University of Alabama in Huntsville

LOUIS

Theses

UAH Electronic Theses and Dissertations

2026

NANDWorks : a low-cost testbed for interfacing ONFI-compatible NAND flash memory devices

Mark Scheidker

Follow this and additional works at: https://louis.uah.edu/uah-theses

Recommended Citation

Scheidker, Mark, "NANDWorks : a low-cost testbed for interfacing ONFI-compatible NAND flash memory devices" (2026). *Theses*. 813.
https://louis.uah.edu/uah-theses/813

This Thesis is brought to you for free and open access by the UAH Electronic Theses and Dissertations at LOUIS. It has been accepted for inclusion in Theses by an authorized administrator of LOUIS.

NANDWORKS: A LOW-COST TESTBED FOR INTERFACING ONFI-COMPATIBLE NAND FLASH MEMORY DEVICES

Mark Scheidker

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in
The Department of Electrical and Computer Engineering
to
The Graduate School
of
The University of Alabama in Huntsville

May 2026

Approved by:

Dr. Aleksandar Milenković, Research Advisor/Committee Chair
Dr. David Coe, Committee Member
Dr. Rahul Bhadani, Committee Member
Dr. Aleksandar Milenković, Department Chair
Dr. Keith Hollingsworth, Interim College Dean
Dr. Jon Hakkila, Graduate Dean

Abstract

NANDWORKS: A LOW-COST TESTBED FOR INTERFACING ONFI-COMPATIBLE NAND FLASH MEMORY DEVICES

Mark Scheidker

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering

Electrical and Computer Engineering

The University of Alabama in Huntsville

May 2026

NAND flash memory is the basis for modern digital storage, but direct access to raw flash functionality remains difficult due to limitations in available interface tools. Industrial characterization systems can issue low-level commands but are prohibitively expensive for most researchers and students. USB-based bit-banging tools lack the timing fidelity needed for certain experiments. This thesis introduces NANDWorks, a testbed based on the Raspberry Pi 4 that provides direct access to ONFI-compatible NAND flash devices via GPIO bit-banging in Linux user space, building on prior work for the Terasic DE1-SoC platform. NANDWorks has a layered software stack that includes a hardware abstraction layer, ONFI command sequencing, a C++ API, a command-line driver, and an embedded LuaJIT scripting interface, and can be assembled from low-cost, readily available components. Evaluation included a structured test suite, performance profiling, and logic analyzer captures. Experiments on real NAND devices demonstrate the capability of NANDWorks without specialized equipment.

I have followed the approved guidelines of my graduate program and the UAH Graduate School for the use of generative Artificial Intelligence (AI) in the writing of this thesis. The following generative AI tools were used for the associated purposes listed after each tool:

- ChatGPT: Used for limited brainstorming, outline refinement, language editing, and formatting assistance during thesis writing, with any AI-assisted output independently verified by me.
- Codex: Used for limited code debugging, syntax suggestions, and codebase review and comprehension support during software development related to the thesis, with any AI-assisted output independently verified by me.
- Prism: Used for proofreading, identifying typos, and correcting LaTeX formatting issues during thesis preparation, with any AI-assisted output independently verified by me. I accept full responsibility for this work.

Acknowledgements

I would first and foremost like to thank my professor and mentor, Dr. Aleksandar Milenković, for providing me with the guidance, tools, and motivation to undertake this research. I am sincerely grateful for his continuous encouragement throughout the development of this thesis. I would also like to acknowledge everyone in the LaCASA Lab for sharing their expertise and offering feedback during the research process, especially Vojin Dzeletovic, Sayan Samanta, and Jackson Lawrence. In addition, I am grateful to the many professors in the UAH ECE department whose instruction and guidance prepared me for this work, especially the other members of my thesis committee, Dr. David Coe and Dr. Rahul Bhadani, for their feedback on this manuscript. Finally, I would like to thank my friends and loved ones for their patience, encouragement, and unwavering support throughout this process. I could not have completed this work without them.

Table of Contents

Abstract . . . . . **ii**

Acknowledgements . . . . . **iv**

Table of Contents . . . . . **viii**

List of Figures . . . . . **ix**

List of Tables . . . . . **xii**

Chapter 1. Introduction **1**

1.1 Need for Low-level Access 2

1.2 Existing Approaches and Gaps 3

1.3 NANDWorks Overview 5

1.4 Contributions 5

1.5 Thesis Organization 7

Chapter 2. Background **8**

2.1 NAND Flash Fundamentals 8

2.2 NAND String Architecture and Peripheral Circuits 15

2.3 Disturb and Interference Mechanisms 19

2.4 Page Structure and Controller Abstraction 21

Chapter 3. Open NAND Flash Interface (ONFI) 24

3.1 ONFI Overview 24

3.2 Parameter Page and Discovery 26

3.3 Timing Modes and Interfaces 27

Chapter 4. Hardware Platform 31

4.1 Platform Selection 31

4.2 Adapters, Wiring, and Practical Setup 33

4.3 Logic Analyzer 35

Chapter 5. Software 38

5.1 ONFI Plus 38

5.2 Development of NANDWorks 42

5.3 NANDWorks Architecture 42

5.3.1 User-facing Interfaces and Device Layer 43

5.3.2 Command-line and Scripting Front Ends 44

5.3.3 Controller and Transport Layers 46

5.3.4 Hardware Abstraction Layer 47

5.3.5 Research-oriented Toolkit Features 49

Chapter 6. Validation Testing 53

6.1 Validation Strategy 53

6.2 C++ Test Application (tester.cpp) 54

6.2.1 Bring-up Tests 55

6.2.2 Data-Path Correctness Tests 55

6.2.3 Spare/Read Consistency Tests 56

6.2.4 Data Integrity Tests 57

6.2.5 Edge-case Robustness Checks 58

6.3 Profiling Tools (profiler.cpp / profiler.lua) 59

6.4 Manual Verification Using a Logic Analyzer 62

6.4.1 Simple GPIO Pin Test 62

6.4.2 Visualization of Basic Operations 63

6.4.3 OS Interrupt Profiling and Mitigation 67

Chapter 7. Experiments 70

7.1 Experiment 1: High/Low Bit Page Analysis 70

7.1.1 High/low Page Latency Analysis 71

7.1.2 Reverse Program Order Analysis 72

7.2 Experiment 2: Partial Program/Erase Operations 73

7.2.1 Visualizing Partial Operations 76

7.3 Experiment 3: Energy Saving With Partial Erase 78

Chapter 8. Concluding Remarks 82

8.1 Summary of Contributions 83

8.2 Limitations 85

8.3 Future Work 86

References 88

Appendix A 92

List of Figures

2.1 Simplified floating-gate NAND flash cell cross-section showing the control gate (wordline), charge storage element, and tunnel dielectric that enables charge transfer during program/erase. 9

2.2 Electrical abstraction of a NAND cell as a MOSFET-like device with a control gate (CG) and charge storage node (FG). Reads apply a control-gate bias and sense the resulting drain-to-source current. 10

2.3 Illustration of NAND threshold-voltage distributions, read reference voltages, and program/erase state movement. 10

2.4 Illustration of MLC NAND program-and-verify behavior, showing the erased state, temporary LSB-programmed state, final MSB-programmed states, and associated verify reference voltages. . . . 11

2.5 Simplified NAND array wiring showing series-connected strings, wordlines (WL), bitlines (BL), and select lines. A page corresponds to the set of cells sharing a wordline across bitlines. 13

2.6 Conceptual organization and datapath: pages grouped into blocks (erase granularity) are transferred through internal registers/latches to the external DQ bus (*e.g.*, DQ[7:0]). 15

2.7 Simplified view of a managed NAND storage system showing the flash controller, Flash Translation Layer (FTL), NAND interface, and internal NAND hierarchy. The FTL typically handles logical-to-physical address translation, garbage collection, wear leveling, error correction, and bad block management. 23

3.1 Generic controller-visible asynchronous ONFI operation sequencing for page read, page program, and block erase. 29

4.1 Diagram showing the pinout of a Raspberry Pi 4's GPIO (reproduced from: [26] under CC BY-SA 4.0.) 33

4.2 Photograph of the NANDWorks hardware platform, consisting of a Raspberry Pi 4, TSOP-48 NAND adapter, and GPIO wiring; the logic analyzer shown is optional and not required for operation. . 35

5.1 Layered software architecture of NANDWorks. 43

6.1 Logic analyzer capture of GPIO pin test. 63

6.2 Logic analyzer capture of a demonstration erase/program/read/erase sequence. 64

6.3 Logic analyzer capture of initial NANDWorks handshake followed by a block erase operation and the initiation of a page write operation. 65

6.4 Logic analyzer capture of latency between the end of an erase operation and the start of a program operation. 66

6.5 Logic analyzer capture of the tail end of a write operation followed by a read operation on the same page. 67

6.6 Logic analyzer capture of an interrupt in the middle of a program operation data transfer phase. 68

7.1 Read latency for each page in a block of a Micron 3D MLC NAND device. 71

7.2 Forward/reverse programming RBER in a Micron 3D MLC NAND device. 73

7.3 Results of partial program experiment showing the relationship between number of correctly programmed bits and the duration of the operation before interrupt. 74

7.4 Results of partial erase experiment showing the relationship between the number of correctly erased bits and the duration of the operation before interrupt. 75

7.5 Results of experiment visualizing bit errors in an image at various partial program durations. 77

7.6 Results of experiment visualizing erased information as an image at various partial erase durations. 77

7.7 Joulescope capture of full erase operation performed using NAND-Works CLI. 79

7.8 Joulescope capture of 1000 microsecond partial erase operation performed using NANDWorks CLI. 80

7.9 Joulescope capture of 300 microsecond partial erase operation performed using NANDWorks CLI. 81

A.1 Structural overview of the ONFI interface layer, showing how onfi_interface bridges the hardware abstraction layer, transport contract, controller, and higher-level device workflows. 92

A.2 Overview of program_order_bit_errors.cpp, showing forward-order and reverse-order page programming with timed ONFI operations and per-page bit-error comparison. 93

A.3 Overview of partial_margin_sweep.cpp, showing how onfi_interface applies partial program and partial erase operations while NandDevice reads data back for mismatch analysis. 94

A.4 Overview of partial_binary_io.cpp, showing how binary input data is written with a partial NAND operation and then read back to an output file. 95

List of Tables

4.1 NAND interface signal mapping to BCM GPIO pins. 33

6.1 Performance benchmark results for the Lua and C++ implementations. 60

Chapter 1. Introduction

NAND flash memory is the dominant non-volatile storage technology in modern computing systems and is widely used in low-power embedded devices, smartphones, consumer solid-state drives, and enterprise data centers [10, 6]. Its use has continued to expand due to its low cost, high density, and energy efficiency, driven by ongoing advances in multi-level cell technologies, along with vertical 3D stacking [6].

These density improvements, however, introduce significant reliability challenges, including retention loss, read disturb, program interference, and wear caused by repeated program/erase cycles [5, 4]. As more bits are stored in each memory cell, these effects become increasingly pronounced, making it essential to experimentally characterize the behavior and limitations of NAND devices [5, 6].

Experimental access to NAND devices has enabled a wide range of research directions in recent years. Prior studies have explored techniques for improving flash reliability and analyzing physical effects such as layer-to-layer endurance variation and threshold-voltage distributions in 3D NAND devices [19, 17, 18]. Experimental platforms have also supported research in storage security, including secure data sanitization, instant data deletion through page-level overwrite mechanisms, data hiding, device fingerprinting, and other security primitives based

on intrinsic flash-device characteristics [2, 16, 18, 22, 24, 23]. These research efforts rely heavily on the ability to directly interact with NAND devices at the command and timing level, highlighting the importance of experimental platforms that provide flexible low-level access to flash hardware.

Despite this need, the current landscape of platforms that provide low-level access to NAND devices remains limited, and many existing options present significant cost and usability barriers. Consequently, there is a clear need for a low-cost and accessible platform that enables precise low-level control of NAND flash devices for research and educational purposes.

1.1 Need for Low-level Access

Understanding the physical properties of NAND flash requires direct access to devices in order to conduct experiments that quantify low-level behavior [5, 4, 27]. Such experiments may involve measuring data retention and degradation over time, evaluating read-disturb effects across pages, or intentionally deviating from a device's intended operating conditions to gain deeper insight into its internal architecture [10, 27].

These experiments cannot typically be performed through conventional flash controllers, which abstract the operation of raw NAND devices and restrict access to low-level commands. Researchers instead require the ability to issue raw NAND commands [13], control timing parameters, modify voltage margins, and manipulate the order of operations in order to construct repeatable test sequences that isolate specific physical effects.

This level of control is essential not only for researching topics such as error correction, wear-leveling strategies, and modeling device lifetime [5], but also for educating students on topics in computer architecture, embedded systems, and storage technologies.

1.2 Existing Approaches and Gaps

Commercial platforms such as the NplusT NanoCycler and the Advantest T5200 provide powerful capabilities for NAND characterization and production-scale memory testing [11, 1]. However, these systems are designed primarily for industrial environments and are prohibitively expensive for most researchers and academic institutions. Their primary customer base consists of memory manufacturers that require high-performance validation equipment and are willing to invest substantial resources to obtain it. As a result, these platforms function more as specialized validation systems than as accessible experimentation tools.

At the opposite end of the spectrum is the DumpFlash tool developed by Jeong Wook Oh, which uses an inexpensive FTDI USB interface and Python software to bit-bang signals to and from a NAND device [12]. When wired to an adapter and connected to a NAND chip, the platform allows experiments to be orchestrated through a Python library. While convenient and inexpensive, this approach has several limitations. USB communication introduces additional latency between the host system and the NAND device, and the interpreted Python execution environment further reduces performance. Consequently, DumpFlash is not well suited for experiments that require precise timing measurements

or minimal command latency, which may reduce experimental precision and repeatability.

FPGA-based platforms provide a compromise between these two extremes by offering significantly better timing control than USB-based tools while remaining less expensive than commercial characterization systems. One example is Ming II, developed by the Non-Volatile Systems Lab at the University of California, San Diego [3]. Ming II is a sophisticated NAND experimentation platform that has enabled numerous research studies in non-volatile memory systems. However, the platform is not openly distributed and requires licensing in order to obtain the hardware and associated software, which limits its accessibility.

OpenSSD is another FPGA-based research platform. Although its hardware is commercially available and its software ecosystem is relatively open, the platform is primarily intended for experimentation with flash controller architectures rather than direct characterization of raw NAND devices [7].

At the University of Alabama in Huntsville, a previous effort by Prawar Poudel developed an experimental framework using the Terasic DE1-SoC development board [15]. While this work served as a starting point for the system presented in this thesis, the platform had several limitations, including a higher hardware cost, limited flexibility, and a limited set of implemented features. These constraints limited its usefulness as a general-purpose low-cost NAND experimentation platform and motivated further development.

1.3 NANDWorks Overview

This thesis presents NANDWorks, a NAND experimentation platform designed to bridge the gap between cost, accessibility, and experimental capability by implementing a flexible NAND interface using a Raspberry Pi controller. The platform uses a software-controlled bit-banging approach to transmit and receive signals through the Raspberry Pi's general-purpose input/output (GPIO) pins [9].

This approach provides a practical middle ground between FPGA-based systems and USB-based tools. Direct software access to GPIO enables significantly lower communication latency than USB-based platforms, while avoiding the cost and complexity associated with FPGA development.

NANDWorks builds upon prior work developed on the DE1-SoC platform but extends it substantially by introducing a more accessible hardware platform and a richer software environment. The system supports rapid experiment development through multiple interfaces, including Lua scripting, a command-line driver, and a C++ programming interface. The NANDWorks repository can be cloned and compiled directly on a Raspberry Pi, enabling reproducible research workflows using inexpensive and widely available hardware.

1.4 Contributions

The primary contributions of this work are summarized as follows:

- **Low-cost NAND hardware interface.** A hardware interface is developed using a Raspberry Pi 4 that exposes an asynchronous ONFI-compatible

signal interface to raw NAND flash devices. The system uses GPIO wiring and commercially available adapters to provide direct electrical connectivity to NAND devices while maintaining a low overall hardware cost.

- **Scriptable user space experimentation framework.** A modular software stack is implemented in Linux user space that exposes fundamental ONFI operations, including read, program, erase, status, and reset, through multiple interfaces. These include a command-line driver, a C++ programming interface, and an embedded LuaJIT scripting environment that enables rapid development of experimental workflows.

- **Protocol validation and performance characterization.** The correctness of the implemented ONFI protocol is validated using logic analyzer traces of NAND command sequences. System overhead is characterized through microbenchmarking experiments that compare Lua-driven control to native C++ execution when interacting with a Micron MLC NAND device.

- **Demonstration of reproducible NAND experiments.** The platform is used to construct several representative experiments that demonstrate its usefulness for device characterization. These include device validation through bad-block scanning, lightweight timing analysis through average block read latency measurements, and 3D MLC characterization using per-page read-latency mapping and out-of-order programming bit-error analysis.

