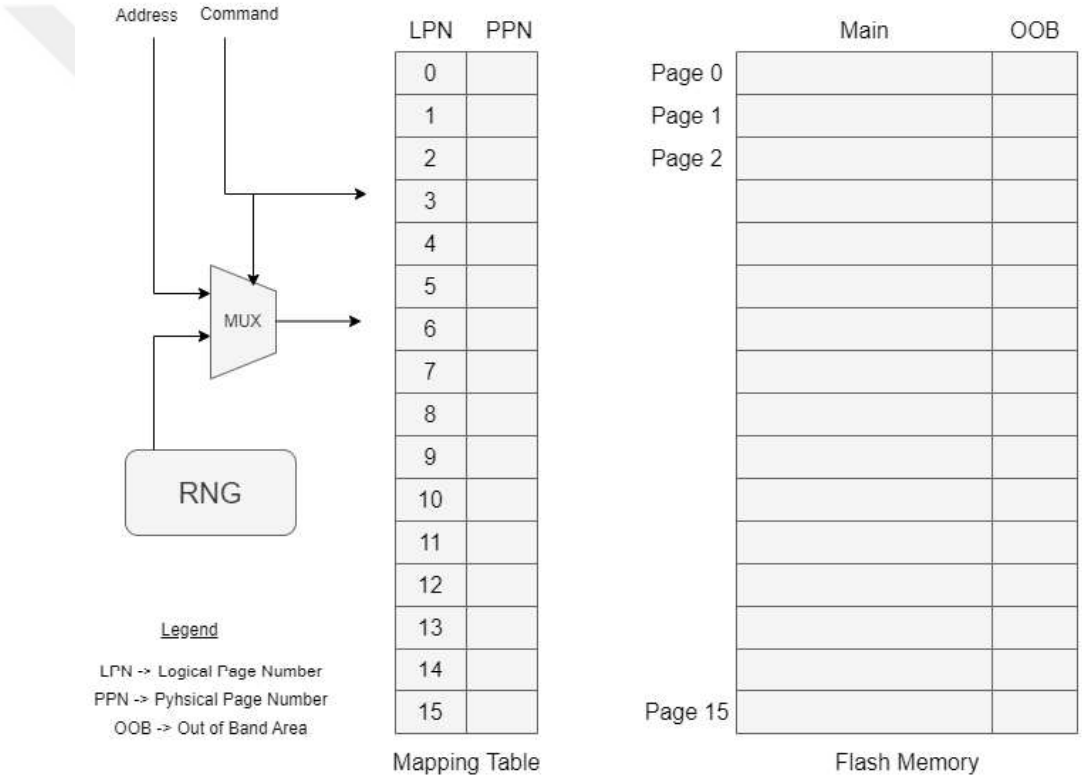

Figure 4.1. RFTL Architecture.

In this demonstration, the flash memory consists of 16 pages. The functionality of the regions LPN, PPN, and OOB are the same as usual page-level FTL schemes, which were explained in Section 3.1.1. Page read and page erase operations are carried out in the same way. However, when a page write operation is intended to perform,

address generation is done with the help of a random number generator. A detailed explanation will be made in the following sections, and the following sections show how the RFTL algorithm handles different operations.

4.1. Read Operation

Figure 4.2 shows how the read operation is done in the RFTL algorithm. The data in logical address 7 is meant to be read. The corresponding physical address in the mapping table is 4, so the data "DEF" in the physical address 4 of flash memory is read.

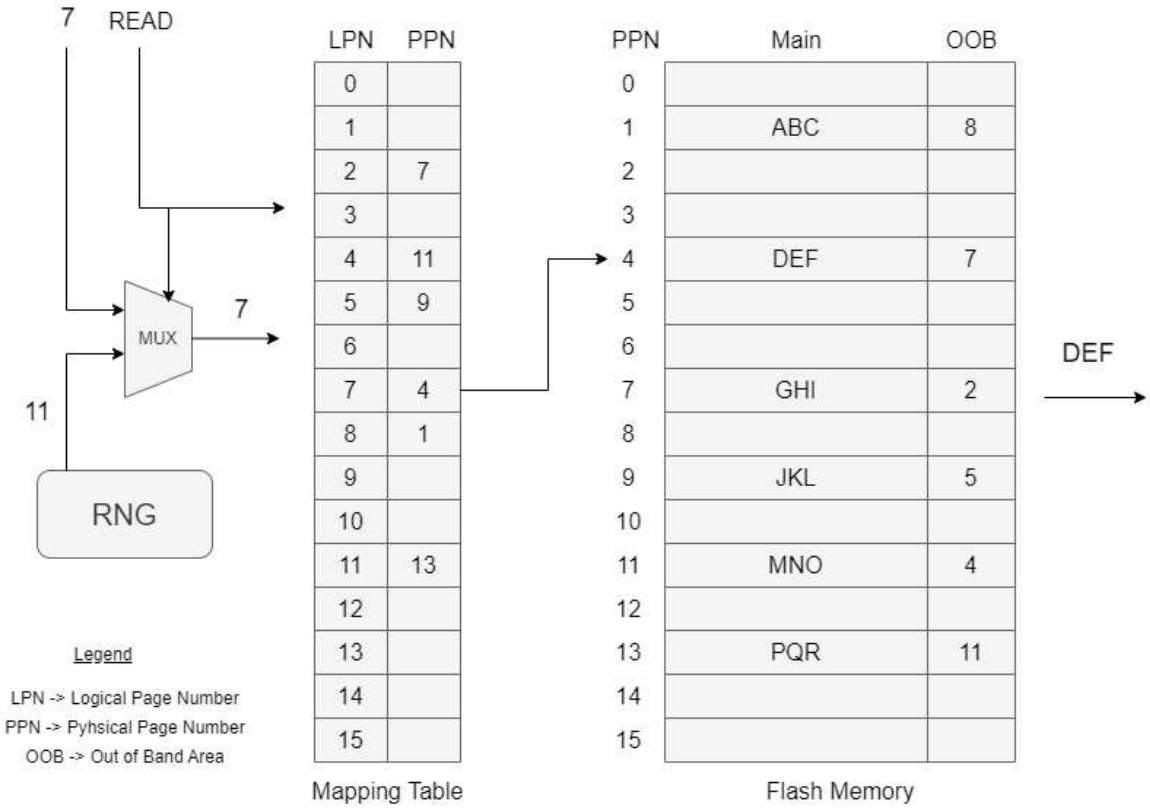

Figure 4.2. Read Operation in RFTL Algorithm.

4.2. Erase Operation

Erase operation has the same approach as read operation. Consider the case, which is shown in Figure 4.2 by changing the command to "ERASE". The mapping table gives the physical address as "4" and the physical address 4 in flash memory is erased. After erase operation, the mapping table and the OOB region of the flash memory must be updated. The resulting scheme of Erase (7) operation is shown in Figure 4.3.

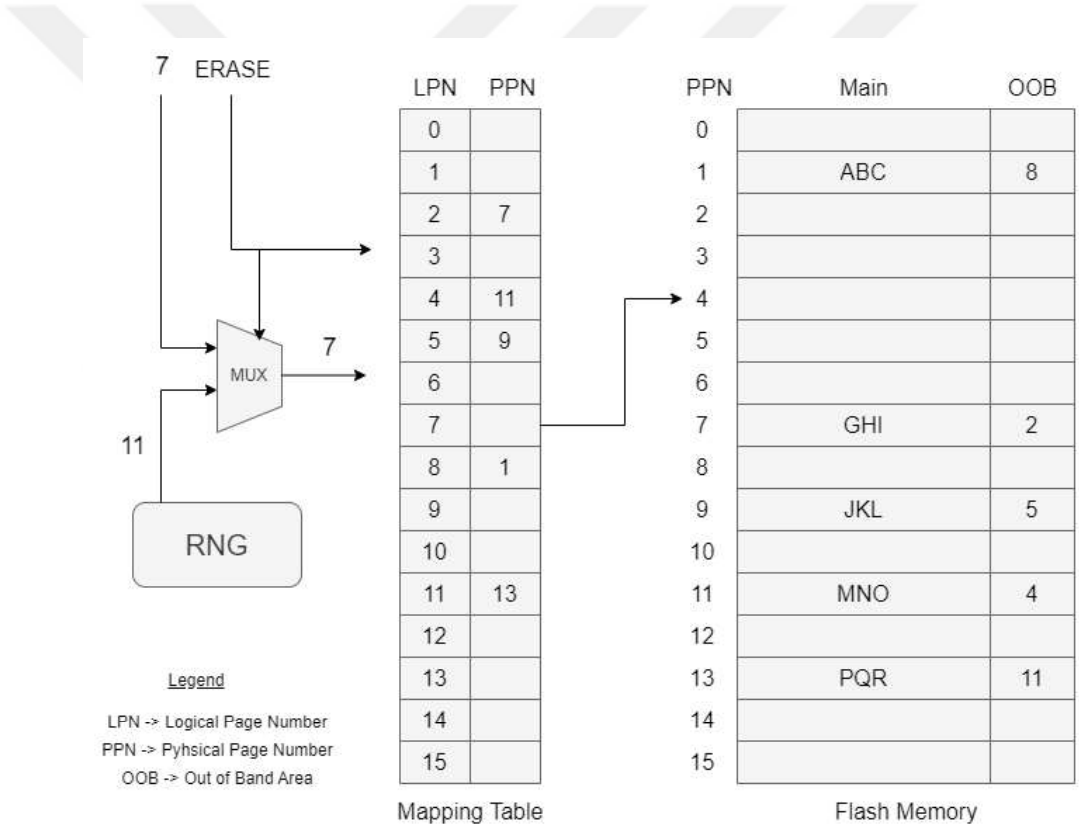

Figure 4.3. Erase Operation in RFTL Algorithm.

For read and erase operations in the RFTL algorithm, the user might give an invalid address, which means the given address is not mapped. In this case, the algorithm returns 0x0 values for the read operation and takes no action for erase operation.

4.3. Write Operation

The page write operation is the most important operation because it reveals the unique aspects of FTL algorithms compared to others. In the following subsections, we will see how the RFTL algorithm accomplishes different cases for the write operation.

4.3.1. Case 1

As shown in Figure 4.4, the user tries to write "FTH" data to address 0, but the address that comes from RNG is 14.