Additional experiments characterize the effects of partial operations on data integrity and energy consumption.

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 presents background information on NAND flash memory and explains the underlying device technology. Chapter 3 describes the Open NAND Flash Interface (ONFI), the communication standard used to interact with NAND devices. Chapter 4 details the hardware setup used in NANDWorks, including the design decisions that led to the final configuration and the resulting implementation. Chapter 5 describes the evolution of the software stack from ONFI Plus to NANDWorks, including the refactoring process, layered architecture, and user-facing interfaces used to support research-oriented experimentation. Chapter 6 presents the validation methodology used to verify NANDWorks functionality and evaluate its performance. Chapter 7 demonstrates the platform through a series of experiments, including MLC characterization and partial-operation testing. Finally, Chapter 8 concludes the thesis and discusses directions for future work.

Chapter 2. Background

This chapter introduces the technical background needed for the rest of this work. It covers NAND flash fundamentals, beginning with cell operation, threshold-voltage distributions, and read/program/erase behavior. It then covers key reliability mechanisms, including wear, retention loss, and disturb, and relates these effects to system-level error trends. Finally, it introduces the architectural constraints needed to discuss the NAND interface and memory characterization.

2.1 NAND Flash Fundamentals

The smallest unit of NAND flash is called a cell, which is a modified Metal Oxide Semiconductor Field Effect Transistor (MOSFET) with a few key changes. While a standard MOSFET has a single gate, a NAND cell adds a charge storage layer, in the form of a conductive floating gate for 2D NAND, and more complex charge trapping structures in modern 3D NAND, which are separated from the channel and control gate using insulating dielectric layers. These changes introduce a characteristic where high voltages on the control gate result in a strong electric field in the thin tunnel oxide and cause electrons from the channel to tunnel into the floating gate. This enables long-term storage of charge and is the basic principle used for the storage of data in a NAND cell [10, 6]. The amount of stored

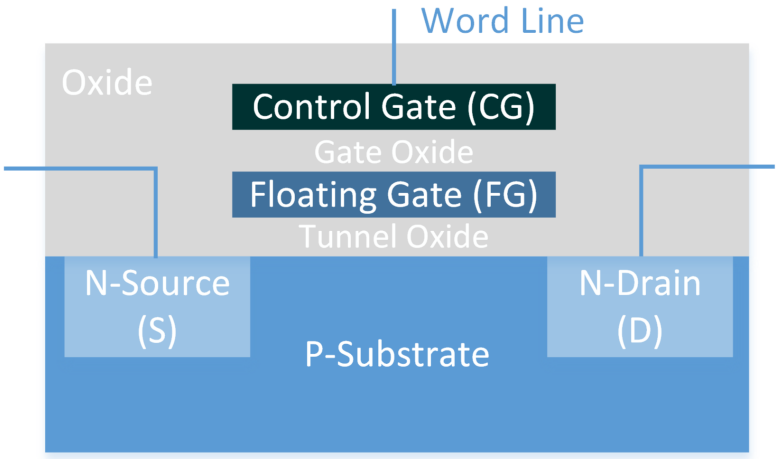


Figure 2.1: Simplified floating-gate NAND flash cell cross-section showing the control gate (wordline), charge storage element, and tunnel dielectric that enables charge transfer during program/erase.

charge shifts the cell's threshold voltage, which changes how easily current can flow between the source and drain for a given control-gate voltage. This allows the stored state to be read by applying a read voltage to the control gate, biasing the source and drain, and measuring the resulting current. Charge storage within the floating gate is continuous, meaning there can be more than two distinct charge states. Originally, NAND technology used Single-Level Cell (SLC) operation, which stores two distinct charge levels per cell; Multi-Level Cell (MLC) technology soon followed, allowing two bits to be stored per cell by storing four distinct charge levels. Modern NAND flash implements Triple-Level Cell (TLC) with eight distinct charge levels and even Quad-Level Cell (QLC) with 16 levels [10, 6].

Figure 2.1 shows a simplified floating-gate NAND flash cell. The control gate, which is driven by the wordline, is capacitively coupled to the charge storage element through the dielectric stack, and an insulating thin tunnel dielectric layer

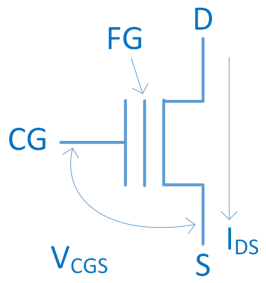


Figure 2.2: Electrical abstraction of a NAND cell as a MOSFET-like device with a control gate (CG) and charge storage node (FG). Reads apply a control-gate bias and sense the resulting drain-to-source current.

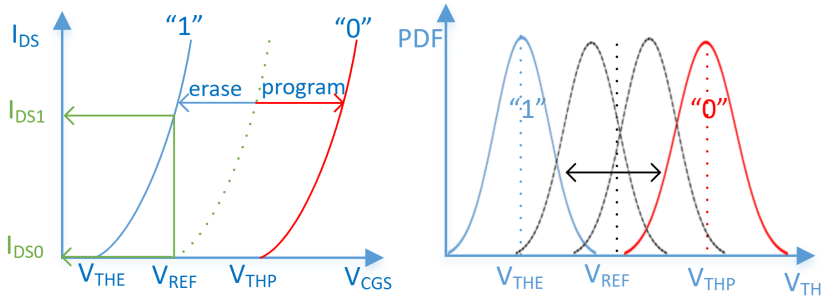


Figure 2.3: Illustration of NAND threshold-voltage distributions, read reference voltages, and program/erase state movement.

provides separation between the storage node and the channel region. For circuit-level discussion, the same structure can be abstracted using the two-gate transistor symbol shown in Figure 2.2, with the charge shifting the voltage threshold, which affects the current seen during read operations.

NAND devices store data at specific voltage intervals called *threshold voltages*. Instead of using a specific and deterministic value of threshold voltage for a given state, each programmed level has an associated *distribution* of threshold voltage levels across many cells, as shown in Figure 2.3. As a result, reads are

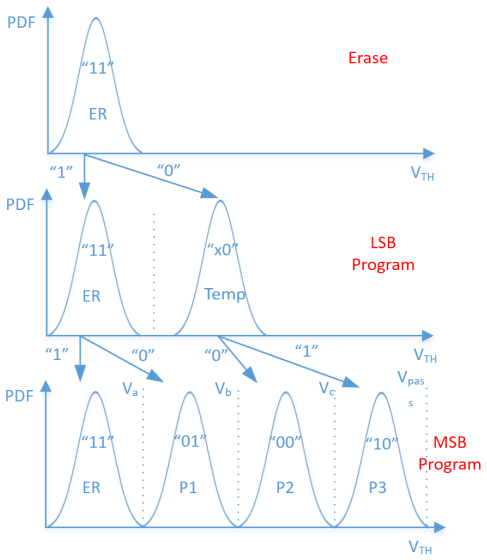


Figure 2.4: Illustration of MLC NAND program-and-verify behavior, showing the erased state, temporary LSB-programmed state, final MSB-programmed states, and associated verify reference voltages.

performed by comparing the cell's threshold voltage against reference voltages, which are used to decide which distribution a cell most likely belongs to. As the number of bits per cell increases, the boundaries between adjacent decision regions become closer together. As such, there is little room between the distributions of neighboring charge states, and ultimately, throughout the life of all cells, these distributions shift and spread due to wear, retention effects, and disturb mechanisms, which increases the probability of errors [5].

Writing data to a cell requires it to be driven by high electric fields across the tunnel dielectric; however, modern NAND does not program a target threshold voltage in a single step. Instead, programming is typically performed using an iterative *program-and-verify* process. Figure 2.4 illustrates the staged LSB/MSB

programming behavior of MLC NAND and the associated *verify* reference voltages. During a write operation, the NAND IC applies a series of short programming pulses to the selected wordline. After each pulse, the IC has an internal *verify* step where each cell is compared to one or more reference voltage levels. Cells which are at the desired threshold window are prevented from further programming, through *program inhibit* biasing, and cells which are not yet programmed to the desired voltage will be subjected to more pulses [10, 5]. This incremental approach is an important mechanism which enables MLC, TLC, and QLC to function, as the increase in the number of stored charge levels requires increased accuracy of the threshold voltages. This approach results in additional latency due to an increase in the number of required programming pulses and program verification checks, and this combination of incremental programming and program inhibit behavior is a major reason why writing to NAND is fundamentally slower and more complex than reading [10].

In addition to increased data density using multiple charge levels per cell, NAND is highly compact due to the layout of its wiring. Unlike NOR flash, NAND flash connects cells in series to form strings, which reduces routing complexity and improves density. As shown in Figure 2.5, the control gates of cells are tied together across strings by *wordlines* (WL), while *bitlines* (BL) run orthogonally and connect strings to peripheral sensing and buffering circuitry. This organization naturally defines a *page* as the population of cells sharing a single wordline across many bitlines, enabling page-granular read and program operations [10].

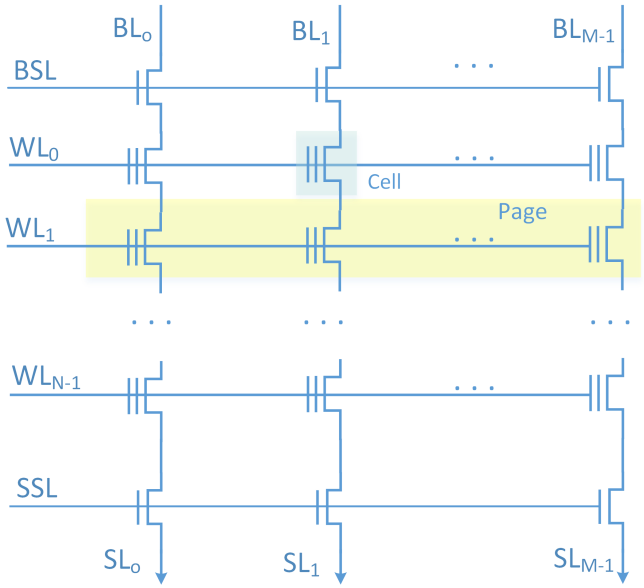


Figure 2.5: Simplified NAND array wiring showing series-connected strings, wordlines (WL), bitlines (BL), and select lines. A page corresponds to the set of cells sharing a wordline across bitlines.

Increasing density in NAND ICs is a balancing act between maintaining high manufacturing yields and continuing to miniaturize cells and their wiring on the silicon substrate, while also maintaining enough reliability to store multiple distinct charge levels in each cell. Planar NAND struggles with interference and cell wear as features continue to shrink, so higher capacity devices were either more expensive due to increased silicon area, or less reliable due to decreased feature size. To get around these limitations, manufacturers have needed to scale using 3D or vertical V-NAND, which is the underlying technology behind nearly all modern high density flash memory. 3D NAND decouples density from feature size, allowing for more cells per unit area on a silicon die while continuing to use larger and more robust cells [6, 27].

These improvements, however, come at the cost of increased manufacturing complexity and introduce new challenges. In order to reduce the number of manufacturing steps, 3D NAND is typically not manufactured one layer at a time; rather, a set of processing steps is performed on the vertical stack all at once. Holes are punched into alternating layers of polysilicon and oxide, and materials are then deposited into these holes in order to "grow" the necessary channel, dielectric, and charge-trapping materials, forming a vertical string of cells. Due to this manufacturing process, cells in a stack are not perfectly uniform, and cells near the top of the stack can behave differently from cells near the bottom [19]. Manufacturing variations in the vertical direction can result in different threshold distributions, retention behavior, and program/erase characteristics [6, 27]. Mitigation strategies at the device and controller levels are then employed to maintain usable margin throughout the entire stack. In most 3D NAND architectures, each physical wordline represents one layer in the stack, which means each page consists of cells with similar physical characteristics, simplifying management of layer-dependent behavior in higher-level firmware and controllers. While this page configuration addresses cell non-uniformity in 3D NAND, there are still fundamental challenges with operating on series-connected NAND cells which are still present in older planar NAND, and need to be managed through sequencing constraints and controller policy [6, 27].

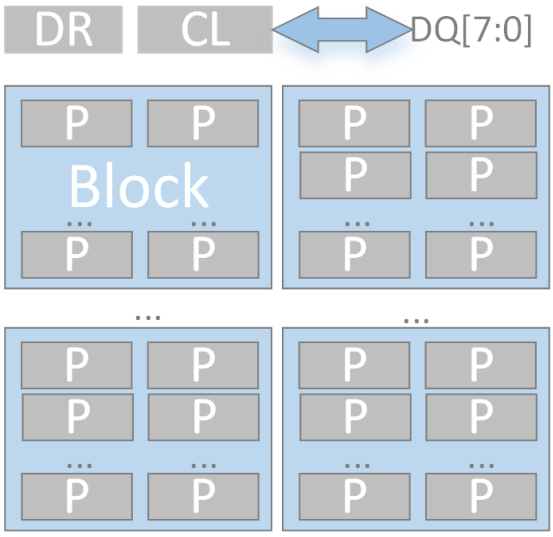


Figure 2.6: Conceptual organization and datapath: pages grouped into blocks (erase granularity) are transferred through internal registers/latches to the external DQ bus (*e.g.*, DQ[7:0]).

2.2 NAND String Architecture and Peripheral Circuits

Each NAND string includes additional select transistors at its ends to connect or isolate the series-connected cells from the peripheral bitline and source line circuitry. In typical organizations, NAND strings are connected to a bitline with voltage-sensing circuitry, while the other end is connected to a common source line. On-die peripherals are used for functions such as pre-charging and sensing bitline behavior during read operations, as well as generating and distributing the elevated voltages required during program and erase operations [10].

In addition to its unique underlying structure, the presence of internal page buffers and charge pumps separates NAND flash from alternative non-volatile solid-state memory technologies. A page program operation first loads data

into an internal buffer via the external bus, then runs an internal programming sequence that is independent of the host. Similarly, a page read operation moves a page into an internal buffer before sending it out over the bus. Because of this, NAND latency includes both bus transfer time and internal array time, and the impact of each factor depends on the interface speed, page size, and whether any extra internal steps, like multiple sensing passes in multi-level NAND, are needed. Figure 2.6 shows a simplified view of this data path. A full page is first sensed or assembled into internal registers and latches, then sent over the external DQ bus, with pages grouped into erase blocks [10].

Though pages and blocks are the minimum read/program and erase granularity, NAND ICs are often organized into additional hierarchy levels. Multiple planes can exist in a die, and multiple dies or logical units, often represented as independent internal targets, can exist in a package. Such organization allows interleaving and concurrency, and one or more units can be busy performing internal operations while another unit transfers data on the bus. As a result, measured bandwidth and latency often depend on how operations are distributed across these internal units [10, 13].

Read operations present a unique challenge due to the cells being connected in series. Determining the state of a single cell's floating gate requires a pass-through voltage to be applied to the wordline of each unselected cell so that each one ignores the state of its floating gate and behaves like a closed switch. A reference voltage is then applied to the wordline of the selected cell, and depending on the state of charge in its floating gate, the cell will either pass or block current

through the bitline to the source line. Adjusting this reference voltage and reading more than once allows for the measurement of multiple charge states in the floating gate, making MLC, TLC, and QLC possible. This repeated pass-through biasing also contributes to *read disturb*, discussed below [4, 5]. In multi-level devices, the IC executes multiple sensing operations with different reference voltages in order to discern which of several threshold voltage windows a cell occupies. These reference voltages define the decision boundaries between adjacent states. Due to the fact that these thresholds shift over time from retention loss and wear, many devices and controllers support adaptive read mechanisms, where reference points can be dynamically adjusted in order to recover margin as the device ages. The process of making such adjustments is often referred to as *read-retry* or simply as reference voltage tuning, which trades additional sensing time for better accuracy [8, 5].

Beyond the read path, the process of erasing cells introduces one of the major design constraints in NAND flash architecture. Erasing requires removal of charge from the floating gate, which is achieved by applying a high voltage to the p-substrate of a cell and grounding its wordline. To do this, the high voltage must be applied to the underlying substrate of the cells, which means erasing not just all cells in a string, but all strings sharing the same substrate. The result of this is that cells cannot be erased individually; rather, they are erased in large, electrically isolated blocks. Erase operations are also iterative. Rather than using one long erase pulse, one or more erase pulses are used, along with internal erase verify checks which determine if the cells have been successfully restored to

the erase threshold range. If erase verify fails, additional pulses may be applied until the erase operation is complete. Since erase is block granular and impacts device wear directly, both erase frequency and erase distribution are important contributors to device lifetime [10, 5].

The asymmetric nature of programming and erasing cells necessitates separate minimum sizes for these operations, called blocks and pages. Blocks are electrically isolated groups of cells with a shared substrate and are the smallest unit that can be erased. Each block contains many pages, which are the smallest unit that can be programmed or read. Managing storage on a NAND device is therefore more involved than other technologies, because reclaiming space requires all valid pages be moved and rewritten elsewhere before the block can be erased. These architectural asymmetries place constraints on how NAND can be interfaced with, which is why there are specific flash command and addressing standards unlike regular memory interfaces [10, 13].