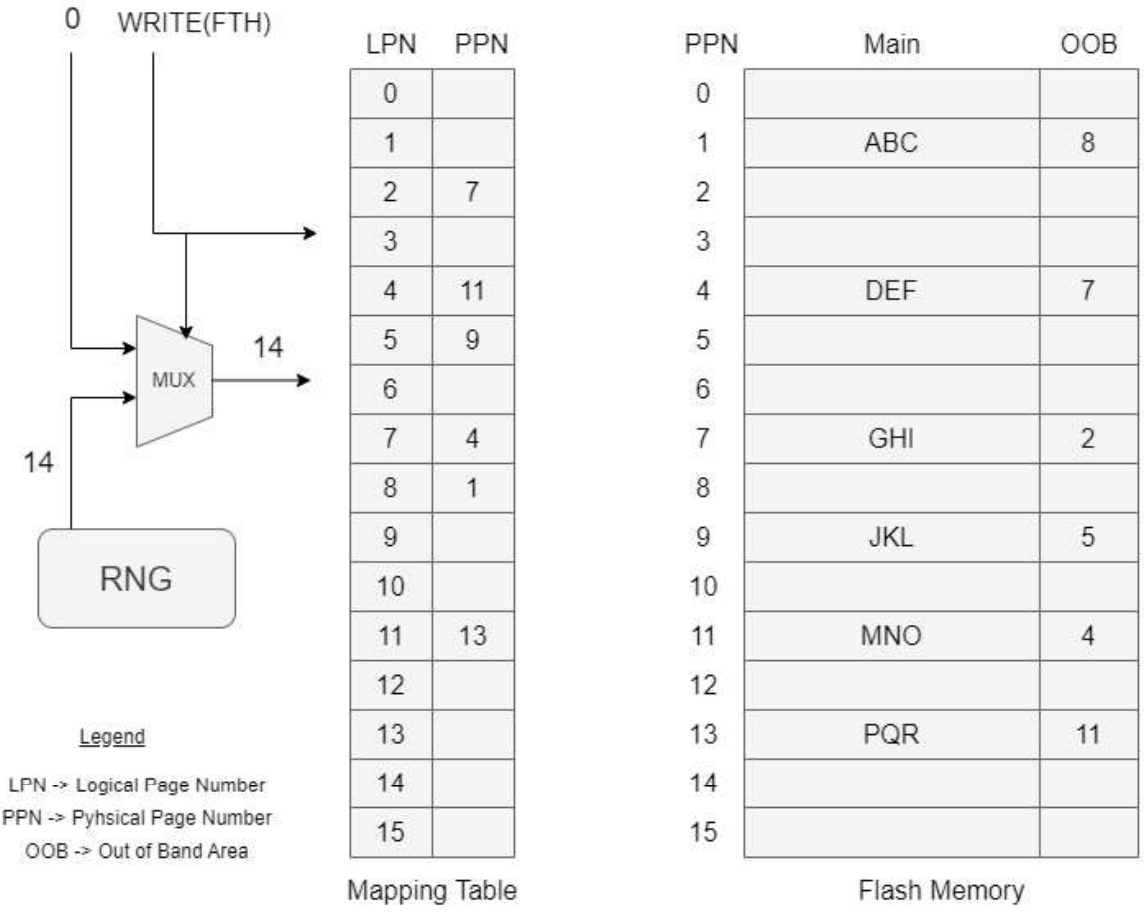

Figure 4.4. Case 1 of Write Operation in RFTL Algorithm.

In this situation, there is no mapping in logical address 0 in the table and the physical address 14 is also empty. Therefore, the data “FTH” will be written into the physical address “14” and the mapping table will be updated as shown in Figure 4.5.

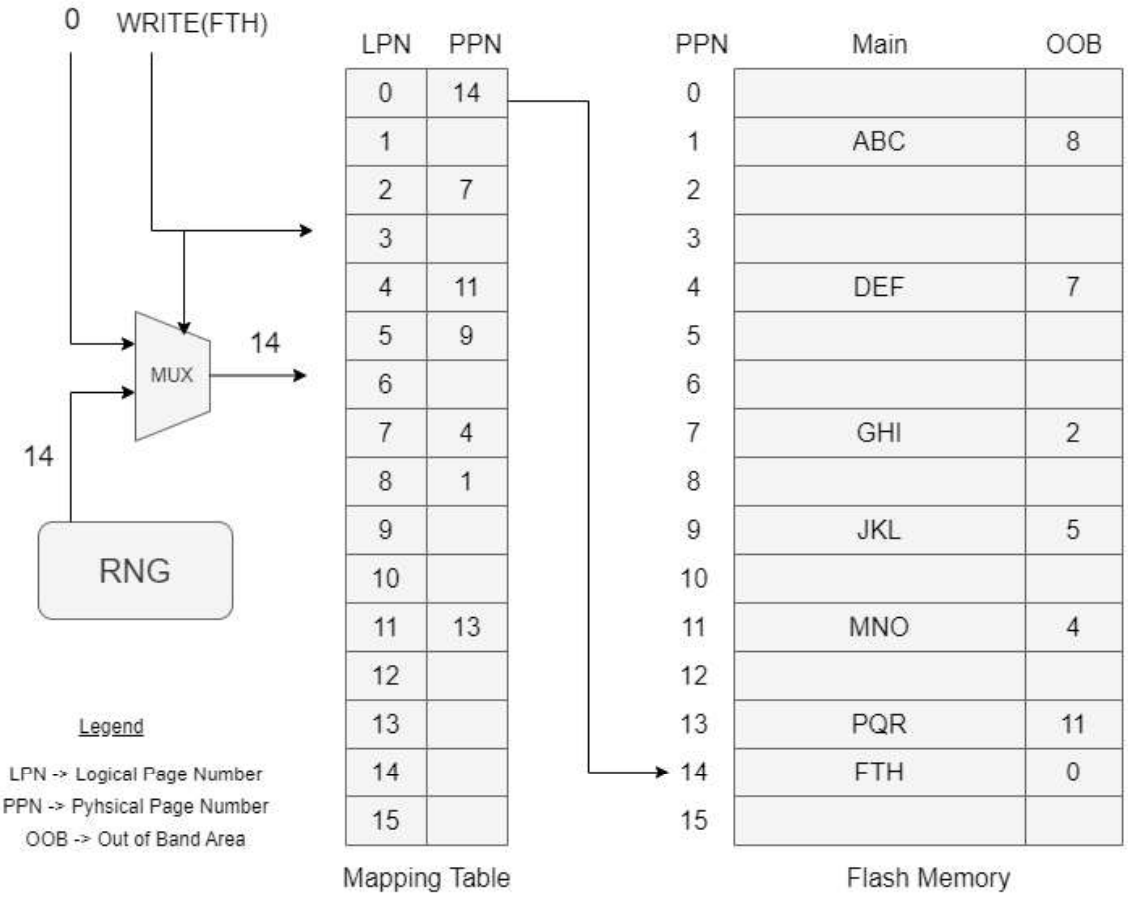

Figure 4.5. Case 1 Resulting Scheme of Write Operation in RFTL Algorithm.

4.3.2. Case 2

In this case, which is shown in Figure 4.6, the user wants to write “FTH” data to address 11, but the address that comes from RNG is 15. Moreover, logical page address 13 keeps the physical address 15 in the mapping table, which means the physical address 15 is occupied. So, the algorithm writes the data “FTH” into there without erasing

because the page was already erased. Then, it erases the physical page 13 of the flash because it is no longer needed for the user. Finally, the algorithm updates the mapping table by replacing the value 13 with 15 in logical page number 11.

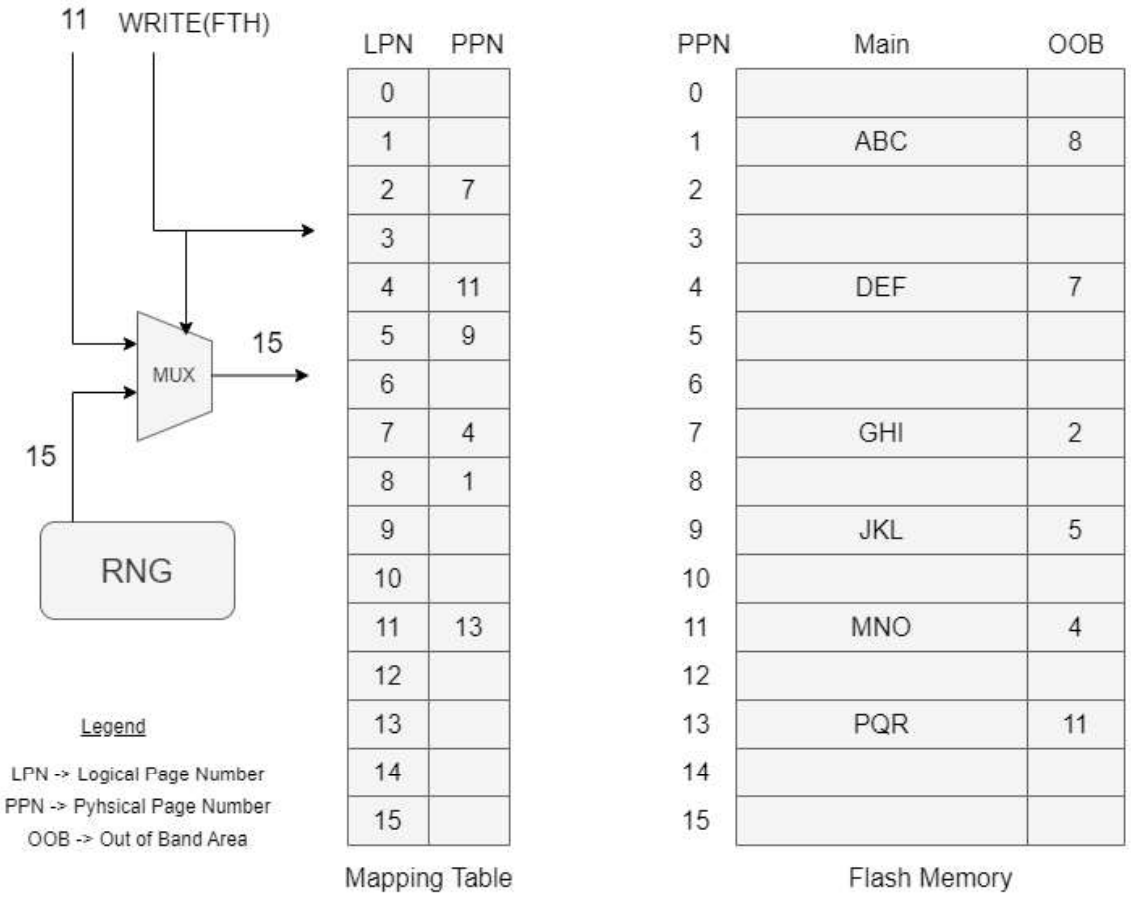

Figure 4.6. Case 2 of Write Operation in RFTL Algorithm.

The resulting mapping table and flash memory are shown in Figure 4.7. It can be seen that in this case of the write operation, RFTL did not cause an extra erase operation due to writing data to address different than the user's demand. Without any FTL algorithm, address 11 would have been erased and the data would have been directly written into address 11. Therefore, the operation could finish with only one erase as in this case. As it will be explained later in detail, FTL algorithms aim to

utilize flash memory with a good address distribution to increase flash's number of write operations. The benefit of this will be seen in the application section.

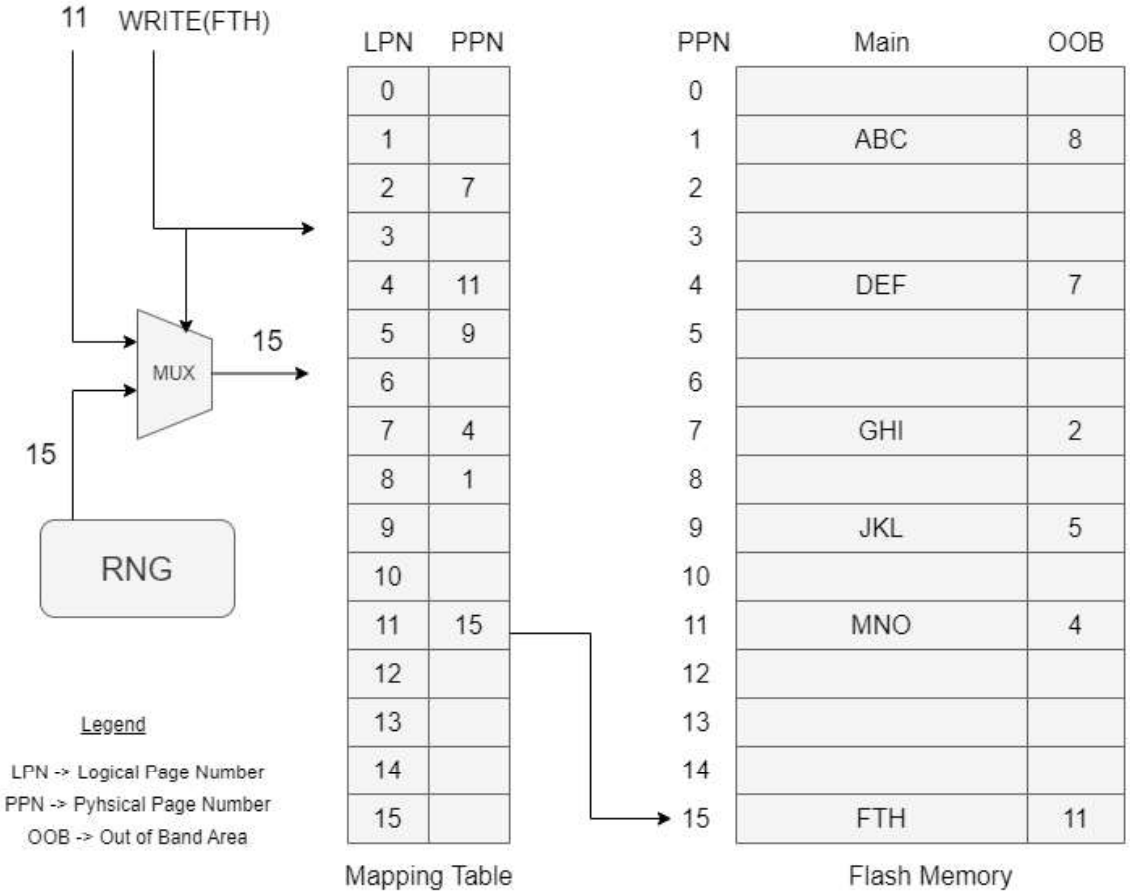

Figure 4.7. Case 2 Resulting Scheme of Write Operation in RFTL Algorithm.

4.3.3. Case 3

As shown in Figure 4.8, the user tries to write "FTH" data to address 0, but the address that comes from RNG is 7. In this case, the physical page 7 in flash memory is occupied, and it is kept on logical page number 2 in the mapping table. In such a situation, the RFTL algorithm first finds the next available physical address in flash memory by increasing it one by one. The next empty physical page in our case is the

page 9. Then RFTL copies the data in physical page 7 to physical page 9. After that, logical page number 2 in the mapping table is updated with the new value, which is 9. Then, physical page 7 is erased, and the new data “FTH” is written into it. As the final step, the logical page number is mapped with physical page number 7 in the mapping table.

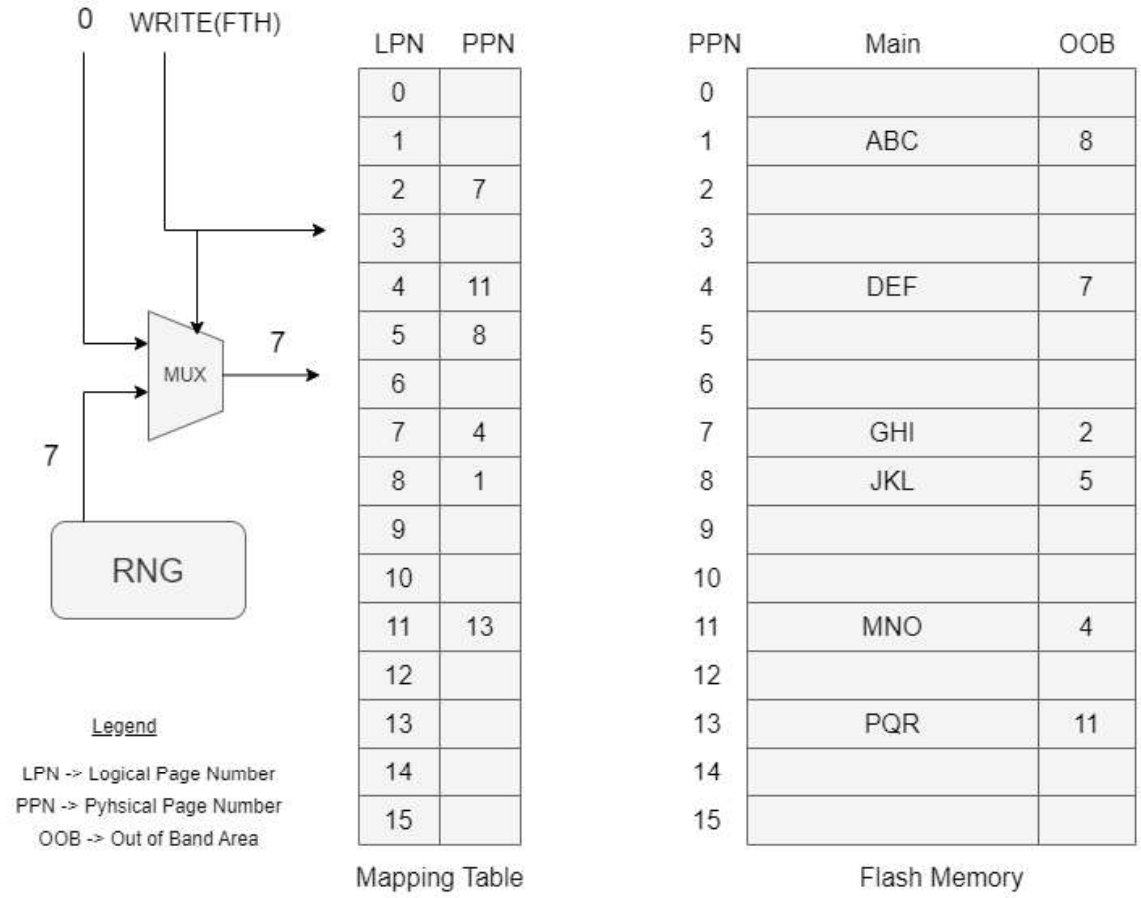

Figure 4.8. Case 3 of Write Operation in RFTL Algorithm.

The resulting scheme of the mapping table and the flash memory is shown in Figure 4.9. Without any FTL algorithm, this writing operation would end up without erase operation because address 0 is already empty and there is no need to erase it. As in case 2, RFTL brought along with one erase operation, which is the erase of address

7 before the new data is written. Again this is a quite tolerable trade-off considering its effect on how effectively flash is utilized.

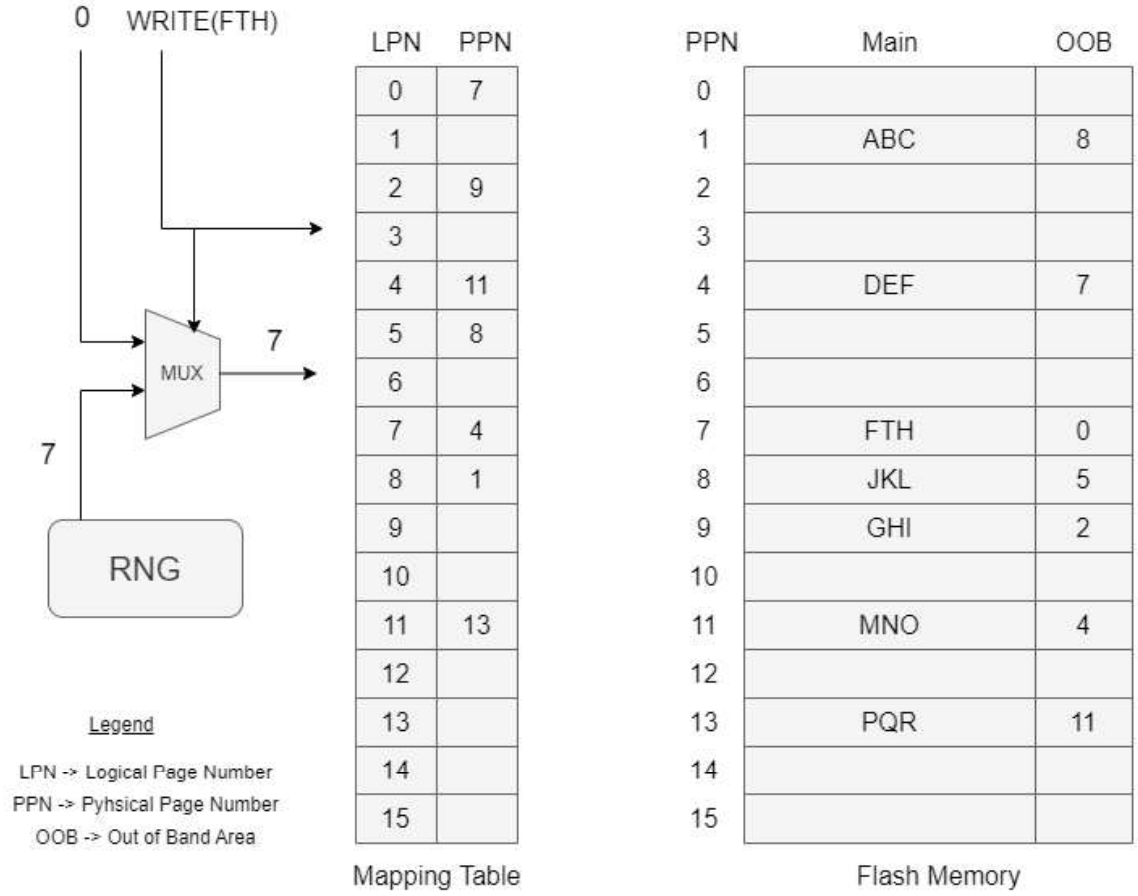

Figure 4.9. Case 3 Resulting Scheme of Write Operation in RFTL Algorithm.

4.3.4. Case 4

Case 4 is almost the same as case 3 with only one difference which is the logical page, which the user wants to write has a mapping physical page in the mapping table. Consider the case given in 4.8, but now the user tries to write the data “FTH” into the logical page 11. It is obvious that the logical page is mapped to the physical page 13 in the mapping table. The procedure is no different than case 3 except for one more

step. After all operations, physical page 13 must be erased. In this case, the number of erasures is two and this is the worst case considering the number of erasures. The final scheme of the architecture is shown in Figure 4.10.