At the page level, there are new complexities introduced by multi-level NAND that must be considered. These devices store multiple logical pages as separate bits (such as least significant and most significant bits) in the same memory cells, read using different reference-voltage boundaries on the same physical wordline and its associated cells. There are sequencing constraints and programming schemes imposed by the vendor that minimize interference and maximize reliability by programming the pages in a particular order within a block. This ensures that partially programmed cells are programmed and verified in a controlled manner, which in turn minimizes cell-to-cell interference and helps

keep the threshold-voltage distributions for each bit within the expected margins defined by the read reference levels and ECC [5].

2.3 Disturb and Interference Mechanisms

In addition to wear of the tunnel and dielectric layers over repeated program/erase cycling, NAND reliability is also impacted by mechanisms that shift threshold voltages unintentionally during normal operation. One of these effects comes from the high bias voltage applied to non-selected cells while accessing a selected cell. During reads, the unselected wordlines are biased with a pass-through voltage so that their cells behave like conductors, and allow for reading of the charge state of the selected cell. Similarly, during program operations, the unselected wordlines are biased in order to keep certain cells conductive, while the targeted cells receive the programming pulses. Repeated exposure to these higher bias voltages can create threshold shifts in the unselected cells and lead to read disturb or program disturb [4, 5].

Another type of interference occurs due to electrostatic coupling between cells and adjacent structures. Due to the small size of on-chip wiring and storage elements, threshold voltages may be affected by the charge state and programming activity of their neighboring cells. This electrostatic coupling becomes more apparent as the number of distinct charge levels within cells increases. As the voltage difference between each charge state decreases, the probability that a cell's charge is moved into a new charge state due to nearby activity increases [5].

At the system level, these physical effects can manifest as bit errors when attempting to read data from the device. A standard measure of the extent of this issue is called *Raw Bit Error Rate* (RBER), which measures the fraction of bits read incorrectly from a page before error correction is applied. As NAND cells are subjected to more wear, the RBER will increase due to effects of wear-out from program/erase (P/E) cycling, long-term charge retention loss, and read-disturb from repeated reads [5, 4, 8].

There are two major factors that contribute to this increased overlap of threshold-voltage distributions: retention loss and endurance wear. Retention loss occurs because stored charge is gradually lost or redistributed over time through defect-/trap-assisted leakage mechanisms in the storage node and surrounding dielectric stack, causing threshold voltages to drift. Endurance wear primarily happens as a result of repeated P/E cycles, which increases defect and trap densities in the oxide stack. These defects not only increase the spread of the threshold voltage distribution but also make future programming less precise [5].

In order to make NAND more robust and useful as a long-term storage medium, error correction coding (ECC) is employed. In many cases, the ECC coding operation is applied across fixed-sized codewords within a page, where the size of the ECC codewords is defined by the number of correctable bit errors per codeword (usually normalized across 1 KB or 2 KB of data) [5, 8].

There has been some discussion in the literature regarding the corrected error rate in higher layers as defined by an *uncorrectable bit error rate* (UBER), where the probability of an error remaining within the ECC code when it cannot

be corrected by the ECC code is factored in [8]. This issue shows the importance of reporting performance characteristics (latency, bandwidth), as well as reliability indications like RBER, distribution of corrected bits, and read-retry occurrences in experimental research, as these characteristics determine the practical limits of NAND devices [5, 8].

2.4 Page Structure and Controller Abstraction

A NAND page typically consists of a main data area and an extra region, often referred to as the *spare* or *out-of-band* (OOB) area. The spare area contains metadata for the page, which typically includes error correction information and bookkeeping data for a controller or translation layer. NAND devices also utilize reserved bytes in the spare area to indicate bad blocks. Factory marked bad blocks are expected in NAND and are determined at manufacturing time, while additional blocks can become unusable over the device lifetime due to wear, retention failures, or repeated erase-verify or program-verify failures, and are marked bad during use [10, 5].

ECC handling in NAND-based systems can vary between specific devices and designs, but in conventional raw NAND designs it is typically handled by the controller. Additionally, especially in applications requiring high data integrity and long lifespan, further controller-based ECC and integrity checks may be used. For platforms that access NAND at a raw level, information on the main/spare representations and the location of bad block markers is important, as these can have an impact on how lifetime behavior is characterized [10, 13].

These constraints also motivate flash translation. When communicating over a raw NAND interface, the granularity of access is fixed, with reads and writes happening at the page level, and erases happening at the block level. Because NAND does not support overwriting pages in-place, any updates to the contents of a page must be written to a different page, and space must later be reclaimed by erasing whole blocks. In addition, bit errors are an inherent characteristic of NAND and increase with wear, retention time, and disturb effects. Together, these constraints make exposing raw NAND directly impractical for most systems. Because of this, a NAND controller is employed as a middleman to carry out error correction and bad block management, maintain a Flash Translation Layer (FTL) that translates logical addresses to physical pages, manage out-of-place updates, and reclaim space by moving valid pages to another block prior to a block erase. Figure 2.7 shows a simplified view of these controller and FTL functions within a managed NAND storage system [10].

The next chapter discusses the ONFI command, address, and data interface, which the controllers use to carry out basic operations and observe the status of NAND devices [13].

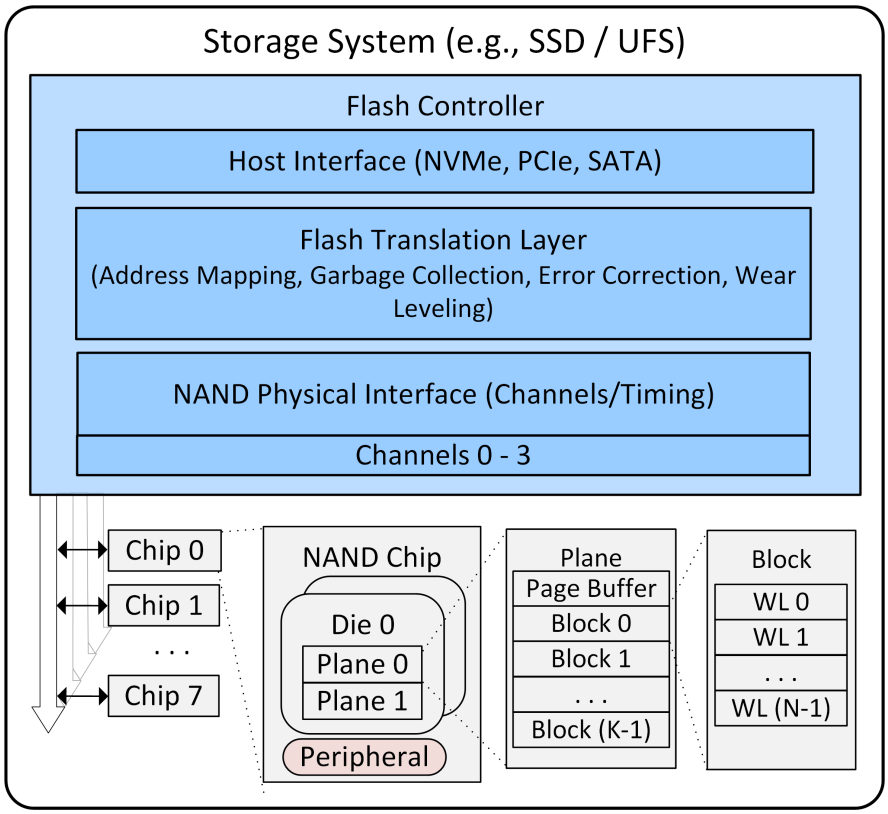


Figure 2.7: Simplified view of a managed NAND storage system showing the flash controller, Flash Translation Layer (FTL), NAND interface, and internal NAND hierarchy. The FTL typically handles logical-to-physical address translation, garbage collection, wear leveling, error correction, and bad block management.

Chapter 3. Open NAND Flash Interface (ONFI)

A system can only benefit from NAND flash if the controller and memory device have a common communication protocol. By standardizing the timing behavior, signaling, and command structure used to access NAND devices, the Open NAND Flash Interface (ONFI) establishes that common language. This chapter presents the fundamental ONFI model, which includes timing-mode configuration, command and address sequencing, and device discovery via the parameter page. Additionally, it examines how ONFI updates have improved interface performance while maintaining a uniform controller-facing programming paradigm [13].

3.1 ONFI Overview

In 2006, the Open NAND Flash Interface Working Group was established as a consortium of NAND manufacturers for the purpose of developing shared open NAND flash standards, and released the first, v1.0, version of the Open NAND Flash Interface standard in December of the same year. ONFI provides a common low-level interface for NAND devices, allowing customers to design products using a shared electrical standard. This allows devices from multiple manufacturers to be interoperable since they have pin-compatible interfaces and

communicate using the same rules. ONFI is not a consumer-oriented standard like SD or CompactFlash cards; rather, it is primarily designed as an interface for flash memory controllers and NAND devices. ONFI treats a NAND device as a single command-driven peripheral where abstracted operations are commanded by a host flash controller, which are then carried out by an internal state machine implemented within the IC. Manufacturers include this state machine in their ICs so that customers can use different ONFI compliant devices in their designs without having to drastically alter their board or software designs. The state machine takes care of the block and page layouts, as well as translating requested operations so they can be executed independently over the raw NAND memory. The end result is that the customer does not have to understand the underlying characteristics of a specific NAND device, and can interface with it using command abstractions. ONFI defines the physical pins, and the commands required to interface with the chip. Some pins are used for control signals to manage the chip, and other pins are used to send commands, addresses, and data on a shared bus. This approach reduces the number of required pins by re-using the same bus for communication in both directions. The core ONFI control and data signals are as follows [13].

- **I/O[7:0] or I/O[15:0]**: An 8 or 16 bit bidirectional data bus that carries command, address, and data bytes between the NAND IC and controller.
- **CLE (Command Latch Enable)**: Asserted when sending command bytes to the IC on the I/O bus.

- **ALE (Address Latch Enable)**: Asserted when sending address information to the IC on the I/O bus.
- **WE# (Write Enable)**: A controller-driven signal to clock data into the chip on the I/O pins.
- **RE# (Read Enable)**: A controller-driven signal to clock data out of the chip on the I/O pins.
- **CE# (Chip Enable)**: Used to select the chip; useful in situations where multiple chips share a common bus.
- **R/B# (Ready/Busy)**: An active-low, chip-driven signal that indicates device status (low = busy, high = ready/complete).

3.2 Parameter Page and Discovery

Given that ONFI supports a variety of timing modes and optional features, initial communication is often started with a conservative discovery sequence. A controller will reset the device, read basic ID information, and read a standard parameter structure that includes information regarding the device's geometry, such as page size, spare area size, and blocks per LUN. The controller uses this information to understand the characteristics of the connected device, and allows ICs with different features to communicate their capabilities, ensuring correct operation [13].

In ONFI, this standardized parameter structure is formalized as the ONFI *parameter page*, obtained via the Read Parameter Page command (ECh) and

beginning with a standardized signature in bytes 0–3 that allows the host system to confirm it has correctly accessed a valid parameter page [13]. The parameter page indicates various identification fields such as the manufacturer and device model strings and date code, as well as operational parameters that the host system must correctly interpret to successfully perform command sequences, including page size, spare area size, pages per block, blocks per Logical Unit Number (LUN), the number of LUNs, the required number of address cycles, and basic array characteristics such as bits per cell and endurance limits [13]. This process is aided further by ONFI flash devices, which contain several redundant copies of this parameter page, each of which has integrity checks that allow the host to verify the information or use another copy if necessary [13]. Later versions of the ONFI specification also define the optional *extended parameter page*, offering capability information in addition to the standard 256-byte parameter page structure [13]. Once the device has been identified, the controller configures its operational features such as timing mode selection, optional cache behavior, and, if possible, enabling of higher-speed DDR interfaces. Before this, however, the initial discovery and configuration rely on a baseline asynchronous command/address protocol so that the controller can safely communicate with the device before calibrations and feature configuration are performed [13].

3.3 Timing Modes and Interfaces

Additionally, ONFI standardizes how a host configures a device's timing modes and supported interfaces. After capability discovery, the host uses the

standard *Set Features* mechanism to configure the device with a valid timing mode by setting a feature address for timing-mode selection [13]. The device indicates busy for the necessary amount of time during a timing change, and higher-speed interfaces might need an extra training or calibration step before the new mode is usable [13]. Thus, instead of using a distinct mode-specific command sequence, timing selection is performed using *Set Features* and can be queried via *Get Features* [13].

The IC interprets bus activity through the combination of a latch enable signal and the toggling of the RE# or WE# pins. For example, to write an address, the controller asserts ALE, places a value on the bus, and pulses WE# to advance the transfer [13]. All ONFI operations must be sequenced by the controller by asserting the correct signals in order to advance the NAND IC through each phase of an operation, and each high-level operation consists of a short command/address sequence followed by a waiting period while the NAND IC completes the requested work. ONFI defines a wide range of commands; however, most ONFI operations are completed using a small set of common operation sequences, consisting of a command phase, an address phase, an internal NAND IC execution phase, and an optional data transfer phase [13]. Figure 3.1 illustrates the generic controller-visible sequencing of the most common asynchronous ONFI operations, including page read, page program, and block erase.

Data transfer is decoupled from the flash cells through the use of an internal page buffer. During a page program, the controller issues a program command along with a page address, then the data is sent across the bus to be stored in

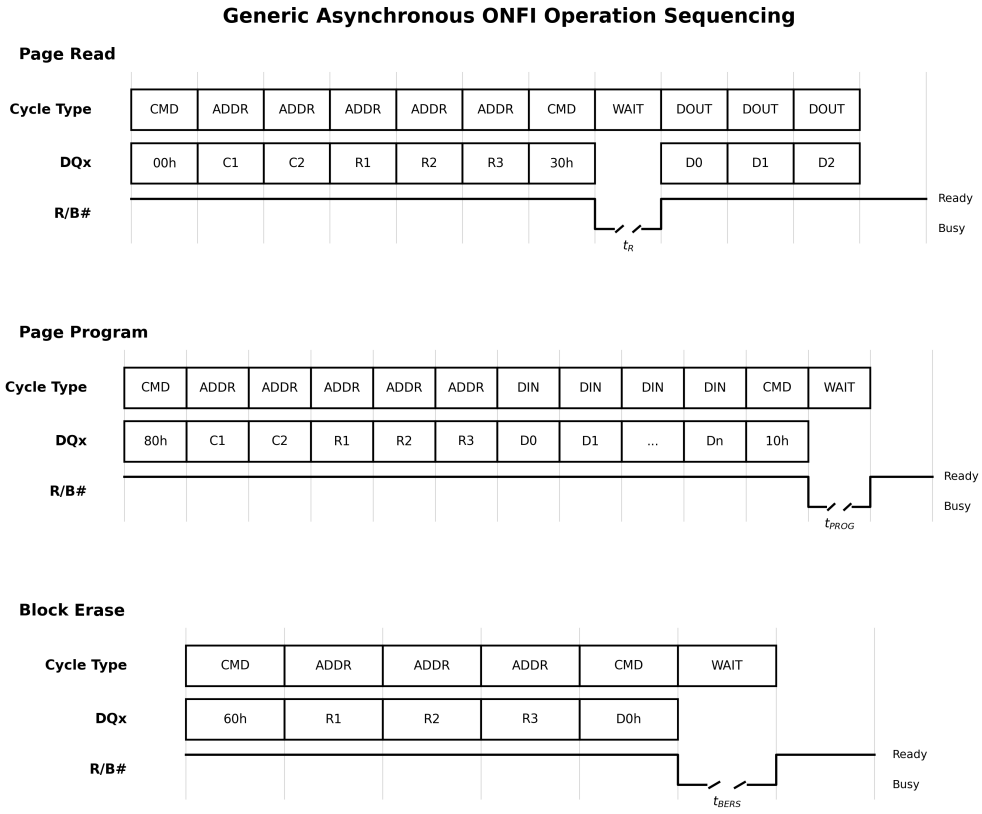


Figure 3.1: Generic controller-visible asynchronous ONFI operation sequencing for page read, page program, and block erase.

the buffer. After the controller sends a confirmation command, the IC will write the data from its internal buffer to the requested page, returning R/B# high (ready) when complete. Reading pages reverses this order, where the IC enters a waiting period to fetch a page from the NAND flash, stores it in the buffer, and returns R/B# high (ready) to notify the controller it may begin transferring data on the bus. Block erase operations are even simpler, as they do not require a data transfer step, so the controller only needs to send the command and block address, and wait for the operation to complete [13]. Therefore, creating an

ONFI compliant controller means creating a device that can reliably generate the command, address, and data sequences within the NAND IC's timing constraints [13].