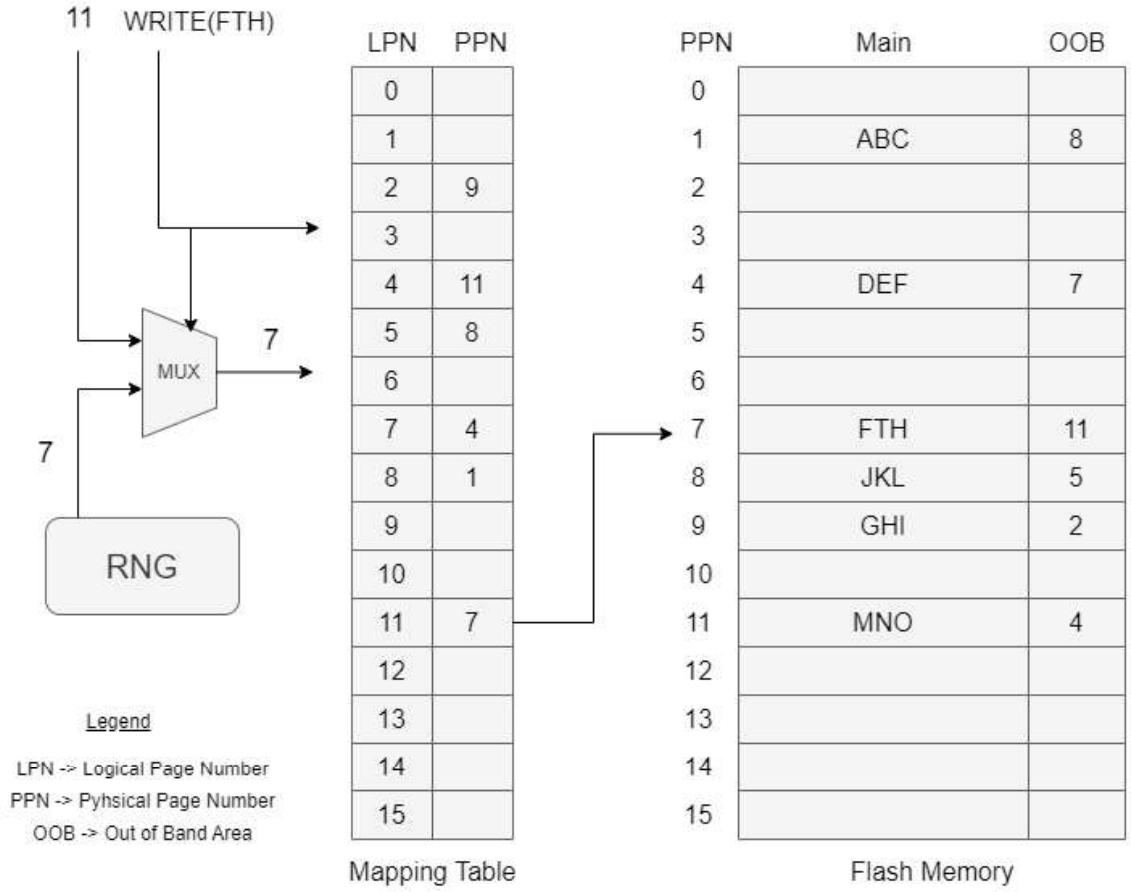

Figure 4.10. Case 4 Resulting Scheme of Write Operation in RFTL Algorithm.

Up to this point, we have examined all the situations that can occur in the page write process. As can be seen, the most complex of these cases is the 4th case. In this case, a single write operation requires two erasing operations. Despite all these disadvantages, the RFTL algorithm makes a great contribution to the efficiency of flash memory usage. In the following sections, we will examine the implementation of the RFTL algorithm as RTL and the test environment and results of this algorithm.

5. RTL DESIGN OF THE ALGORITHM

RTL coding of the design is made in accordance with the APB bus structured system-on-a-chip. A complex integrated circuit known as a "system-on-a-chip," or "SoC," has CPU cores, memory, hardware logic, peripherals, and other components that are all connected by internal data buses or networks. Basic SoC architecture is shown in Figure 5.1.

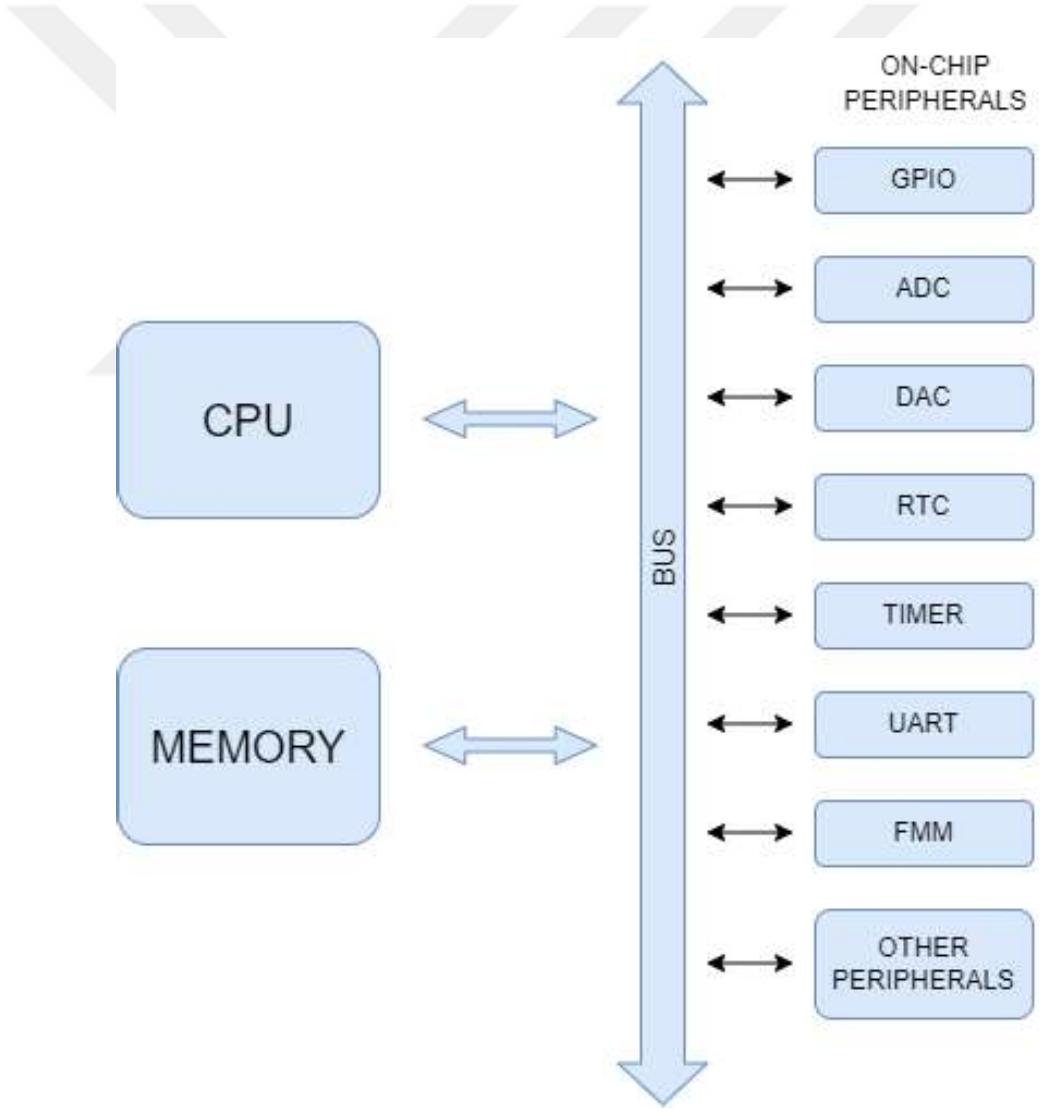

Figure 5.1. Basic SoC Architecture.

More and more SoC architectures are built on many cores. Applications are divided among various processor cores in symmetric multiprocessing. In asymmetric multiprocessing, the cores may play distinctly diverse roles, with some handling I/O and executing demanding real-time work while others do executive chores. These different SoC architectures present different connectivity and programming difficulties.

SoC architectures can support a variety of memory configurations and memory types. While dynamic random-access memory (DRAM) frequently makes up the lower-level primary memory of SoCs, static random-access memory (SRAM) may be employed for processor registers and fast level 1 (or L1) caches. Flash memory block, which is driven by Flash Memory Management (FMM) is also involved in the memory block.

SoC architectures have added a lot of peripherals, frequently to handle well-known communications protocols. GPIO, PCI-Express, CAN, SPI, USB, UART, and I2C are a few popular interfaces. Analog-to-digital converters (ADC), digital-to-analog converters (DAC), and pulse-width modulators (PWM) are examples of peripherals found in SoC architectures for microcontrollers.

SoC architectures require communication between the various modules in order to convey instructions and data. This function has been served by bus-based communication ever since the first SoCs were created. Arm's Advanced Microcontroller Bus Architecture (AMBA) standard is one of the most widely used bus architectures. The semiconductor industry has widely adopted the AMBA Advanced eXtensible Interface (AXI).

As mentioned before, the FTL algorithm was designed compatibly with the APB bus structure. APB stands for Advanced Peripheral Bus and it is part of the Advanced Microcontroller Bus Architecture (AMBA) protocol family. APB is mainly used as general-purpose register-based peripherals such as UARTs, timers, interrupt controllers, etc. The new SoC architecture with FTL addition is shown in Figure 5.2.

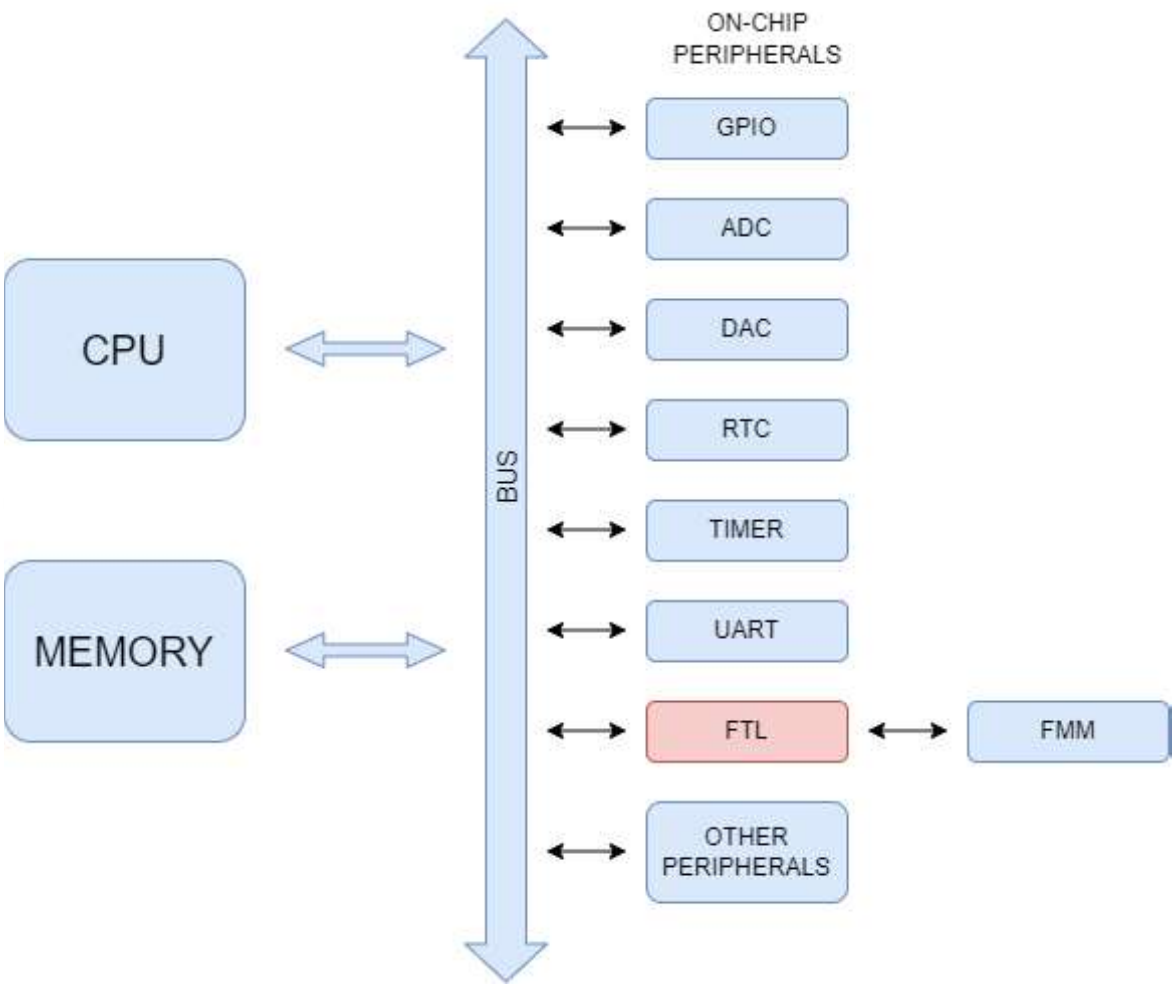

Figure 5.2. SoC Architecture with FTL Addition.

In this design, the RFTL block is placed between the bus structure and the Flash Memory Management. The RFTL block communicates with the CPU via the APB bus, as well as the FMM block with the APB bus signals. In this case, if the RFTL block is removed, the FMM can be directly connected to the bus structure and the system can be operated without any adjustment. In this structure, the RFTL block can understand and manipulate the commands sent by the CPU to the FMM, such as page reading or page writing, or transmit it to the FMM as it is. It also transfers the commands going out of the FMM to the Flash memory to the bus structure. The RFTL block keeps the manipulation information it makes on the incoming commands

completely within itself. The RFTL block keeps the manipulation information on incoming commands completely within itself and does not inform the CPU about the changes it has made. Therefore, the operating system thinks that the data in the flash memory is kept in its own way. The functions of the RFTL algorithm can be classified as follows.

- Command bypass: The RFTL algorithm understands from the address space of the instruction that it should bypass the incoming instruction and forwards the instruction to the FMM as it is. It also transmits the feedback of the bypass command from the FMM to the bus.
- Read status reg: The RFTL algorithm transmits the status register it contains to the processor over the bus when this command arrives.
- Initialization: With the initialization command, all pages of the flash memory are scanned one by one. The mapping table is updated according to the information on whether the pages are used or not.
- Page read: When this command comes, the RFTL algorithm reads the page corresponding to the page to be read in the mapping table from the flash memory.
- Page erase: The page corresponding to the incoming page number in the mapping table is deleted in the flash memory and the mapping table is updated.
- Page write: When this command comes, the page is written according to the algorithm described in Section 4.3.
- Integrity check: The integrity check operation is used to check the content of the mapping table for corruption. The odd-numbered and even-numbered addresses of the table are XORed between themselves. The two results are combined and transmitted to the bus.
- Read table: When this command comes, the RFTL algorithm transmits the data at each address of the mapping table to the bus one by one.

5.1. Structure of the RFTL Algorithm

The general structure of the proposed RFTL algorithm was mentioned in the previous sections. In this section, we will examine in detail the RFTL blocks designed as RTL. The structure of the design is shown in Figure 5.3.

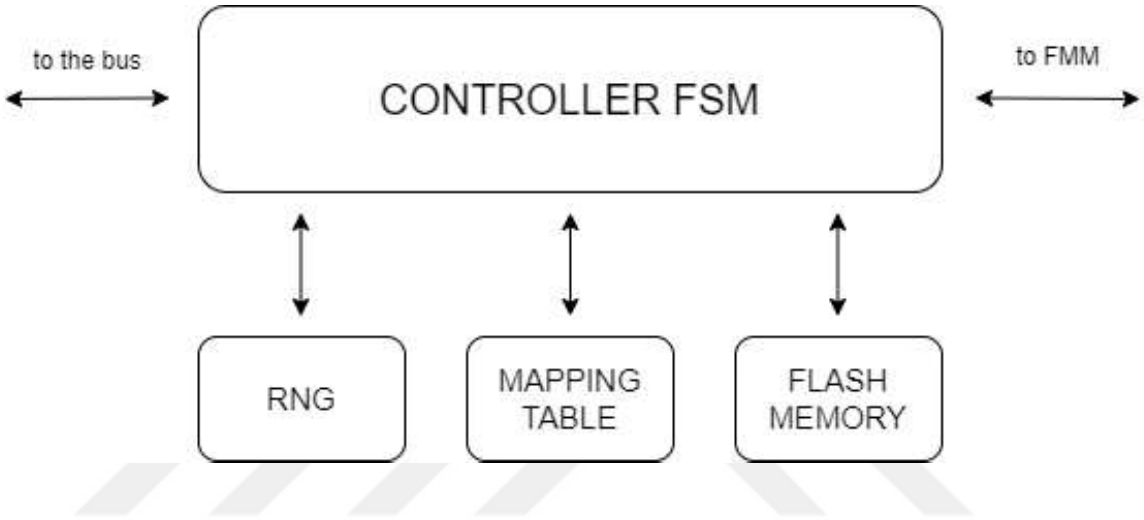

Figure 5.3. The Design Structure of RFTL Algorithm.

As seen in the figure, a controller finite state machine (FSM) is designed to manage the whole system. The controller manipulates the instructions and provides the connection between the bus, flash memory, and FMM. Also, the RNG block, mapping table, and flash memory are not interconnected. These blocks are directly controlled by the FSM. In the following subsections, these blocks are examined in detail.

5.1.1. Flash Memory Block

No real flash memory was used in the RTL design of the system. Instead, a random access memory (RAM) that was modeled according to the working mechanism of flash memory was used. Thus, the design process has been simplified and the testing process has become more flexible.

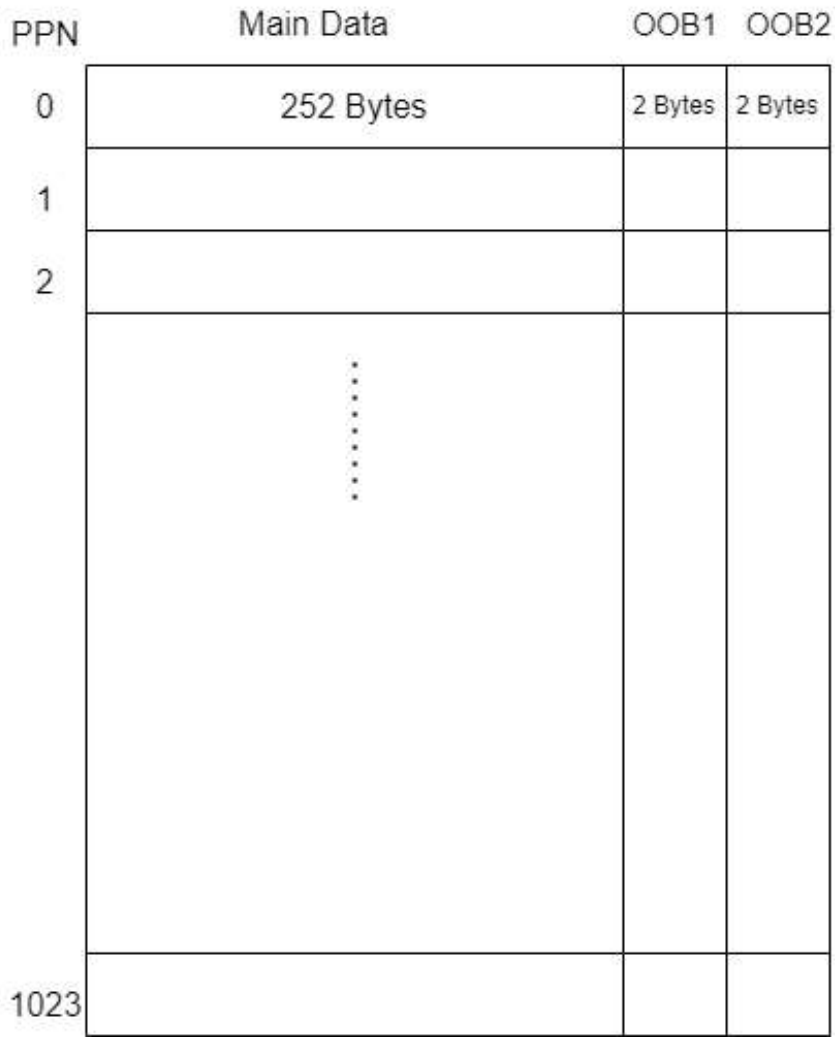

Figure 5.4. The Design Structure of Flash Memory.

The structure of the flash memory design is shown in Figure 5.4. Each row in the design is called a page, and each page is 256 bytes in size. There are 1024 pages in the flash memory. Therefore, the flash block has a total size of 256 kilobytes. User data is kept in the main data section. OOB1 and OOB2 regions are manipulated by the controller. The OOB1 part of the page where data is written in the flash memory is filled with 0xAA55 data, and this part shows whether the page is valid or not. The OOB2 section shows the logical page equivalent of this page in the mapping table.

5.1.2. Mapping Table

As in other algorithms, RAM is used as a mapping table in the RFTL algorithm. Since there are 1024 pages in the flash memory, there are also 1024 lines in the RAM. If the logical page valid (LPV) is 1 in any row, that logical page is mapped to the physical page written in the PPN (physical page number) section. If the physical page valid (PPV) value in any row is 1, this page is physically used and mapped to a logical page in the table. Although the PPN part is 14 bits, only 10 bits are used because there are 1024 pages in the flash memory.