ONFI has had several major revisions and improvements since its introduction in 2006 as it has continued to adapt to increasing demand for higher transfer speeds and added features [13]. The early revisions used an asynchronous interface and are often quoted at roughly 50 MB/s on an x8 bus [13]. ONFI 2.0 added NV-DDR (often referred to as NV-DDR1) with throughput up to about 133 MT/s (133 MB/s on an x8 bus) [13]. ONFI 3.0 added NV-DDR2 (extended up to 533 MT/s with version 3.2) [13]. Version 4.0 introduced NV-DDR3 with ZQ calibration and a 1.2 V VccQ, and version 4.2 extended NV-DDR3 up to 1600 MT/s [13]. ONFI 5.0 introduced NV-LPDDR4 up to 2400 MT/s, ONFI 5.1 extended NV-LPDDR4 up to 3600 MT/s with additional signal-integrity features, and ONFI 5.2 added protocol refinements such as Separate Command Address (SCA) [13]. The NAND IC may support one or more revisions for backwards compatibility, but devices typically power up in a baseline asynchronous timing mode for discovery and configuration. Higher-speed NV-DDR/NV-LPDDR4 modes require the controller to read the parameter page and apply feature configuration (and, when applicable, perform training or calibration) before enabling the faster interface [13]. Once configured, bulk data transfers can be performed using the fastest mutually supported interface, while the overall ONFI command- and address-sequencing model remains broadly consistent across revisions, with optional enhancements introduced in newer versions [13].

Chapter 4. Hardware Platform

This chapter outlines the physical foundation of NANDWorks, providing a justification for the hardware decisions that allow reliable ONFI communication on a low-cost, reproducible platform. It begins with a discussion of the selection of the systems, justifying the choice of the Raspberry Pi 4 over the original DE1-SoC target system and newer alternatives. It continues with a discussion of the work done in integrating the systems, including the GPIO signal mapping, the choice of the necessary adapters, and the wiring strategy used with the Thin Small Outline Package (TSOP) and Ball Grid Array (BGA) NAND devices. Finally, it concludes with a discussion of the role of external measurement tools, particularly the logic analyzer, in the validation and debugging of the protocol communication.

4.1 Platform Selection

Selecting hardware for this project required several considerations, namely availability, support, ubiquity, and acceptable performance. The original DE1-SoC platform on which this project was based provided a useful starting point, but continuing development on it would have required relying on a relatively uncommon platform. Another issue with using DE1-SoC is that it is a relatively old platform with limited compute capability. Its 800 MHz dual-core ARM Cortex-A9 CPU

and 1 GB of DDR3 RAM are capable of running Linux; however, its software-only performance is limited for this application. DE1-SoC is primarily an education platform designed for experimentation with FPGAs, with a supplemental ARM CPU included to increase its usefulness as a tool for developing software that can interface with custom hardware. Continuing to develop a CPU-only project for this hardware would limit the project's usefulness, and other hardware platforms have the potential to provide a much better alternative.

One candidate platform is a single-board computer from the Raspberry Pi family. Raspberry Pi boards are widely used, readily available, and have a mature software ecosystem. The Raspberry Pi 5 is a natural candidate because it is the newest and most capable model; however, its GPIO architecture changed in a way that reduced compatibility with several older direct-hardware-access GPIO libraries used by existing low-level software. Since this project depends on low-latency, software-controlled GPIO access through a mature and well-understood C/C++ library stack, the project instead adopted the Raspberry Pi 4. This model retains the older and more widely compatible peripheral interface, while still providing a significant increase in compute capability over the DE1-SoC, with a quad-core ARM Cortex-A72 processor and LPDDR4 memory [21].

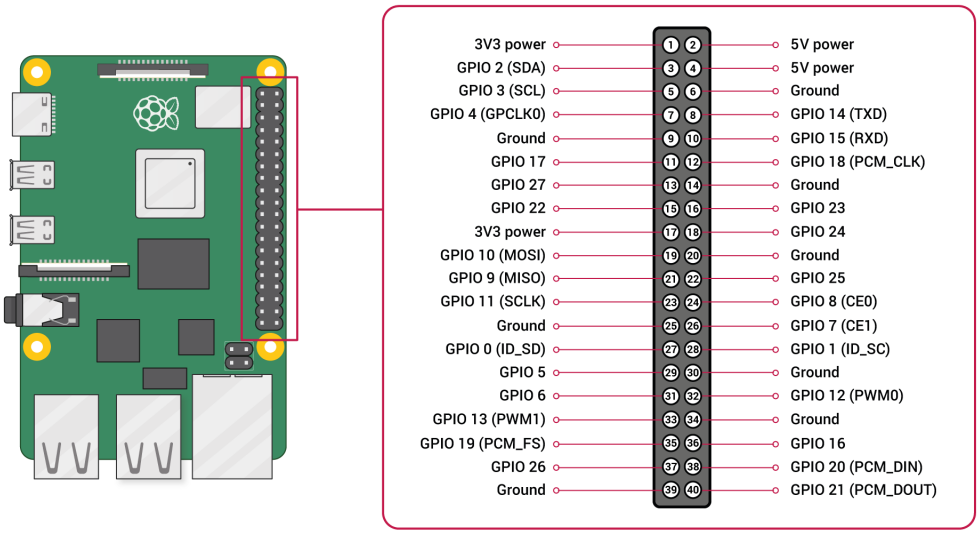


Figure 4.1: Diagram showing the pinout of a Raspberry Pi 4's GPIO (reproduced from: [26] under CC BY-SA 4.0.)

Table 4.1: NAND interface signal mapping to BCM GPIO pins.

Signal	BCM GPIO
DQ0...DQ7	21, 20, 16, 12, 25, 24, 23, 18
WP#	26
WE#	19
ALE	13
CLE	11
CE#	22
RE#	27
R/B#	17

4.2 Adapters, Wiring, and Practical Setup

Deciding which device would run the ONFI interface is only one component of the design, and there were other considerations that needed to be taken into account. One of these design considerations was the pinout for the GPIO wires.

The Raspberry Pi's GPIO pins are not uniform in terms of functionality. Figure 4.1 shows the functionality of each pin on a Raspberry Pi, including pins used for 3.3 V and 5 V power, grounding pins, specialty pins for features like PWM, I2C, or SPI, and some simple GPIO-only pins. The pinout was divided between the left and right half of the physical 40-pin connector, with the control signals on the left, and the data bus on the right, which reduces confusion and improves grouping of physical wires. As shown in Table 4.1, pins were chosen from the bottom to the top, starting at GPIO 21 and ending at GPIO 18. By default on the Raspberry Pi 4, GPIO pins 0–8 use pull-up resistors, whereas pins 9–27 use pull-down resistors; GPIO28 and GPIO29 are exceptions and default to no internal pull resistor [20]. In order to prevent any unintentional start-up behavior, only pins numbered 9 and above were used, meaning pins 1, 7, and 8 were skipped when choosing pins for the data bus. The next set of wiring needed to go to the control signals, and are also shown in Figure 4.1. Like the data bus lines, pins above GPIO 8 were selected for each control signal to avoid these default pull-state issues. Finally, NAND devices rely on continuous power and grounding rails, which are provided by the fixed 3.3 V and ground pins. While the chosen pinout is not fixed and can be changed in software, it provides a reasonable balance between separating control and data wiring onto different rows of GPIO, using suitable pins for the task, and establishing a useful baseline configuration.

The remainder of device hardware setup requires wiring GPIO pins to an appropriate adapter. Both TSOP-48 and BGA-132/152 devices have commodity adapters available on the market, and provide a means to interface with the much

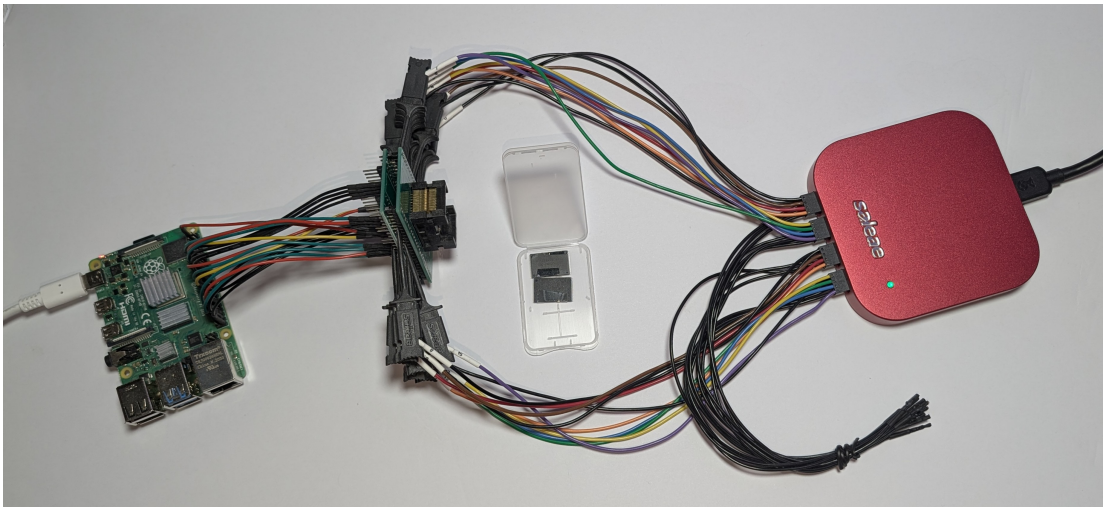


Figure 4.2: Photograph of the NANDWorks hardware platform, consisting of a Raspberry Pi 4, TSOP-48 NAND adapter, and GPIO wiring; the logic analyzer shown is optional and not required for operation.

smaller pins or BGA pads using standard breadboard wires, which is the final required component. The final step is to wire the pins on the Raspberry Pi to the correct pins on the adapter. As an example, the TSOP-48 package used by many NAND devices exposes the expected control, I/O, power, and ground pins needed for the wiring shown in Table 4.1, and the exact package pinout can be found in the datasheet for the desired NAND device. The final result of this setup is shown in Figure 4.2, with the Raspberry Pi being correctly wired to a TSOP-48 device.

4.3 Logic Analyzer

Figure 4.2 also shows a Saleae logic analyzer on the right. While this hardware is not necessary for the interface to function, it is an important component for debugging purposes, and provides insight into precise timing and exact

waveforms generated by both the Raspberry Pi and the connected NAND device as they communicate. The particular model shown is the `Logic Pro 16`, which supports digital sampling rates up to 500 MSPS on up to 4 channels at once, up to 100 MSPS on all 16 digital channels, and analog sampling rates up to 50 MSPS [25]. Its 16 pins are well suited to this application, as ONFI typically uses only 15 (8 data pins and 7 control pins), meaning all relevant data can be captured. In addition to its hardware capabilities, its software is intuitive and makes it easy to visually analyze digital signals, which is instrumental for development and optimization.

Altogether, the hardware setup for this new platform is straightforward and easy to replicate provided access to the correct hardware. Except for the logic analyzer, the required components are generally easy to obtain, as they can be purchased from a variety of online retailers at reasonable cost. The most difficult part of initial setup is wiring GPIO pins to an adapter correctly, and setting up the Raspberry Pi to run Linux. The only shortcoming of this specific setup is that some NAND devices are not designed for 3.3 V power and data connections and therefore will not function without additional interface circuitry, such as voltage translation hardware or other hardware modifications that would allow the Raspberry Pi GPIO interface to operate at a different voltage. This limitation was not explored during NANDWorks development; however, it would be a useful direction for future work because it could improve compatibility across a wider variety of devices. Despite this drawback, the chosen platform meets NANDWorks' main design goals. It is inexpensive, readily obtainable, and powerful enough to

support a complete Linux-based toolkit. It also allows for timing-focused, low-level GPIO control for an ONFI-compatible interface. Because of this, researchers would be able to replicate this setup without the need for specialized tools, and the wiring is clearly documented to make the setup process easier. The next chapter describes the corresponding software stack, including the GPIO/transport layer, ONFI command sequencing, and user-facing CLI and scripting interfaces, and explains how these layers are structured to support rapid experimentation.

Chapter 5. Software

This chapter describes the evolution of the software stack from ONFI Plus to NANDWorks and how the resulting toolkit is structured for reliability, extensibility, and research-oriented experimentation. The chapter first describes the original ONFI Plus codebase and its shortcomings, followed by the path taken to modify and refactor the codebase as part of the NANDWorks development process. Subsequent sections of the chapter cover the layered software stack, user interface, and timing-oriented features.

5.1 ONFI Plus

NANDWorks began as an effort to enhance and expand upon an existing software NAND flash interface called ONFI Plus, which was developed by former UAH student Prawar Poudel [15]. Since a large portion of the work to implement the fundamental ONFI interface functionality was already finished, ONFI Plus served as an important foundation for what would eventually become NAND-Works. It is written in C++ and is based on a single class called `onfi_interface` that contains the user-facing functions which let programmers work with NAND devices and initialize an ONFI connection over GPIO. The codebase is simple in terms of code organization, and the majority of the code exists as part of the

`onfi_interface` class, along with its member variable names and functions, which are declared in `onfi_head.h` and defined in `onfi_head.cpp`. User-facing functions are provided as a framework for individual operations (some complex and others more fundamental), fitting into six basic categories:

Initialization and cleanup These functions bring the interface up, map hardware, and shut the interface down when done. Calling `get_started()` runs through the necessary steps to initialize the GPIO interface, initialize the connected NAND device, and read the device's ID and parameters. `deinitialize_onfi()` or `~onfi_interface()` clean up GPIO mapping and file handles to cleanly shut down the connection.

Device discovery and geometry These functions identify the connected chip and its parameters, including block/page geometry, which allows for address translation and bounds checks. `read_id()` and `read_parameters()` pull the raw values from the connected chip, while `decode_onfi_version()` and setters like `set_page_size()` or `set_block_size()` translate selected bytes from parameter pages into usable fields.

Data operations These are the primary operations for moving data to/from the connected device and changing the state of physical cells or reading those changes back. Examples include `read_page()` or `read_block_data()` for reading data, `program_page()` or `program_pages_in_a_block()` for writing data, and `erase_block()` for erasing data.

Verification and health checks These higher-level functions simplify the process of verifying whether operations have completed successfully and checking for issues in hardware. `verify_block_erase()` and `verify_program_page()` are used to spot failed operations, while `is_bad_block()` is used to check markers to avoid known bad blocks.

Status, features, and modes These functions check the connected device's status and configure features or operating modes. `read_status()` and `check_status()` provide status information, while `get_features()` and `set_features()` read and write the device's internal features page to modify device behavior.

Diagnostics, logging, and timing These features provide standard methods of profiling and tracing output during operation. `open_onfi_debug_file()` and `open_time_profile_file()` open debug files for logging events during runtime, while `delay_function()` and `profile_time()` support controlled timing measurements.

In addition to these core categories, a few helper functions were included to automate parts of specific experiments or assist with device-specific operations:

- `convert_to_mlc()` and `revert_to_mlc()` simplify the process of converting some Micron NAND devices between SLC and MLC modes. However, this is implemented by modifying a hard-coded feature address, so it may not work on all Micron devices, or on devices that do not support the feature.

- A set of toggle-interface helpers (*e.g.*, `program_page_tlc_toshiba()`) were added for Toshiba and Samsung devices, which need to be handled separately and differ from ONFI because they use a separate toggle interface. This feature is not entirely functional and is more of an experimental scaffolding than a complete alternative interface.

- The `partial_program_page()` and `partial_erase_block()` functions begin program/erase operations and then interrupt them after a configurable delay to test how a NAND device responds.

- `verify_block_erase_sample()` provides a faster block erase verification method by checking only a single page in a block.

Instead of managing GPIO pins directly, user-facing functions in the `onfi_interface` class use abstractions for sub-operations defined in the `interface` class, which is implemented in `microprocessor_interface.cpp`. `initialize_onfi()`, for example, relies on functions such as `open_physical()`, `map_physical()`, `get_base_bridge()`, and `convert_peripheral_address()` to open `/dev/mem` and set up pointers to the register space. Additionally, the code makes use of GPIO pin management helpers such as `set_pin_direction_inactive()`, `set_default_pin_values()`, and `set_ce_low()` to regulate pin state prior to, during, and following operations. The raw bus transaction functions `send_command()`, `send_addresses()`, and `send_data()` encapsulate tight, timing-critical loops that toggle and sample GPIO pins to reliably transfer data to and from the connected NAND device. A software-emulated ONFI interface over GPIO is created by these

low-level operations using definitions from `hardware_locations.h`, which provide memory-mapped base addresses and GPIO pin offsets, as well as timing delay operations that use inline assembly to perform a predefined number of no-ops.

5.2 Development of NANDWorks

Development of NANDWorks began as an attempt to clean up the existing ONFI Plus codebase and use it as a foundation for future work on NAND devices. Although ONFI Plus compiled and ran on the DE1-SoC, the platform provided a less responsive development experience than other small Linux computers. The DE1-SoC also had limited performance and was slow even for basic tasks such as compiling C++ code. Overall, the platform's limitations made it less practical for continued development, especially given that it was only being used for its onboard CPU and GPIO pins. It was therefore decided that the best way to move forward would be to switch to a different hardware platform entirely. The initial thought was to use an NVIDIA Jetson Nano, since one was already available; however, for the many reasons discussed earlier, a Raspberry Pi 4 was borrowed and the code was adapted to the new platform.

5.3 NANDWorks Architecture

Figure 5.1 shows the layered software architecture of NANDWorks. In contrast to ONFI Plus, which concentrated most functionality in a largely monolithic implementation, NANDWorks separates the software stack into clearer layers with distinct responsibilities. The discussion in this section follows that hierarchy from

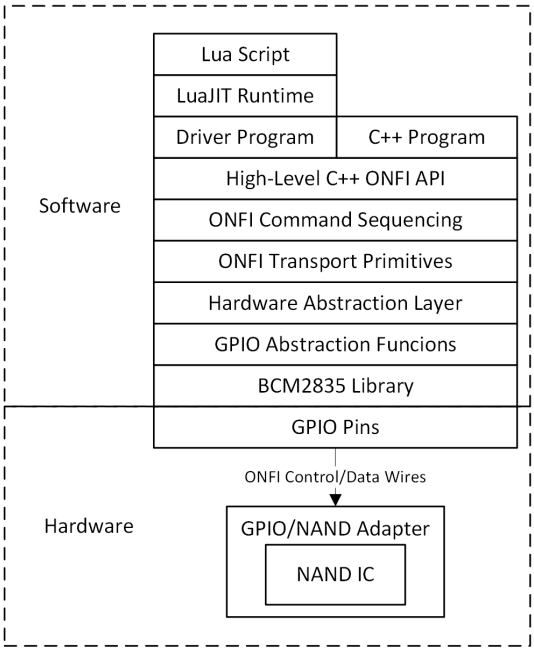


Figure 5.1: Layered software architecture of NANDWorks.

top to bottom, beginning with the user-facing interfaces and ending with the hardware-specific GPIO control layer.

5.3.1 User-facing Interfaces and Device Layer

At the highest level in Figure 5.1, `onfi_interface` remains the primary user-facing facade for direct interaction with NAND chips in C++. Another refactor provided by NANDWorks is the higher-level device wrapper, `onfi::NandDevice`. While `onfi_interface` still provides raw operations for pages and blocks, `NandDevice` provides geometry-aware operations like page reads and program/verify helpers in a form that is easy to compose into tools and experiments. While ONFI Plus largely assumed users were experienced C++ developers and

was oriented around building experiments using function calls, NANDWorks also includes additional front ends that reuse the same underlying ONFI implementation while offering simpler interfaces.

5.3.2 Command-line and Scripting Front Ends

In addition to the C++ interface shown at the top of Figure 5.1, NANDWorks includes a dedicated driver binary, also called `nandworks`, which provides centralized command-driven operation and tool-like workflows without requiring direct use of the C++ interface for every task. The driver binary centers around a command registry, which stores each command, its usage, and whether it is safe or requires explicit confirmation. The binary also provides two important checks to prevent damage to connected devices. First, commands marked as requiring hardware access will verify root privileges before execution and exit with a clear error if insufficient permissions are detected. Second, the user is required to use the `--force` flag in order to perform destructive operations such as programming, erasing, or using raw bus-driving commands. This is done to prevent accidental data modification or placing the device in an unintended state while exploring the CLI. One important note is that `raw-read-data` does not need the `--force` flag, but the other raw commands such as `raw-command`, `raw-address`, `raw-send-data`, and `raw-change-column` all require it. The binary's commands are designed with three levels of user interaction depth. The first tier provides common commands such as `probe`, `read-page`, `program-page`, and `erase-block`. The next tier is intended for diagnosis and verification by

turning common correctness checks into single-command operations. The third tier exposes raw commands such as `raw-command`, `raw-address`, `raw-send-data`, and `raw-read-data` so that the driver remains useful for low-level research. The timed commands provided by the binary for measurement are `measure-read`, `measure-program`, and `measure-erase`. These commands support the `--json` flag for automated experiments and data-capture pipelines.

NANDWorks also includes an embedded scripting system which runs LuaJIT and makes the CLI's command registry accessible as callable Lua functions [14]. LuaJIT is well suited to NANDWorks because it is optimized for speed and simplicity. It is embeddable in C++ projects and offers strong performance through just-in-time compilation, which enables time-sensitive operations written in C++ to be coordinated from a higher-level language with minimal overhead. With this approach, experiments can be developed rapidly without modifying the core code and recompiling. The scripting system was designed with an architecture similar to that of the command-line interface: commands go through the same argument parser and share the same registry. Lua calls dispatched through `nandworks.commands.*` use the same parser and command metadata as the CLI, so `--force` and per-command root requirements are enforced in the same way. The lower-level `nandworks.timed.*` helpers are bound directly and therefore do not go through CLI `--force` gating. Lua scripts also include session management functions, including `driver.start_session()`, `driver.shutdown()`, `driver.is_active()`, and `with_session`. All commands available through the command-line interface have corresponding scripting bindings, which makes it straightforward for users to move

from interactive CLI testing to full automation with only minor process changes. Additional helpers are also provided, including a `nandworks.timed` namespace for timed operations, along with `nandworks.geometry()`, `nandworks.chip_info()`, and `nandworks.now_ns()`. The standard library modules are disabled by default, meaning that the `os` and `io` libraries are unavailable unless the `--allow-unsafe` command-line option is specified when the script is executed. This improves safety by requiring explicit opt-in before Lua scripts can access host operating system and filesystem functionality through `os`/`io`.

5.3.3 Controller and Transport Layers

Beneath the user-facing layer in Figure 5.1, are the ONFI sequencing and transport abstractions. One of the major differences from the original ONFI Plus codebase is that NANDWorks is no longer organized around a monolithic implementation file for `onfi_interface`. Rather, the current codebase is organized into separate layers that can each be managed independently. In the NANDWorks implementation, `onfi_interface` acts as a bridge object that inherits from the hardware abstraction class `interface` as well as the protocol abstraction class `onfi::Transport`, as shown in Figure A.1. This refactoring step is important because it allows the sequencing layer to depend only on a narrow transport contract (send command, send address cycles, transfer data, wait-ready, delay, read data, read status), rather than directly depending on GPIO implementation details. Command sequencing is broken down using a dedicated controller type called `onfi::OnfiController`. This class is lightweight, as its primary job is to

break down larger ONFI operations into individual control sequences (*e.g.*, `0x00` → address → `0x30` → wait for a page read, or `0x60` → row → `0xD0` for block erase). Because this code is above the GPIO abstraction, it is easier to review for correctness and reuse in different front ends such as C++ programs, command-line interfaces, and Lua interfaces. The transport layer, `onfi::Transport`, serves as the interface between ONFI logic and the manipulation of physical GPIO pins.

5.3.4 Hardware Abstraction Layer

At the lowest level in Figure 5.1 is the hardware abstraction layer (HAL), implemented around the `interface` and `hal/gpio` helpers. This layer handles the low-level details of changing pin directions, managing multi-pin writes, and performing tight polling loops. Two software-wide design constraints are evident throughout the codebase. First, NANDWorks must correctly implement ONFI bus cycles, including command, address, and data phases as well as ready/busy synchronization, in software using memory-mapped GPIO. ONFI's asynchronous interface timing and signal definitions, including `CLE` or `ALE` for managing command and address phases, `WE#` or `RE#` for clocking data in and out of a connected device across the bus, and `R/B#` for detecting completion of long-latency operations, therefore become software-defined hardware in this platform. Second, because this GPIO-based approach runs under a non-real-time operating system, NANDWorks must tolerate imperfect timing determinism while reducing latency and jitter wherever possible to preserve correct protocol behavior. Despite this, the software attempts to reduce latency and jitter through a variety of techniques, some of

which are good mitigation strategies, while others such as real-time scheduling are much more important for the program to function correctly.

NANDWorks uses the bcm2835 library to standardize GPIO access [9]. This library offers a standard interface to select pin functions, perform set/clear operations, and perform low-level reads while using memory-mapped peripheral access. The HAL uses these library functions as well as low-level helpers in order to decouple the sequencing of operations from the Raspberry Pi's register-level details, which minimizes hardware-specific code and keeps it localized. NANDWorks' `gpio_init()` does more than basic GPIO initialization: it attempts to reduce sources of latency variability that matter for GPIO bit-banging. In particular, it locks current and future mappings into RAM via `mlockall(MCL_CURRENT | MCL_FUTURE)`, elevates the running thread to a real-time scheduling policy (`SCHED_FIFO`), and pins execution to a specific CPU core via `sched_setaffinity`. The CPU target can be configured at runtime through the `ONFI_PIN_CPU` environment variable, allowing experiments to isolate the thread on a chosen core. Importantly, these choices are not all treated equally by the operating system, so memory locking and CPU affinity are considered best-effort, while the real-time scheduler change is treated as a hard requirement. If elevating to `SCHED_FIFO` fails, initialization aborts and the HAL will not run, which reduces jitter where possible and makes latency-critical assumptions explicit to the user.

At the transaction-primitive level, NANDWorks uses performance-oriented design patterns that are explicitly optimized in the current codebase. The data bus (DQ0–DQ7) write path uses a precomputed lookup table that maps byte

values (0x00–0xFF) to GPIO masks. This allows each of the eight data bits to be driven with a single masked GPIO write operation. The lookup table is initialized during the load operation, eliminating branches in the fast path and minimizing overhead in the inner loops of command, address, and data transactions, which can contain thousands of byte strobes per operation in the ONFI protocol. In addition to this, Ready/Busy polling follows a fast path that reads the GPIO level register (GPLEV), which significantly reduces latency. The HAL's blocking wait continuously checks a single GPLEV with a nop in the loop to keep polling overhead minimal. The timed wait operation reads the same register using a spin counter and only performs expensive time checks every 256 iterations. This design makes R/B# polling precise while minimizing overhead in latency-sensitive loops. GPIO pin assignments are centralized in a single header (hardware_locations.hpp), covering DQ lines, control strobes, Ready/Busy, and data-strobe pins for toggle and synchronous variants. This complements the "validated default mapping" described in the project documentation and supports a maintainability goal of making changes to the physical pinout a simple configuration edit rather than a multi-file audit.

5.3.5 Research-oriented Toolkit Features

Beyond the layered core architecture shown in Figure 5.1, NANDWorks includes a number of toolkit-level features intended to support experimentation. While NANDWorks continues to offer the fundamental primitives that ONFI Plus already included, its current form extends those capabilities in ways that directly

benefit research activities. For example, the `onfi::NandDevice` wrapper offers a geometry-based interface for performing page and block operations, including inclusion or exclusion of spare OOB bytes, and even bytewise column switching when needed for compatibility or debugging purposes. Another useful extension is the `DataSink` abstraction, which enables block reads and some TLC-oriented multi-read patterns to output data to binary files or hex dumps without pre-defining output behavior as part of the NAND interface itself. NANDWorks also adds a timing-focused API surface that explicitly measures the duration of the device's busy interval using the `R/B#` signal. The `onfi::timed` helpers, including the read, program, and erase variants, provide a structured record of timing, status bits, and validity, avoiding the need to implement GPIO polling loops independently. At the implementation level, the timing is tied to `CLOCK_MONOTONIC_RAW`, a kernel-based time source that does not adjust with NTP changes. This clock is not intended to represent absolute time, but rather elapsed monotonic time relative to an arbitrary starting point. At the CLI level, these helpers are provided as `measure-read`, `measure-program`, and `measure-erase`. At the Lua level, they are provided as `nandworks.timed.read_page`, `nandworks.timed.program_page`, and `nandworks.timed.erase_block`. NANDWorks also preserves and formalizes low-level primitives that are critical to certain types of research. The availability of both `raw-*` CLI helpers and the transport mechanisms provided by `onfi_interface` means that command, address, and data cycles can be sent with fine-grained control, including incomplete or interrupted cycles that may be necessary during exploration of failure modes. The code also maintains explicit support for partial

program and partial erase, which are important for margining-style work in which the duration of the operations are intentionally shortened or interrupted. In the current implementation, the partial operation pulse width is controlled by a busy-wait delay measured in microseconds and then terminated by issuing reset. This is implemented in the API layer (`onfi_interface`), with partial erase additionally implemented in the controller layer.

Unlike ONFI Plus, NANDWorks is designed as a complete toolkit rather than a single library class. Both logging and profiling are explicit compile-time options in NANDWorks, implemented by header-only macros with compile-time level definitions per subsystem, such as ONFI versus HAL, with the aim that no code is generated on hot paths if logging is disabled. Profiling hooks such as `PROFILE_TIME` are also centralized and controlled, and a single monotonic clock source is used for consistent benchmarking and busy-waiting. The build system further reflects this toolkit-oriented design. It includes NANDWorks as a core static library as well as several binaries for the driver, applications, tests, and examples. The top-level `Makefile` auto-discovers all programs in the `apps/`, `tests/`, and `examples/` directories, which are then built against a shared `libonfi.a`, with optional regeneration of documentation through Doxygen when available. This represents an important change in how users interact with the system, as there is no longer a need to modify a single main program to create new workflows. Instead, new tooling can be added as new binaries alongside existing ones. By default, the NANDWorks repository includes several pre-defined applications that both demonstrate capabilities and provide users with ready-made programs for basic

tasks. The `profiler` app, for example, runs timed ONFI operations and reports distribution statistics such as mean, median, p95, and p99 for both busy time and wall-clock time, providing a useful summary of the performance characteristics of a given system.

Taken together, NANDWorks represents a deliberate evolution of the simpler ONFI Plus tool into a more layered, reproducible, and experiment-friendly toolkit. The ONFI semantics remain the same, but the current implementation introduces clearer architectural boundaries between the user-facing interfaces, controller, transport, and HAL layers, along with tighter controls on timing and scheduling, safer command-line usage, and a scripting interface that provides the same guarantees as the CLI when using `nandworks.commands.*` while also exposing lower-level timed helpers. NANDWorks is therefore a more complete, extensible, and experiment-oriented platform while still retaining the low-level flexibility of ONFI Plus that made it useful for research in the first place.

Chapter 6. Validation Testing

This chapter describes how NANDWorks has been validated throughout its development process to ensure the correctness of ONFI while allowing rapid refactoring and feature growth. It outlines a multi-layered approach of using automated C++ tests, profiling tools in both native and scripted versions, and manual waveform inspection using a logic analyzer. Together, these validation tests provide broad evidence of functionality as well as pulse-level evidence that protocol behavior remains correct under real operating conditions.

6.1 Validation Strategy

Continuous validation was essential throughout NANDWorks development because the project involved iterative refactoring of the original ONFI Plus codebase along with the addition of several new features. Since NANDWorks development began from a functional starting point, it was necessary to preserve that functionality despite major revisions to the code's layout and structure. This required thorough testing throughout development not only to verify new features, but more importantly to prevent regressions that could be introduced unintentionally.

This validation effort consisted of multiple components. The first component was automated testing scripts, which were developed to verify existing functionality and provide insight into issues that might have been introduced by a refactor. The second component was individual tests written to identify specific issues. The third component was manual verification of signal timing and ordering through the use of a logic analyzer, which was useful not only for confirming the functionality of specific components, but also for providing objective ground-truth information about the underlying operation of the platform. Together, these three methods provided the validation needed to develop NANDWorks with confidence while preserving correct ONFI behavior [13].

6.2 C++ Test Application (tester.cpp)

One of the first attempts at testing NANDWorks began with a simple C++ program titled `tester.cpp` which went through iterative development along with the core NANDWorks functionality. Its capabilities have grown during development, and it is now in a state where it can test and verify a significant amount of functionality in just one program. It is designed to compile to a single executable, and provides a fixed set of tests with pass/fail conditions, and summaries of each test printed to the terminal. These tests fall into five distinct categories: device bring-up, data path correctness, spare/read consistency, data integrity, and edge case robustness checks.

6.2.1 Bring-up Tests

Device bring-up tests are the first to be run, and are used to verify the most basic operations of ONFI communication are functional. The first of these is the `Device Init/Reset Test`, which verifies NANDWorks is able to successfully run the `onfi_instance.device_initialization(verbose)` and `onfi_instance.reset_device()` functions. The next test in this category is the `ONFI ID and Parameters Test` which runs and prints results from `onfi_instance.read_id()` and `onfi_instance.read_parameters(ONFI, false, verbose)`. The final test in this category is the `GET FEATURES Read Test` which runs the `onfi_instance.get_features(a, buf)` function in order to verify that NANDWorks can successfully read feature values across multiple addresses on a connected NAND device.

6.2.2 Data-Path Correctness Tests

The next set of tests are used for verifying data-path correctness, which assisted development by ensuring new changes did not break essential read/write/erase functionality. The most basic of these tests are the `Block Erase Test`, and the `Single Page Program Test`, which are used to check the functionality of the `.erase_block()`, `.verify_erase_block()`, `.program_page()` and `.verify_program_page()` functions. As a side effect of testing the verification functions, the `.read_page()` function is also tested during this process. There are also some more complex data-path verification functions, including `Multiple Page Program`

`Test` which is used to test the `.program_block()` and `.verify_program_block()` functions. Finally, there are the `Random Program/Read/Verify/Erase Test`, `Block Program Subset/Verify Test`, and `Full Block Program/Verify Test` which are used to test more end-to-end read/write/erase functionality across randomized, subset, and full block workloads. These tests are used to validate addressing and sequencing behavior at a larger scale, and are used to catch regressions that might not show up from smaller tests.