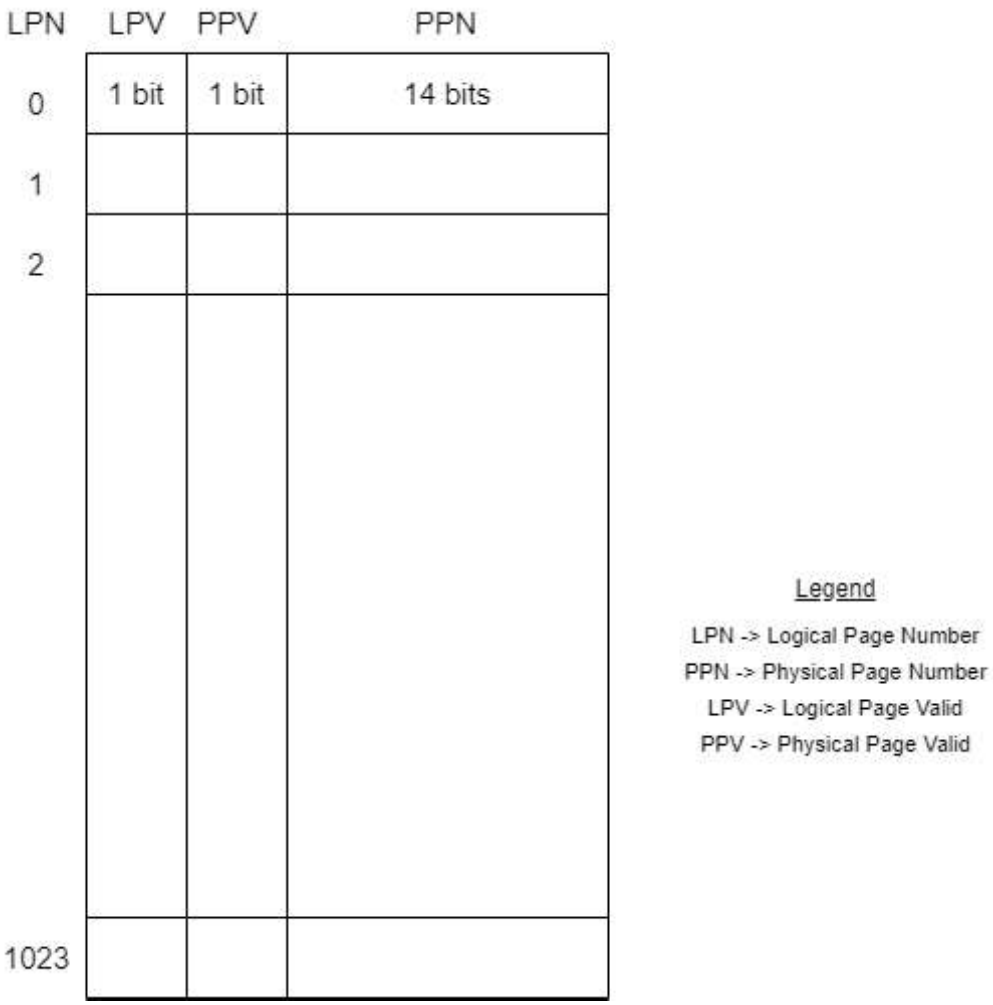

Figure 5.5. The Design Structure of the Mapping Table.

5.1.3. RNG Block

Linear-feedback shift register (LFSR) is used as a pseudo-random number generator (RNG) in the RFTL algorithm. A shift register called a linear-feedback shift register (LFSR) has an input bit whose value is linearly related to its prior state. Exclusive-or (XOR) is the most used linear function for single bits. As a result, an LFSR is typically a shift register whose input bit is driven by the XOR of some of the shift register's total value. Since the operation of the register is deterministic, the sequence of values that the register produces is entirely governed by its present (or previous) state. The beginning value of the LFSR is referred to as the seed. Similarly, the register must eventually enter a repeating cycle since there are only a finite number of potential states. A well-chosen feedback function on an LFSR, however, can result in a series of bits that appear random and have a very long cycle.

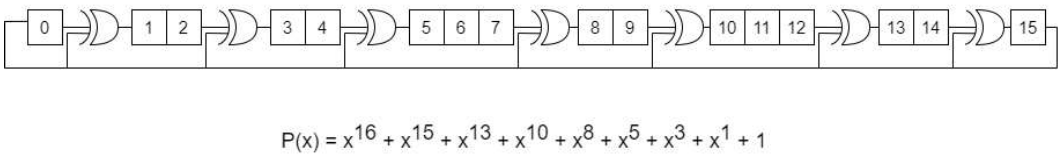

Figure 5.6. The Schematic of the LFSR.

Figure 5.6 shows the schematic of the linear-feedback shift register (LFSR) block used in the RFTL algorithm. The characteristic polynomial of the LFSR is also shown in the figure. A seed value can be entered externally to LFSR in the design. Although the designed LFSR block is 16 bits, only the first 10 bits are used because the number of pages in the flash memory is 1024. This circuit, which generates a pseudo-random number sequence, provides the necessary randomness for the writing operation.

5.1.4. Controller Finite State Machine

The state diagram of the finite state machine is shown in Figure 5.7. This diagram is an overview of the state machine in the control block. The state diagram shows how page writing and all other commands are executed.

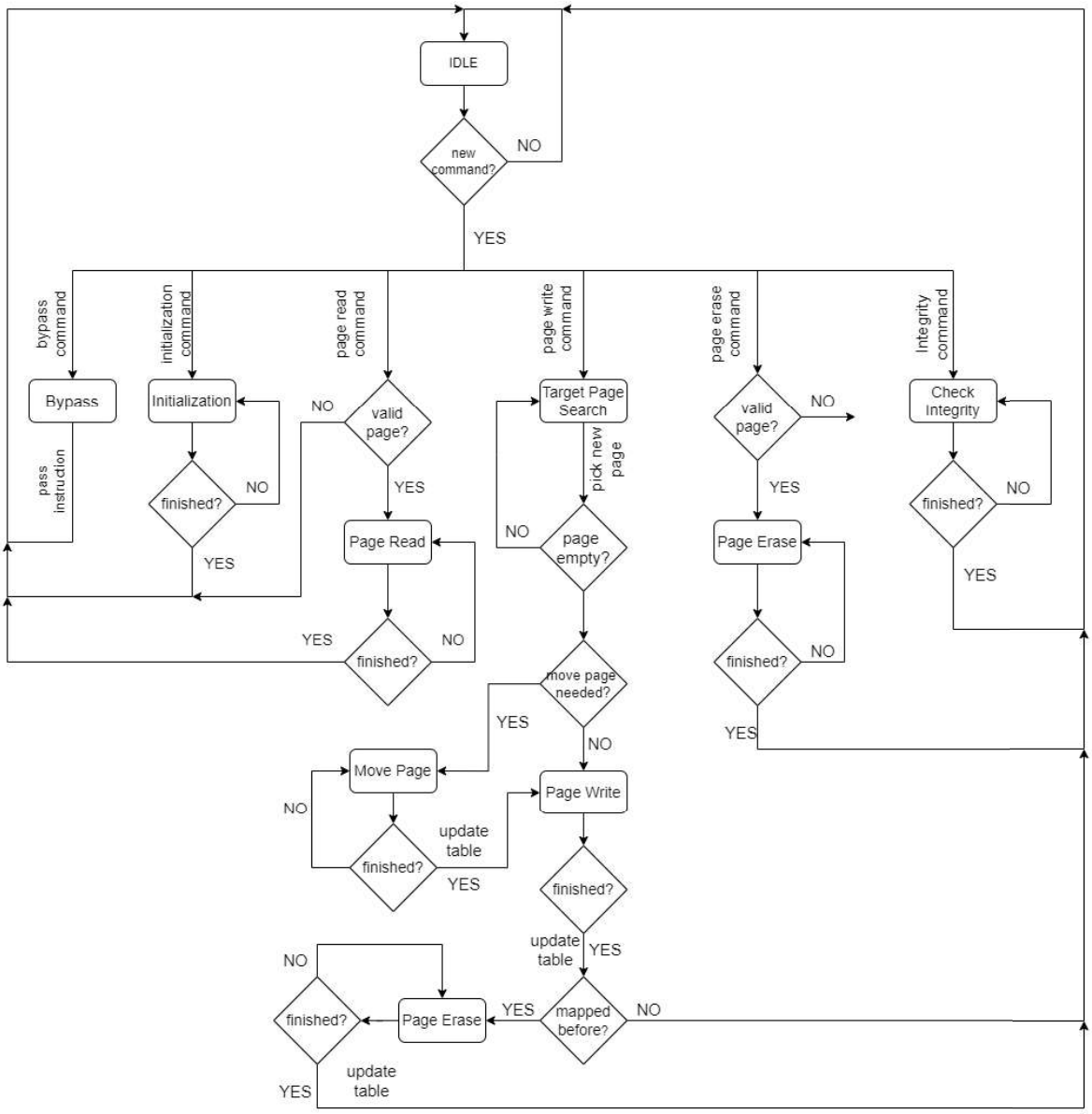

Figure 5.7. The State Diagram of the Controller Logic.

6. TEST RESULTS AND COMPARISON

6.1. Test Setup

The test environment prepared to validate the RFTL algorithm is shown in Figure 6.1. A test software was developed on the personal computer (PC) to test the algorithm. This software sends some commands such as page write, page read and page erase to Field Programmable Gate Array (FPGA) via Universal Asynchronous Receive Transmit (UART) Interface. On the FPGA, the RFTL algorithm and the UART block that provides communication between it and the test software are implemented.

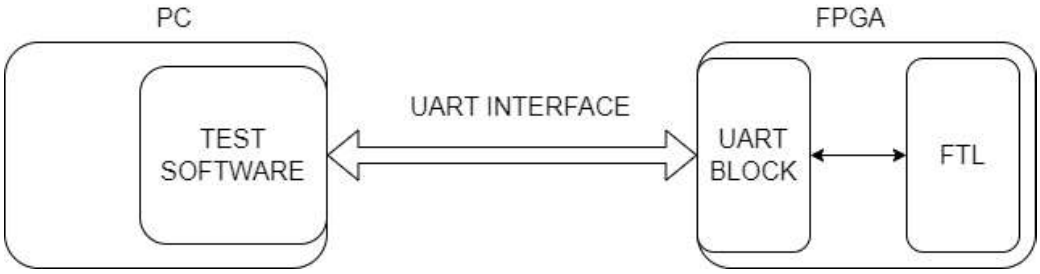

Figure 6.1. The Test Setup of the RFTL Algorithm.

Constructed from a grid of configurable logic blocks (CLBs) coupled by programmable interconnects, field programmable gate arrays (FPGAs) are semiconductor devices. After production, FPGAs can be reprogrammed to meet specific application or feature needs. FPGAs are distinguished from Application Specific Integrated Circuits (ASICs), which are made especially for particular design tasks, by this property. Despite the existence of one-time programmable (OTP) FPGAs, the most common models are SRAM-based and allow for reprogramming when the design changes.

The data format and transmission speeds of a universal asynchronous receiver transmitter (UART), a computer hardware component for asynchronous serial com-

munication, are adjustable. In order to ensure that the communication channel can handle exact timing, it delivers data bits one by one, starting with the least significant and working up to the most significant.

The test software has a structure similar to that of RFTL. It contains a flash memory structure and a mapping table. It also implements the commands it sends to RFTL via the UART interface. By sending an integrity check command periodically, it checks the incoming result within itself and verifies that it is in the same state as RFTL. The characters that the test software sends to the FPGA with the UART interface and the commands corresponding to these characters are as follows:

- i: stands for initialization command. After the RFTL block finishes the initialization, it performs an integrity check. Then, it returns "OK" characters.
- r: stands for page read command. The page address to be read is also sent with this character. Returns OK characters with the first 32 bits of the page read after performing the RFTL page read
- e: stands for page erase command. Along with this character, the page address to be erased is also sent. After the RFTL block finishes the page erase, it performs an integrity check. Then, it returns the 32-bit integrity value and OK characters along with the ":" character in response.
- w: stands for page write command. The RFTL block writes the data in its buffer to the random page address it gets from the RNG. After the write operation is finished, it performs the integrity check operation. The RFTL block returns the ":" character, along with the 32-bit integrity value, the page address from the RNG, and the OK characters.
- d: stands for reading the table. The values in RAM are exported one by one and OK characters are sent when the process is finished.

ASCII values of all characters used in communication between test software and RFTL are sent. In addition, the integrity control process performed within the commands is as described in Chapter 5.

The commands sent from the test software with ASCII values are first interpreted by the UART block on the FPGA. The UART block transmits the commands corresponding to the incoming characters to the RFTL block in accordance with the APB bus signals. And finally, it sends the data it receives from the RFTL block as a response to the test software in ASCII format.