6.2.3 Spare/Read Consistency Tests

The spare/read consistency tests are an extension of the data path correctness tests, and focus more on semantic correctness in how data is handled instead of simply raw throughput. The `Spare Area I/O Test` is used to test whether or not NANDWorks is able to read/write to the OOB area, which is important, as this verifies the program has the capability to interact with block/page metadata like bad block markers, and ECC/parity fields using `program_page(..., including_spare=true)`. A similar test is the `Spare Preserved (Exclude) Test` which tests the same `program_page()` function, but with `including_spare=false` which proves that page reads still behave correctly, even when the spare area is excluded from the transfer. Another test is the `Bulk vs Bytewise Read Consistency Test` which compares the difference between the standard "bulk" read mode, and a slower but more explicit byte-by-byte reading mode, which issues explicit commands to the controller to send each byte from a page stored in the NAND device's internal read cache/register. Both reading

modes must produce exactly the same bytes for this test to pass. The final test in this category is the `Read Block With Sink Test`, which is a streaming-path smoke test used to verify that NANDWorks can read selected pages from a block and route the output correctly through the `DataSink` interface (rather than only through in-memory buffers). In practice, this checks the end-to-end `read_block()` pipeline, including subset-page iteration and formatted output streaming.

6.2.4 Data Integrity Tests

The next category of tests are data integrity tests, which are used to verify that verification logic is properly detecting and reporting errors. The first of these is the `Error Analysis Test`, which provides a more manual verification that reads and writes to pages on a NAND device are being performed correctly. It accomplishes this by programming a known pattern to a page and comparing it to what is read back while reporting the number of byte errors. Another test in this category is the `Verify Mismatch Detection Test`, which validates the built-in `verify_program_page()` function. It first writes a known-correct data buffer to a page, then creates a second expected buffer with one intentionally flipped byte; when this incorrect expected buffer is passed to `verify_program_page()`, the function should report failure. The final test in this category is the `Verify Error Counters Test`, which checks that `verify_program_page()` reports useful error information and not just a pass/fail result. It programs known data, introduces a few intentional mismatches in the expected buffer, and then confirms the reported

byte-error and bit-error counts are correct, which helps ensure NANDWorks provides trustworthy diagnostics when verification fails.

6.2.5 Edge-case Robustness Checks

The final category is edge case robustness checks, which are used to ensure NANDWorks behaves as expected, even when dealing with edge cases and device-specific behaviors that are easy to overlook during development. The `Bad Block Scan Test` and the `First/Last Block Erase Test` are both used to ensure that NANDWorks can identify usable blocks by excluding bad blocks with the `is_bad_block()` function, and can correctly erase blocks at the boundaries of the usable address space. By selecting early, boundary-like safe pages within a block to examine for addressing or sequencing regressions, the `Boundary Pages Test` is used to verify boundary behavior at the page level. Additionally, there is a `Reset Persistence Test` that makes sure data-path operations remain valid following a call to `reset_device()` and re-initialization. This is beneficial because many iterative debugging loops require frequent resets. The `Partial Erase Test` checks that the block verifies as erased after calling `partial_erase_block(...)`, a non-standard but crucial erase flow that is highly helpful in a variety of experimental setups. Lastly, NANDWorks can test complex device-specific logic without compromising portability on unsupported platforms thanks to the `TLC Subpages Test (if supported)`, which conditionally runs TLC-specific logic like `program_tlc_page()` on supported platforms.

6.3 Profiling Tools (profiler.cpp / profiler.lua)

`apps/profiler.cpp` was another useful validation component that served as a performance and capability profiling framework for NANDWorks, not merely a pass/fail unit test. In addition to offering statistical analyses (mean, median, standard deviation, and tail latency percentiles such as P95/P99) of basic read and write operations, it also runs a series of timed ONFI commands and distinguishes between ONFI busy time throughput and end-to-end wall clock throughput. As it allowed the measurement of stable, repeatable values over several iterations rather than a single point in time, it was an important tool for deterministically measuring performance while optimizing the tight ONFI communication loops for maximum performance. The tool enabled the capability to recompile NANDWorks with varying compiler flags and optimization options, execute the same profiler workload, and compare latency and throughput differences, and enabled the direct selection of build options that offered the best real-world hardware performance. It is primarily designed for non-destructive read profiling but also offers the capability to enable destructive benchmarking of program and erase paths with varying cleanup actions.

The Lua profiler script, `apps/scripts/profiler.lua` was designed to be a very similar version of the C++ version, which allowed for performance comparisons between both the scripting interface and a raw compiled C++ implementation. It supports the same core options and latency statistics, and presents the information in an identical format on the command line. This made it especially useful for

Table 6.1: Performance benchmark results for the Lua and C++ implementations.

Timing Basis	Operation	Mean (μs)	Median (μs)	P95	P99	MB/s
Lua implementation						
t_R / busy time	Read cmd	61.410	57.241	65.982	66.000	—
	Read data	61.699	65.815	65.981	66.018	66.387
	Program	933.302	462.019	1706.629	1756.241	4.389
	Erase	3764.340	3391.444	4428.259	4429.704	—
Wall clock	Read cmd	82.502	76.852	94.093	197.315	—
	Read data	1047.613	1020.389	1175.686	1314.148	3.910
	Program	2203.393	1906.352	2982.722	3125.074	1.859
	Erase	3783.044	3408.815	4446.797	4458.629	—
C++ implementation						
t_R / busy time	Read cmd	61.397	65.888	65.926	66.018	—
	Read data	61.700	65.815	65.963	66.000	66.386
	Program	933.390	458.556	1683.778	1794.388	4.388
	Erase	3431.894	3390.704	3392.741	4427.463	—
Wall clock	Read cmd	73.716	78.185	78.259	78.351	—
	Read data	1041.517	1018.407	1131.537	1231.445	3.933
	Program	2176.377	1700.519	2928.407	3036.260	1.882
	Erase	3443.999	3402.722	3405.185	4439.518	—

validating that Lua was both functioning the same as C++, but also that its performance and behavior were similar, and provided a quantifiable measurement of how much deviation there was between the two. The two profilers worked together to offer a useful A/B framework for cross-validating runtime cost and correctness across linguistic boundaries, and provided assurance that acceptable performance characteristics were being achieved during development.

Both implementations yield nearly identical distributions with similar relative ordering of the sub-operations for these benchmarking tools, as shown in Table 6.1. However, the results for `timed_read_page_cmd` and `timed_read_page_data` show that Lua adds a small but consistent amount of host-side control overhead. Despite this overhead, the majority of the time is spent on the device's NAND busy interval (the t_R wait). Given that both profile programs report a different

total time and a similar NAND device internal busy wait time, these differences can be explained by the scripting-layer overhead latency rather than any changes in the physical behavior of the NAND device.

These results also suggest that scripting overhead is less important for longer operations because device execution takes up more of the overall time. The Lua overhead is merely extended over a wider window of device execution, and program and erase operations take a lot longer to complete. This means that while the C++ version is better suited for designing experiments that would be most impacted by host overhead, such as reliance on back-to-back commands, column changes, or raw throughput performance, the Lua profiler is a helpful tool for most workflows to obtain accurate and consistent measurements.

When combined, these two figures support the goal of minimizing the differences in performance and capability between the Lua and C++ NANDWorks interfaces. While the C++ version establishes a lower bound on host overhead for a particular build and platform configuration, the Lua version demonstrates that the scripting layer is thin enough to remain true to the compiled implementation for the majority of realistic NAND experiments. These findings suggest that the trade-offs of Lua's minimal performance impact might be worth it in situations where faster iteration with an interpreted language is seen as more valuable compared to absolute raw performance.

6.4 Manual Verification Using a Logic Analyzer

The final method used to validate and test NANDWorks was to visualize logic analyzer captured data and manually review it. The clocks onboard the Raspberry Pi, while accurate enough for measuring larger operations, are not accurate enough to reliably measure the many sub-operations performed over the course of a single operation. The Saleae Logic Pro 16 used for this purpose supports digital sampling rates up to 500 MSPS, which enabled accurate observation of the exact timing of each pulse generated by the Raspberry Pi or the connected NAND device [25]. This process, while not as automated, provided some of the most valuable insight into a variety of issues including unnecessary or incorrect pin toggling behavior, timing and latency issues, and OS interrupt detection and mitigation.

6.4.1 Simple GPIO Pin Test

One of the first things tested with the logic analyzer was a pin test program, which was written to help verify that the correct pins are being used on the GPIO interface. The program uses the CLI to display which pin is about to be powered, and requires the user to press the enter key to toggle each pin one by one. While a multimeter can be used to verify the correct pinout is being used, the logic analyzer makes this process even easier by visually displaying the results, which are shown in Figure 6.1. While this test is simple, it helps rule out issues related to poor electrical connections or conductivity, as well as detect any mistakes made during



Figure 6.1: Logic analyzer capture of GPIO pin test.

wiring. Figure 6.1 demonstrates both that all GPIO pins are able to be toggled, and that both the logic analyzer and the NAND adapter are wired correctly.

6.4.2 Visualization of Basic Operations

Another component tested with the logic analyzer was basic operation functionality. This type of testing was performed by analyzing signals generated by several different programs and scripts. However, for demonstration purposes a simple test program was developed to showcase this process. This program

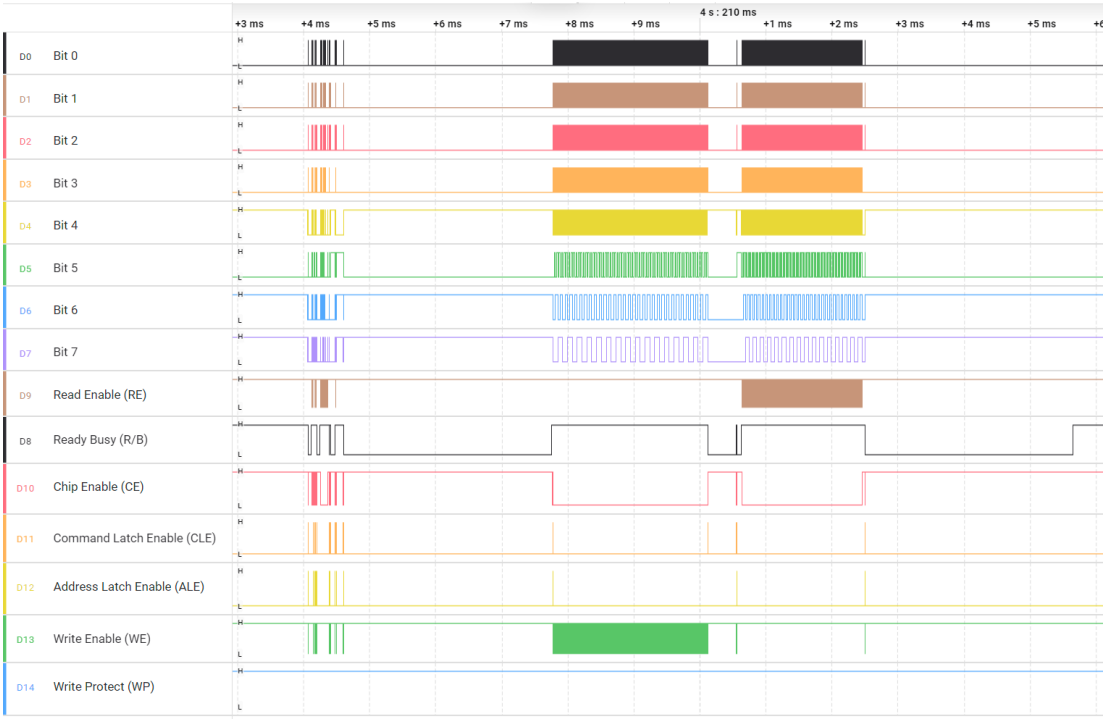


Figure 6.2: Logic analyzer capture of a demonstration erase/program/read/erase sequence.

initiates a device handshake, erases a block, programs a page with values from 0 to 255, reads the page back, and then erases the block again. Figure 6.2 shows the results of this capture, with a visible handshake section, a waiting period for the initial erase, a program step with visible write enable activity, a short waiting period to program the page, a read step with visible read enable activity, and a final waiting period for the final erase.

Taking a closer look at the initialization sequence in Figure 6.3, we can see the initial handshake where NANDWorks gathers the connected device's identification data, interface type, full parameter page, and geometry before initiating an erase command. The majority of the erase command is simply

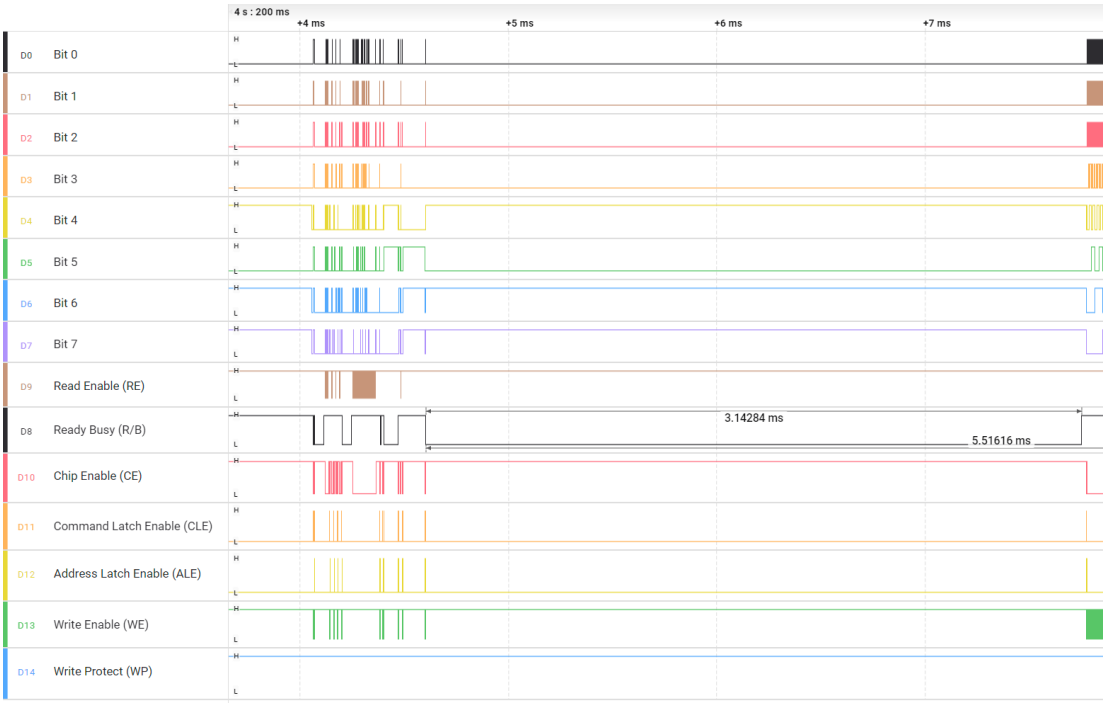


Figure 6.3: Logic analyzer capture of initial NANDWorks handshake followed by a block erase operation and the initiation of a page write operation.

waiting for the device to assert *ready*, which takes approximately 3.1ms in this case. Immediately following completion of the erase command, and the assertion of *ready*, NANDWorks begins the page program sequence.