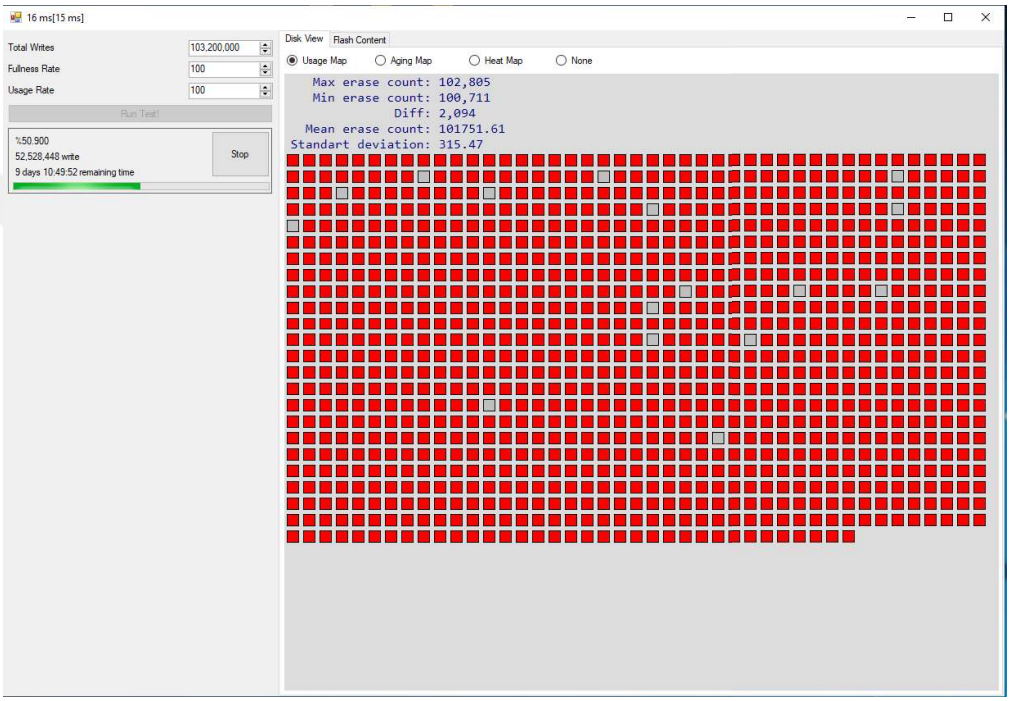

Figure 6.2. The User Interface of the Test Software.

The user interface of the test software is shown in Figure 6.2. As seen in the figure, three parameters can be configured on the interface and these are total writes, fullness rate, and usage rate. Total writes set the total number of writes the software will make during the test. The fullness rate determines how full the 1024 pages in the flash memory will be before starting the test. Usage rate determines how many of the 1024 pages in flash will be used for testing. On the left part of the interface, the number of writes so far and the ratio of this number of writes to the total number of writes set as the target are shown. In addition to these, the remaining time for the end of the test is also displayed.

There are 1024 boxes in the middle of the interface and each box represents a page in the flash memory. These boxes were colored differently during the test. Coloring information for different options appearing in the interface is as follows:

- Usage Map: In this option, pages can have only two colors. it indicates that the pages in red are full and those in gray are empty.
- Heat Map: In this option, pages can take any color in the color spectrum from red to green. The page in red indicates the page with the maximum number of erases, and the page in green color indicates the page with the minimum number of erases.

There is some numerical information at the top of the boxes showing the pages. Of these, the max erases count shows the maximum number of erasures a page has had during the operations so far, while the min erases count shows the least number of erasures a page has had. Diff shows the difference between the maximum number of erases and the minimum number of erases. The mean erase count shows 1024 parts of the total number of pages erased so far. Finally, the standard deviation shows the standard deviation of the number of erases the pages have.

6.2. Test Results

It was explained in the 5.1.1 that a model of RTL is used instead of a real flash memory in design. The flash memory used as a model are flash memories produced with HHGRACE 130nm technology and the lifetime of pages in these memories is about 100 thousand erasing. The usage map and the heat map results of the test are shown in Figure 6.3, and Figure 6.4 respectively.

The aim of the test is to have a tolerable difference between the page with the least number of erases and the page with the highest number of erases when the number of erases reaches 100k. If this goal is achieved, the proposed FTL algorithm, RFTL, will be validated.

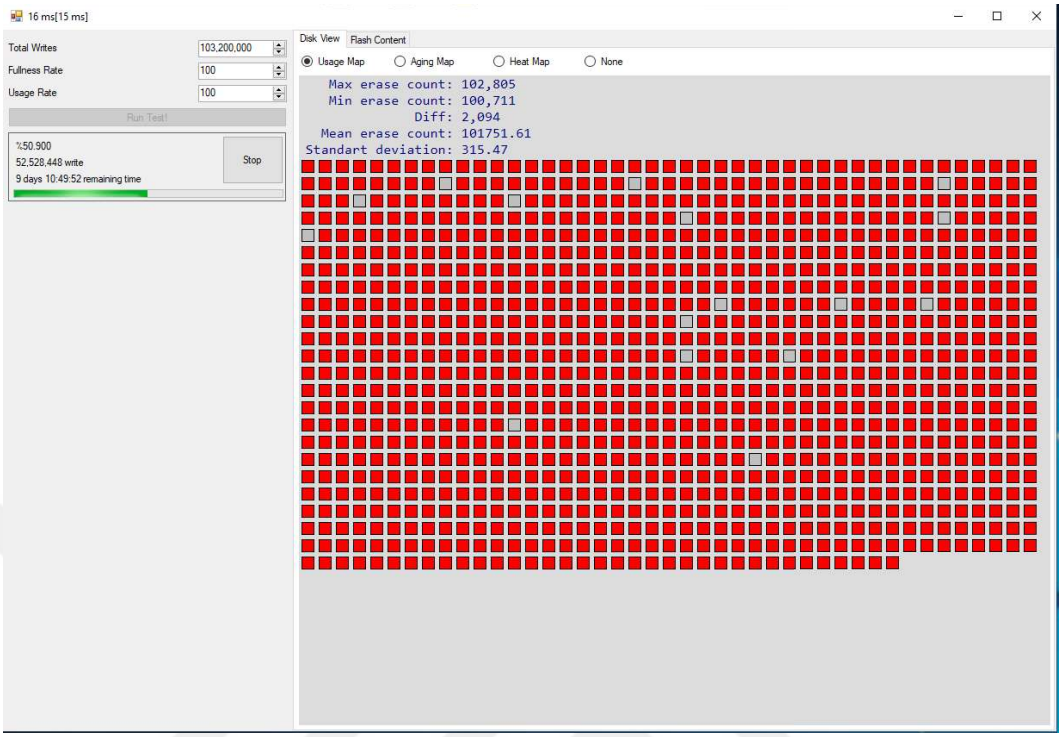

Figure 6.3. The Usage Map of the Test Interface.



Figure 6.4. The Heat Map of the Test Interface.

As seen in Figure 6.3, the maximum number of erases is 102805 and the minimum number of erases is 100711. The difference between them is 2094. The deviation rate between the maximum and minimum number of erases is only 2%. This shows that the use of the pages was realized with a good distribution as aimed. It was also said that mean erase count (which is 101751,61) is the total number of erases of all pages divided by the total number of pages, 1024. The number we get when we average the max erase count and min erase count is 101758,61, and this is almost the same as mean erase count. This result also shows that the distribution between the pages is uniform. Finally, considering that there are 1024 pages and the mean erase count is 100000 thousand, it can be assumed that the total number of writes should be approximately 102400000. But as seen in Figure 6.3, the total number of writes is 52528448, which is about half of the expected number. The reason for this is that since the fullness rate is set to 100, the test starts with all pages filled, which requires erasing two pages for each write as explained in Section 4.3.3 or in Section 4.3.4.

The distribution of colors in Figure 6.4 also shows that the distribution in the number of erasures of the pages is regular. Pages are distributed from red to green. The darkest red one shows the page with the max erase count and the darkest green one shows the page with the min erase count. It is seen that all colors in the color spectrum from red to green are close to each other in terms of quantity.

Finally, the distribution of RNG outputs according to page numbers is shown in Figure 6.5. As can be seen in the figure, the RNG outputs showed a regular distribution around the 52400 value. Considering that the total number of pages is 1024, the distribution of RNG outputs are consistent with the total number of writes shown in Figure 6.3, 52528448. Moreover, it should be noted that the value in the RNG output is not the page address to be written directly. If the flash memory address taken as the output of the RNG is full, the data is written to the next free address as explained in Section 4.3.

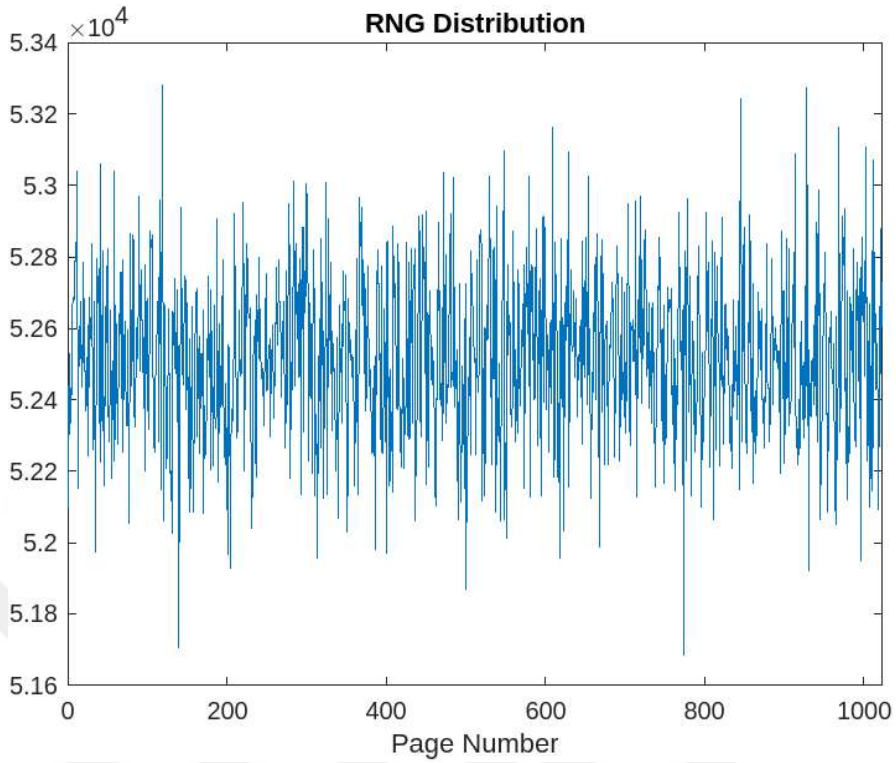

Figure 6.5. The Distribution of the RNG Outputs.

6.3. Comparison

The comparison of the RFTL algorithm with the DFTL and CFTL algorithms is shown in Figure 6.6. DFTL algorithm is page-level FTL while the CFTL algorithm is hybrid FTL. CFTL has been added since it works as a page-level FTL during the write operation. Compared to RFTL, DFTL and CFTL algorithms are much more complex, especially in terms of the write operation. While the DFTL algorithm keeps the entire mapping table in the flash, it also keeps some of it in the SRAM Cache. If the address cannot be found in the cache in the read operation, it must be read from the flash and retrieved. In addition, while writing, it also runs the least recently used (LRU) algorithm for finding empty pages. The CFTL algorithm also keeps the entire mapping table in the flash, and it keeps some of it in the SRAM Cache. The CFTL algorithm defines hot data and cold data for switching between write and read operations. In

order to decide whether the data is hot data or cold data, it runs multiple hash-based algorithms. Considering all these, the read and write operations described in Chapter 4 make the RFTL algorithm much simpler and implementable than DFTL and CFTL.