Figure 6.4 shows a closer view of the *ready* signal being asserted at the end of the first erase operation, along with a short internal waiting period, and the beginning of the write operation. Also visible in the figure are the CLE and ALE lines being toggled with WE and bus wires to send the command and address to the connected device. The remainder of the capture shows the sending of a repeating pattern of data across the bus, toggled by the WE line. Captures like this were useful during development for analyzing and eventually optimizing

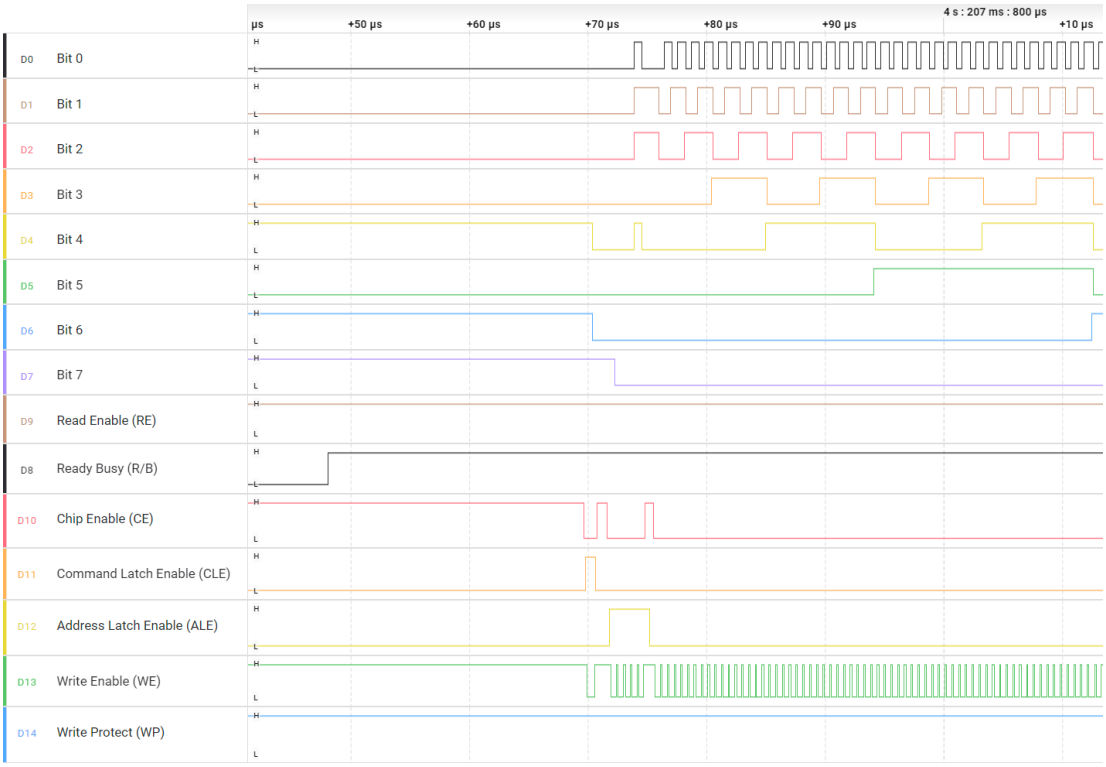


Figure 6.4: Logic analyzer capture of latency between the end of an erase operation and the start of a program operation.

to reduce latency between operations, as precise measurements can be made to quantify the latency between *ready* being asserted, and the beginning of the next operation, or even sub-operations in the case of sending commands and addresses.

Figure 6.5 shows the gap between the end of the program operation, and the beginning of reading the same page back. The major component of this gap consists of an approximately 227 μs waiting period for the connected NAND device to write the data sent over the bus to the physical flash cells, before asserting *ready*. This is then followed by NANDWorks beginning the read operation, by sending the command and address to the NAND device, followed by another

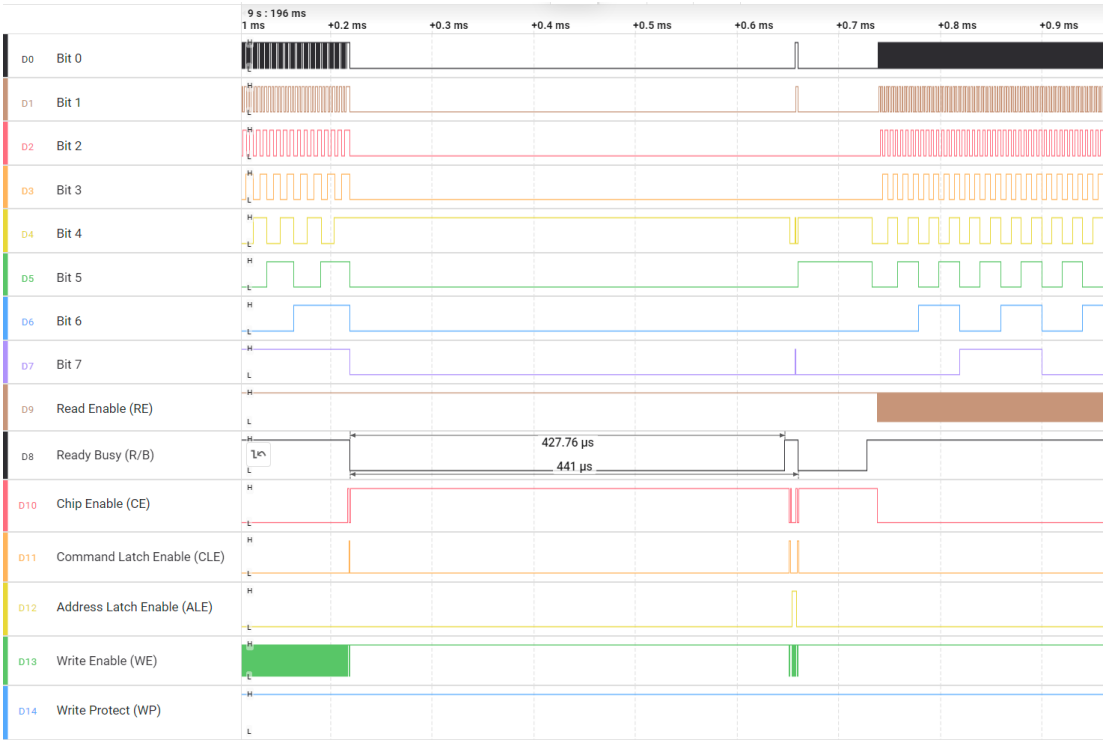


Figure 6.5: Logic analyzer capture of the tail end of a write operation followed by a read operation on the same page.

waiting period where the device fetches the data from the flash cells and stores it inside its internal buffer. The final part of the capture shows the same repeating data pattern being read from the device showing both that the correct block and page were chosen, but also that the data was properly stored and retrieved.

6.4.3 OS Interrupt Profiling and Mitigation

Logic analyzers provide a visual representation of the major components of latency during program runtime along with quantifiable results, which is helpful not only for verifying the impact of optimizations but also for finding previously

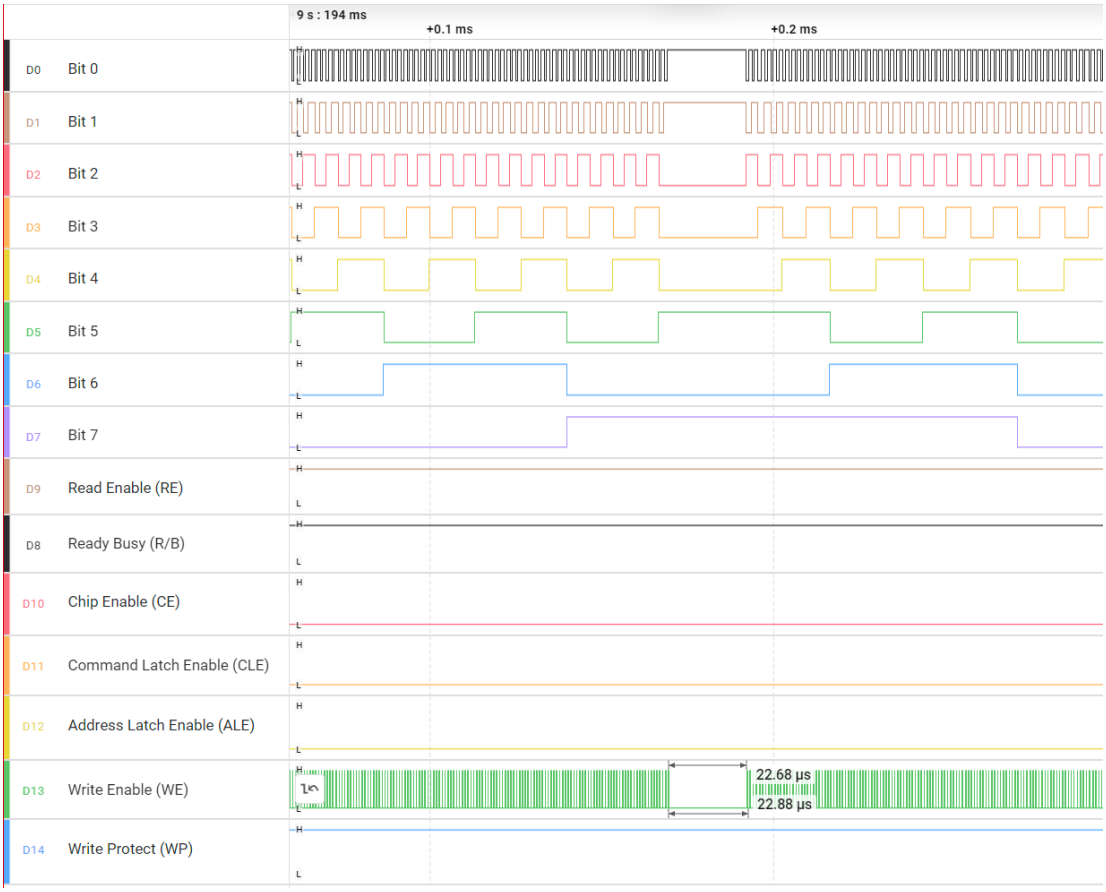


Figure 6.6: Logic analyzer capture of an interrupt in the middle of a program operation data transfer phase.

unknown issues. One of these issues was caused by the tight loops inside NAND-Works being interrupted by the operating system, hardware, or by other means. Figure 6.6 shows a typical occurrence of this type of interrupt, along with a typical delay ($\sim 23\,\mu$s in this case). These kinds of visualizations assisted in mitigating these interrupts, and led to improvements such as locking the NANDWorks process to a single core, increasing its priority, and pre-allocating memory. These changes

greatly decreased the frequency and magnitude of these interrupts, however as evidenced by the figure, the issue has not been entirely eliminated.

Using a logic analyzer allowed for better debugging and optimization through precise time measurement and in-depth pulse-level analysis, giving deeper insight into the underlying signaling between NANDWorks and its connected NAND device. The captures help to quantify the effects of the current state of these mitigations, demonstrating that the remaining jitter is infrequent and limited in magnitude for the targeted experiments. Although OS-induced interrupts and jitter still exist, they have decreased due to closer analysis that allowed the addition of certain mitigation techniques. Logic-analyzer validation was able to serve as a ground truth at the pulse level and was useful for improving throughput profiling and higher-level functional tests.

Chapter 7. Experiments

This chapter builds upon the functional correctness and validation established in Chapter 6 by presenting a series of representative experiments designed to demonstrate NANDWorks' capabilities as a platform for experimentation. Instead of describing a single NAND device, the goal is to demonstrate how controlled command sequences can be repeatedly expressed and measured for timing, error behavior, and energy measurement. This chapter covers the development process and results of experiments including high/low-bit page analysis, partial program/erase characterization, and energy measurement during partial-erase workloads using a Joulescope precision current/energy measurement instrument.

7.1 Experiment 1: High/Low Bit Page Analysis

An experiment was conducted to characterize the page-type layout in a Micron `29F256G08CBCBB` 3D MLC NAND device in order to illustrate NANDWorks in a research-oriented context. Each physical cell is programmed only once and is not paired with another page in many 3D NAND designs, which reserve a set of pages close to block boundaries as *unshared* pages (often operated effectively as SLC) [27]. The least-significant-bit (LSB) page is written before the most-significant-bit (MSB) page in *shared* MLC pages, which are programmed sequentially and store

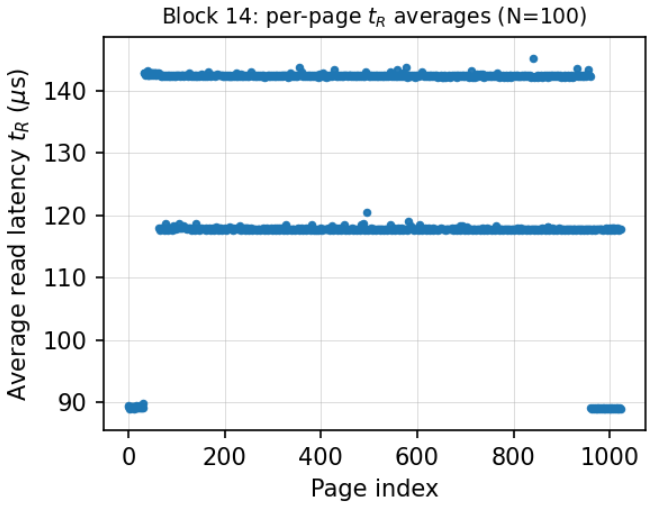


Figure 7.1: Read latency for each page in a block of a Micron 3D MLC NAND device.

two bits per cell [8]. This experiment aims to infer a map from logical page number to page type (unshared/LSB/MSB), and use a programming-order stress test to validate the inferred classification.

7.1.1 High/low Page Latency Analysis

The experiment's first phase measures the page-read latency (t_R) for every page in a single block. The program `mlc_tr_cluster_sample.cpp` uses `onfi::timed::read_page()` to measure the time between issuing the read command and observing the device ready signal while sweeping the page address. To minimize noise from host-side scheduling and I/O jitter, the latency for each page is averaged over 100 read operations. Figure 7.1 shows the resulting per-page latency map.

Because sensing does not require resolving multiple threshold-voltage states, unshared pages exhibit the smallest t_R, which is consistent with previous observations in 3D MLC NAND [8, 27]. The majority of the remaining pages in this device alternate between two higher latency bands (roughly 118 and 142 μs), which correspond to LSB and MSB pages, respectively. The first region of unshared pages reads at about 89 μs, and the block's last section deviates from a straightforward contiguous pattern where shared and unshared page latencies seem to be interleaved, suggesting that the logical page numbering that the NAND interface exposes does not accurately reflect the underlying physical cell pairing.

7.1.2 Reverse Program Order Analysis

The second part of this experiment purposefully breaks the necessary MLC programming sequence in order to support the latency-based classification. MLC pages must be programmed in ascending page order within a block, with the LSB being programmed before the corresponding MSB. Programming in the wrong order can result in significant corruption and leave cells in an undefined intermediate state [13, 8]. This test is implemented in `program_order_bit_errors.cpp` shown in Figure A.2, where a block is erased, all pages are programmed with pseudo-random data in ascending order, the data is read back, and the number of per-page bit errors is reported. The process is then repeated while programming pages in descending order. The two scenarios are contrasted in Figure 7.2. Reverse-order programming induces large error counts on pages identified as shared (LSB/MSB), while pages classified as unshared remain largely unaffected since their physical

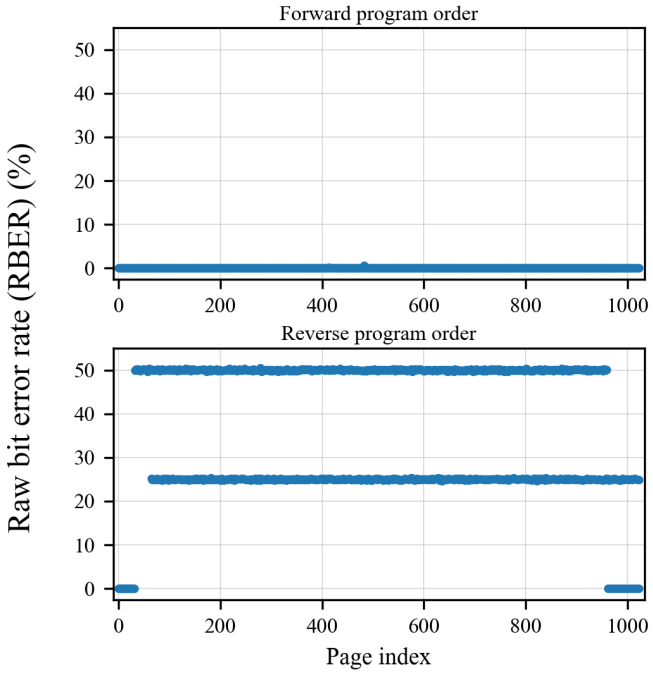


Figure 7.2: Forward/reverse programming RBER in a Micron 3D MLC NAND device.

cells are programmed only once and are not disturbed by a subsequent paired-page program [15, 13, 8].

7.2 Experiment 2: Partial Program/Erase Operations

The partial program and erase functionality in NANDWorks are designed to test the boundaries of NAND devices by interrupting operations at precise intervals. Because operations in NAND are not instantaneous, it is possible to use these partial operations to gather information on exactly what the NAND device is doing during its internal operation time [10]. Rather than occurring as a single instantaneous transition, NAND program and erase operations are performed internally using repeated high-voltage pulse-and-verify cycles. As

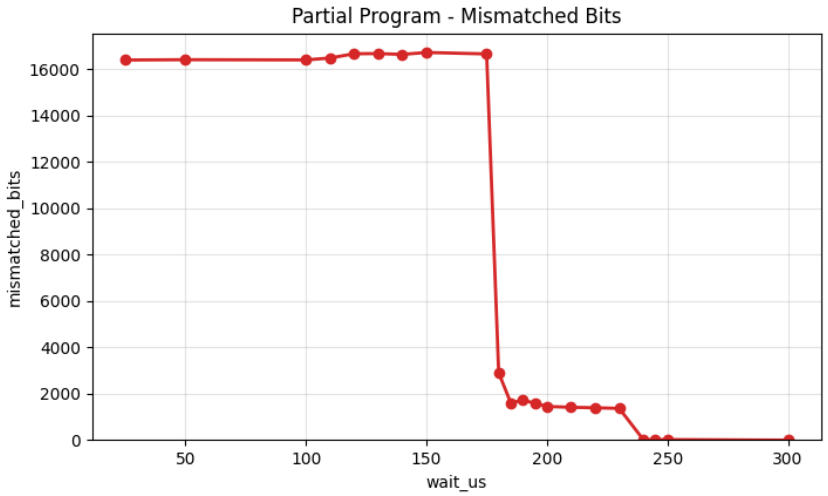


Figure 7.3: Results of partial program experiment showing the relationship between number of correctly programmed bits and the duration of the operation before interrupt.

a result, interrupting an operation at different times effectively truncates this internal sequence at different stages. This explains why Figures 7.3 and 7.4 show threshold-like regions and plateaus: useful state changes mostly take place during the pulse phases, with little to no additional change occurring during intermediate verification phases. Cell-to-cell variation, where slower cells need more pulses to reach the target threshold-voltage distribution, is also consistent with the remaining long tail near completion.