	DFTL	CFTL	RFTL
Complexity	very high	very high	very low
SRAM Usage	low	medium	high
Flash Overhead	medium	medium	very low
Num. of Mapping Table	2	3	1

Figure 6.6. The Comparison of RFTL Algorithm with Other FTLs.

It is the mapping table that causes the use of SRAM in FTL algorithms. The DFTL algorithm uses two mapping tables, the Cached Mapping Table (CMT) and the Global Translation Directory. The CFTL algorithm uses three tables, Cached Page Mapping Table (CPMT), Cached Block Mapping Table (CBMT), and the complete page mapping table (tier-2 page mapping table). these algorithms keep only a part of the whole mapping table in these tables and they are constantly updating these tables. For this reason, the tables used by these algorithms do not cause problematic SRAM usage. As the RFTL algorithm keeps the mapping table created for each page in SRAM, SRAM usage is high. The DFTL and CFTL algorithms keep the entire mapping table in flash. This causes flash overhead in proportion to the size of flash memories. For the RFTL algorithm, leaving only one flash page blank is sufficient to run the entire algorithm. The flash overhead for the RFTL algorithm can be considered zero.

The maximum number of writes that each page can perform in the flash memory model used in the test environment is 100k. Physically, no algorithm can pull this number up. All FTL algorithms aim to maximize the write operation that can be done to distribute the incoming write commands to different addresses. Technically, all algorithms achieve this. However, they use very complex algorithms to achieve the goal and cause flash memory overhead. Unlike the others, the RFTL algorithm does not cause any flash overhead and achieves the same goal with a very simple method. Finally, the random number generator-based approach of the RFTL algorithm has brought a method to the literature, thus paving the way for new studies.

7. CONCLUSION

In this thesis, a simple and quickly implementable Flash Translation Layer (FTL) algorithm for NOR flash memories used in embedded systems is presented. The amount of data managed by applications running on the systems is increasing day by day. Managing such large data requires a lot of access to flash memory. Considering that the technology used in flash memories has a certain lifetime, the efficient use of flash memories plays a key role in system designs. Hence, the topic of this thesis is relevant to the current demands.

In the RFTL algorithm, there is a mapping table as in other page-level FTLs and one-to-one mapping of all flash memory pages is kept in this table. Without making any changes in page reading and page erase processes, the page to be processed directly is found in the table and the operation is performed. The original part of the RFTL algorithm is its solution on the write operation. It uses a random number generator-based algorithm instead of complex algorithms such as the least recently used (LRU)-based on determining the page address to be written. If data has already been written to the page address to which the data will be written, the data at this address is moved to the next empty address and then the data is written to the emptied address. Afterward, all these changes are updated in the table. Compared to other algorithms, this solution method is very effective and simply adaptable.

The register transfer level (RTL) design of the algorithm consists of four blocks, these are the mapping table, flash memory, random number generator (RNG) block, and controller finite state machine (FSM). As flash memory, a RAM-modeled flash memory produced in HHGRACE 130nm technology was used. size of RAM used is 1024X256 bytes. RAM is also used as a mapping table. The size of the RAM is 1024X2 bytes, as this table keeps one-to-one mapping information for each page. The linear-feedback shift register (LFSR) is used as a random number generator. Although the LFSR used is 16 bits, 10 bits are used because the total number of pages is 1024. Finally,

the controller FSM is the part that manages all flash commands such as page writing and page reading, and the manipulations on the mapping table and flash memory. The RFTL algorithm is designed in accordance with the APB bus structure and can be integrated into any system-on-chip design.

A test software was designed on the computer to test the algorithm. The designed RFTL algorithm was implemented on the FPGA and the commands from the test software were transmitted to the FPGA via the UART interface. The RFTL algorithm sends the operations it performs in response to the incoming commands to the test software over the same path as a response. The test software was able to calculate the number of erases for all pages with this information. The results showed that when the number of erases on all pages reaches 100 thousand (the maximum number of erasings that a flash page can do without losing its functionality), there is a difference of about 2,000 between the page with the highest number of erases and the page with the least erases. This shows that even in the worst case, a success rate of 98% is achieved. This success rate means that the rng-based RFTL algorithm has been validated. In addition, the algorithm proposed by this thesis is much simpler and implementable than complex algorithms such as the least recently used (LRU) and hot-cold data detection suggested in other studies. Also, unlike other FTL algorithms, the flash memory overhead of the RFTL algorithm is almost 0. This algorithm can be run by allocating only one flash memory page. Other FTL algorithms keep only part of the mapping table in SRAM Cache, and when an address miss occurs, they have to read that address from the flash and bring it to the Cache. Since the RFTL algorithm keeps the one-to-one mapping information of all pages in RAM, no address miss situation occurs.

REFERENCES

1. Ban, A., "Flash File System Optimized for Page-Mode Flash Technologies", U.S. Patent 5,937,425, 10 August 1999, `https://patents.google.com/patent/US5937425A/en`, accessed on December 28, 2022.

2. Gupta, A., K. Youngjae and B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand based Selective Caching of Page-Level Address Mappings", *ACM SIGARCH Computer Architecture News*, PA, USA, 2009.

3. Kim, J., J. M. Kim, S. Noh, S. L. Min and Y. Cho, "A Space-Efficient Flash Translation Layer for Compact Flash Systems", *IEEE Transactions on Consumer Electronics*, Vol. 48, No. 2, pp. 366–375, 2002.

4. Sanvido, M. A. A., F. R. Chu, A. Kulkarni and R. Selinger, "Nand Flash Memory and Its Role in Storage Architectures", *Proceedings of the IEEE*, Vol. 96, No. 11, pp. 1864–1874, 2008.

5. Zitlaw, C., "The Future of NOR Flash Memory", 2011, `https://www.eetimes.com/the-future-of-nor-flash-memory/`, accessed on January 1, 2023.

6. David A. Patterson, J. L. H., *Computer Organization and Design: The Hardware / Software Interface*, Morgan Kaufmann, Burlington, MA, USA, 2008.

7. Sedra, A. S. and K. C. Smith, *Microelectronic Circuits*, Oxford University Press, New York, USA, 2004.

8. Weste, N. H. E., *CMOS VLSI Design: A Circuits and Systems Perspective*, Addison Wesley, Boston, Massachusetts, USA, 2011.

9. Kahng, D. and S. M. Sze, "A Floating Gate and Its Application to Memory Devices", *The Bell System Technical Journal*, Vol. 46, No. 6, pp. 1288–1295, 1967.

10. Niset, M. and P. Kuhn, "Typical Data Retention for Nonvolatile Memory", 2005, `https://www.nxp.com/docs/en/engineering-bulletin/EB618.pdf`, accessed on January 2, 2023.

11. Bez, R., E. Camerlenghi, A. Modelli and A. Visconti, "Introduction to Flash Memory", *Proceedings of the IEEE*, Vol. 91, No. 4, pp. 489–502, 2003.

12. Sakib, S., P. Kumari, B. M. S. Bahar Talukder, M. T. Rahman and B. Ray, "Non-Invasive Detection Method for Recycled Flash Memory Using Timing Characteristics", *Cryptography*, Vol. 2, No. 3, p. 17, 2018.

13. Fazackerley, S., W. Penson and R. Lawrence, "Write Improvement Strategies for Serial NOR Data Flash Memory", *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, Vancouver, BC, Canada, 2016.

14. Rino Micheloni, P. O., Giovanni Campardo, *Memories in Wireless Systems*, Springer, Heidelberg, Germany, 2008.

15. Rino Micheloni, R. R., A. Marelli, *Error Correction Codes for Non-Volatile Memories*, Springer, Dordrecht, Netherlands, 2008.

16. Jin, Y. and B. Lee, "A Comprehensive Survey of Issues in Solid State Drives", *Advances in Computers*, Vol. 114, No. 1, pp. 1–69, 2019.

17. Lu, C.-Y., K.-Y. Hsieh and R. Liu, "Future Challenges of Flash Memory Technologies", *Microelectronic Engineering*, Vol. 86, No. 3, pp. 283–286, 2009.

18. Member, K., Y. Nonmember, S. Member and N. Nonmember, "Tunnel Oxide Breakdown Characteristics of Floating Gate Type EEPROM", *Electronics and Communications in Japan (Part II: Electronics)*, Vol. 72, No. 10, pp. 1–8, 2007.

19. Witters, J., G. Groeseneken and H. Maes, "Degradation of Tunnel-Oxide Floating-Gate EEPROM Devices and the Correlation with High Field-Current-Induced Degradation of Thin Gate Oxides", *IEEE Transactions on Electron Devices*, Vol. 36, No. 9, pp. 1663–1682, 1989.

20. Caramia, M., S. Di Carlo, M. Fabiano and P. Prinetto, "Flash-Memories in Space Applications: Trends and Challenges", *IEEE 7th East-West Design & Test Symposium EWDTS09*, Moscow, RU, 2009.

21. Shinde Pratibha, M., "Efficient Flash Translation Layer for Flash Memory", *International Journal of Scientific and Research Publications*, Vol. 3, No. 4, pp. 35–40, 2013.

22. Yan Li, T. M., Yupin Fong, "Flash File System Optimized for Page-Mode Flash Technologies", U.S. Patent 7,057,939, 14 September 2006, `https://patents.google.com/patent/US20060203561`, accessed on December 29, 2022.

23. Kwon, S., A. Ranjitkar, Y.-B. Ko and T.-S. Chung, "FTL Algorithms for NAND-Type Flash Memories", *Design Automation for Embedded Systems*, Vol. 15, No. 3, pp. 191–224, 2011.

24. Chung, T.-S., D.-J. Park, S. Park, D.-H. Lee, S. W. Lee and H.-J. Song, "A Survey of Flash Translation Layer", *Journal of Systems Architecture*, Vol. 55, No. 5, pp. 332–343, 2009.

25. Deng, Y. and J. Zhou, "Architectures and Optimization Methods of Flash Memory based Storage Systems", *Journal of Systems Architecture - Embedded Systems Design*, Vol. 57, No. 1, pp. 214–227, 2011.

26. Ban, A., "Flash File System", U.S. Patent 027,131, 4 April 1995, `https://patents.google.com/patent/US5404485A/en`, accessed on December

29, 2022.

27. Chiang, M.-L., P. C. H. Lee and R.-C. Chang, "Using Data Clustering to Improve Cleaing Performance for Flash Memory", *Software: Practice and Experience*, Vol. 29, No. 3, pp. 267–290, 1999.

28. Ma, D., J. Feng and G. Li, "LazyFTL: A Page-Level Flash Translation Layer Optimized for NAND Flash Memory", *ACM SIGMOD Conference*, Beijing, P.R. China, 2011.

29. Park, D., B. K. Debnath and D. H.-C. Du, "CFTL: A Convertible Flash Translation Layer Adaptive to Data Access Patterns", *Measurement and Modeling of Computer Systems*, Minneapolis, USA, 2010.