The program `partial_margin_sweep.cpp` shown in Figure A.3 was written to experiment with these margins and performs several of these partial operations using different intervals. This CLI program takes user input for the block, page, and number and duration of wait intervals to test, and runs a series of partial program and partial erase operations. For partial program operations, the expected contents are compared to the actual contents of the page after the operation in

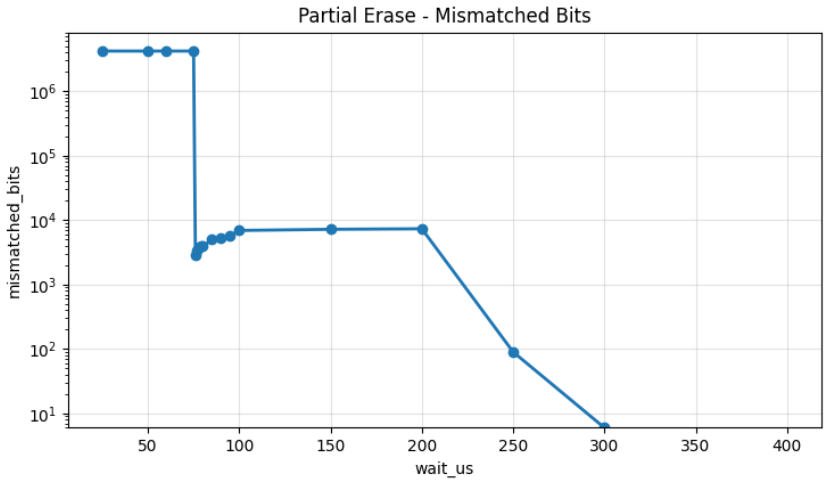


Figure 7.4: Results of partial erase experiment showing the relationship between the number of correctly erased bits and the duration of the operation before interrupt.

order to determine how many bits were successfully programmed. Likewise, for partial erase operations, the block's entire contents are compared before and after the operation in order to determine how many bits were successfully erased.

Running this partial program/erase experiment on a Micron `29F16G08CBACA` 2D MLC device yielded interesting results. The partial program operation shown in Figure 7.3 was performed with interrupt delays from 25 μs to 300 μs and shows a substantial drop in bit errors starting at 175 μs, indicating that the NAND device's internal page programming step does not start immediately. From 175 μs to $\sim$ 280 μs, the number of bit errors plateaus, followed by another drop in bit errors approaching zero, which demonstrates that the NAND device performs multiple steps at specific intervals to iteratively program a page [10]. Figure 7.4 shows the partial erase test with interrupt delays from 25 μs to 300 μs and again indicates that the NAND device does not perform the operation immediately, since

the first drop in bit errors does not appear until 75 μs. This initial drop is followed by a slight rise in bit errors from 75 μs to 200 μs and then by a steady decline to zero from 200 μs to 300 μs. These results show that the NANDWorks partial operations function correctly and can reliably produce timing intervals precise enough to introduce measurable differences in chip behavior in a deterministic way. Having the ability to run these types of experiments allows researchers to test devices in non-standard operating modes and develop a deeper understanding of their internal operation.

7.2.1 Visualizing Partial Operations

In order to further demonstrate the functionality of partial operations, the UAH logo was converted into a 1-bit image so that it could be written to a single page as a binary file. This data was then read back after various durations of partial program and partial erase operations and converted back into grayscale images. A simple C++ program, `partial_binary_io.cpp` shown in Figure A.4 was written to allow users to select binary files, choose which partial operation to perform on the binary data, and select the duration of the partial operation. The grayscale image was converted to binary and used as the data for these experiments. In the case of partial program operations shown in Figure 7.5, the visualizations follow the findings in Figure 7.3, demonstrating little to no correlation with the reference at 150 μs, some correlation at 200 μs, better correlation at 250 μs, and near-perfect correlation at 300 μs.

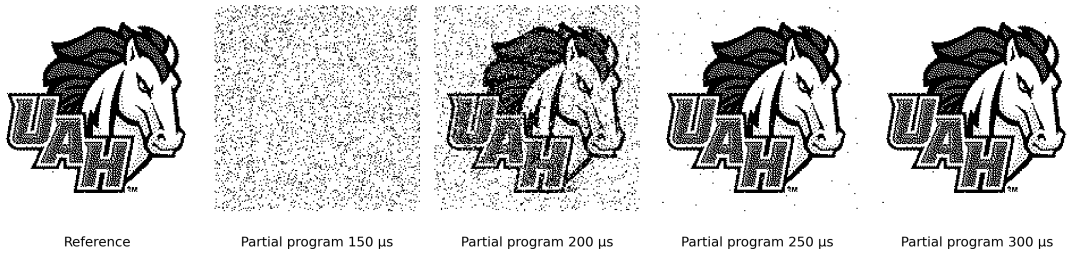


Figure 7.5: Results of experiment visualizing bit errors in an image at various partial program durations.

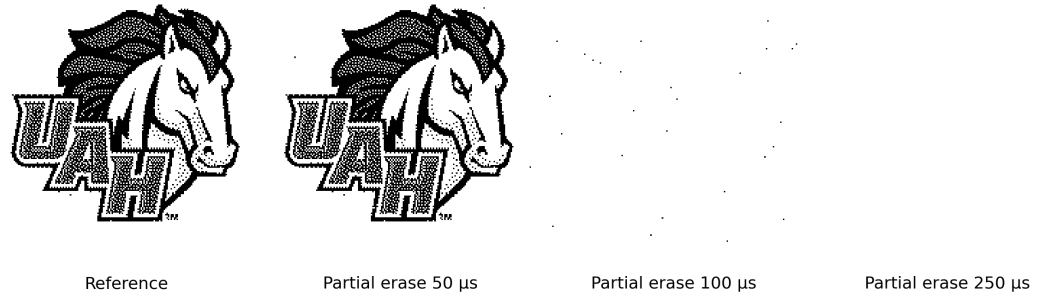


Figure 7.6: Results of experiment visualizing erased information as an image at various partial erase durations.

Similarly, the partial erase visualizations in Figure 7.6 follow the findings shown in Figure 7.4, and the 50 μs operation shows little change, 100 μs shows a near-complete erasure, and 250 μs shows a perfect erasure of all data from the tested page. These visualizations offer a clear connection between abort timing and the bit-level state of the resulting page. Partial operations can be used as a controlled primitive for creating intermediate memory states, as demonstrated by these operations creating incomplete/unfinished states that are neither fully programmed nor erased. When combined with the quantitative analysis, this method becomes a useful workflow for rapidly verifying partial-operation behavior on actual devices and validates duration as a useful experimental parameter.

7.3 Experiment 3: Energy Saving With Partial Erase

A side effect of performing partial operations on NAND devices is that they are not only performed faster, but can also consume less total energy. In these measurements, shorter interrupted operations required less total charge and therefore less total energy. To demonstrate this behavior, an experiment was devised that uses a Joulescope JS220 in tandem with NANDWorks to perform partial erase operations and capture information about energy consumption. For this experiment, the Joulescope's current sensing section was connected in series with the power lines of the connected NAND device in order to measure current, and its voltage sensing section was connected in parallel in order to measure voltage. Once the Joulescope was connected, the same `29F16G08CBACA` NAND device was tested using NANDWorks driver program CLI commands, which simplified the process of conducting the experiment by eliminating the need to write a dedicated program or script to perform single operations. These measurements were intended to reflect end-to-end energy use in a realistic operating scenario rather than only the internal device erase phase. Accordingly, the captured energy includes the full CLI-driven operation, including command startup, handshake, command transmission, and the device-side erase interval, rather than restricting the measurement window only to the time during which the NAND device holds R/B low. Because the supply voltage remains nearly constant during these captures, energy ($E = \int V(t)I(t)\,dt$) is approximately proportional to charge ($Q = \int I(t)\,dt$), so integrating voltage explicitly is unnecessary for this demonstration. Instead, using the average voltage

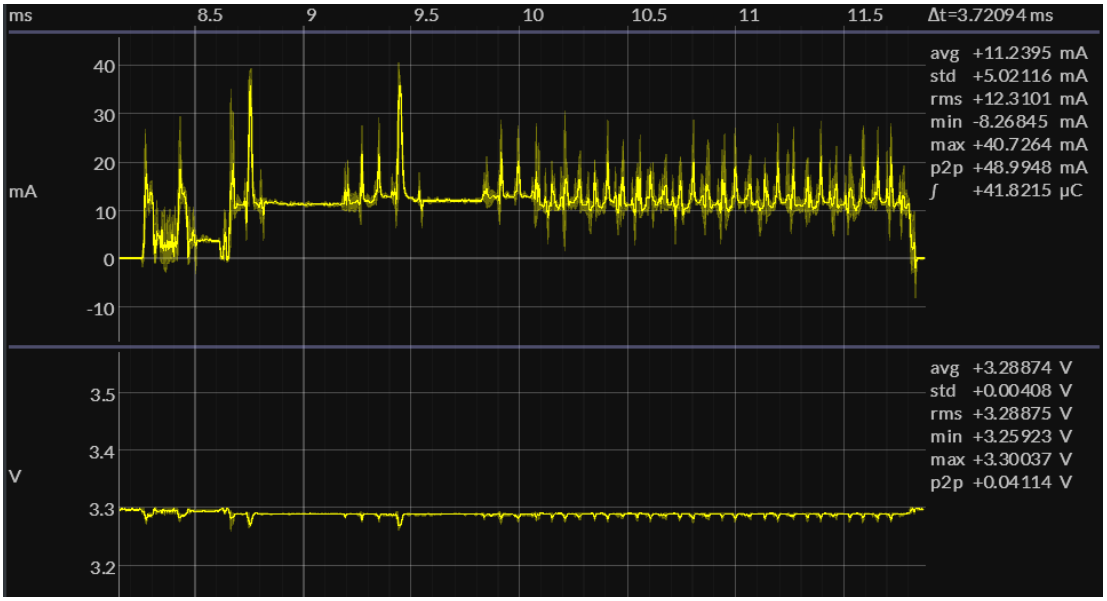


Figure 7.7: Joulescope capture of full erase operation performed using NANDWorks CLI.

during the operation, the total energy can be approximated as

$$E \approx \bar{V}\, Q = \bar{V} \int I(t)\, dt.$$

The first test shown in Figure 7.7 demonstrates a complete erase operation from start to finish including the startup and handshake procedure from running the CLI command. For the total duration of the operation, the charge consumed was $41.8215\,\mu$C at an average of 3.28874 V, meaning energy consumption is approximately $E \approx (3.28874\,\text{V})(41.8215\,\mu\text{C}) \approx 137.54\,\mu$J.

The second test was performed with a partial erase operation, which was cut off at $1000\,\mu$s, which is far greater than the $300\,\mu$s delay that was demonstrated earlier to have been sufficient to fully erase all contents in a block. Despite this

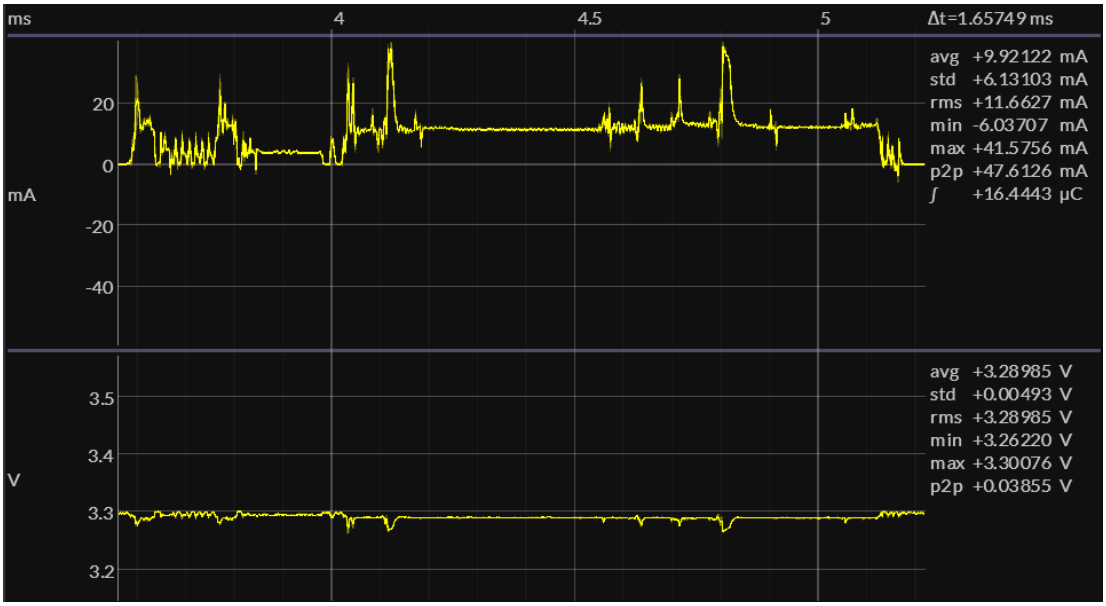


Figure 7.8: Joulescope capture of 1000 microsecond partial erase operation performed using NANDWorks CLI.

extra margin, Figure 7.8 shows a total charge consumption of 16.4443 μC at an average of 3.28985 V, meaning energy consumption is approximately $E \approx (3.28985\,\text{V})(16.4443\,\mu\text{C}) \approx 54.099\,\mu\text{J}$, which is less than 40% of the original energy consumption.

The third test was performed with another partial erase operation, which was cut off at 300 μs in order to test the energy consumption near the experimentally verified minimum duration. Figure 7.9 shows a total charge consumption of 6.67599 μC at an average of 3.28972 V, meaning energy consumption is approximately $E \approx (3.28972\,\text{V})(6.67599\,\mu\text{C}) \approx 21.962\,\mu\text{J}$, which is less than 16% of the original energy consumption.

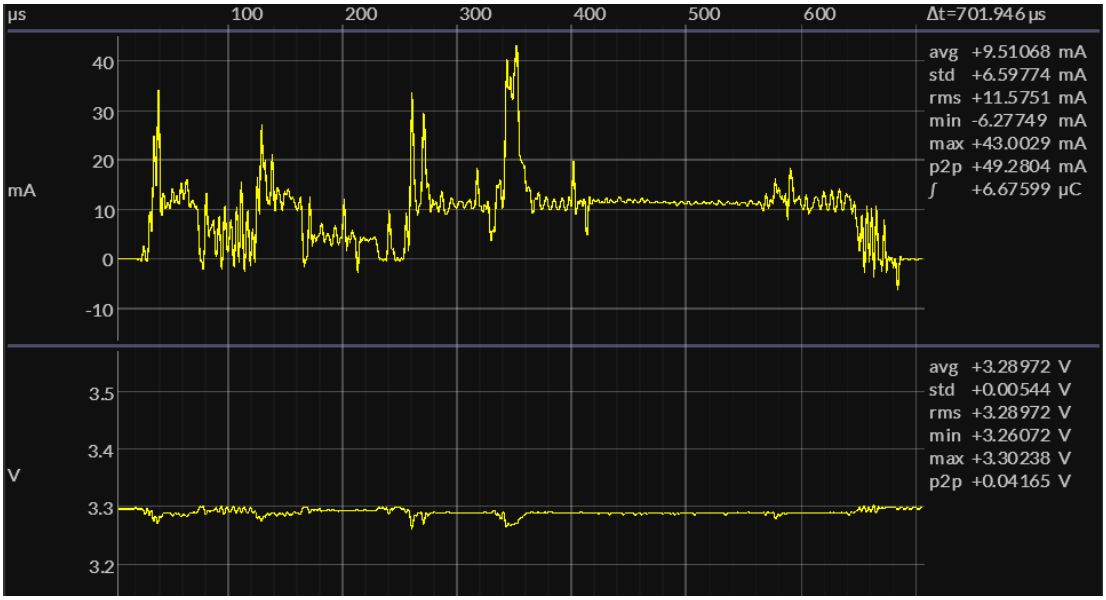


Figure 7.9: Joulescope capture of 300 microsecond partial erase operation performed using NANDWorks CLI.

These results demonstrate that performing partial erase operations can be used to reduce energy consumption; however, more research would be needed on specific devices in order to understand the long-term effects of continuously performing partial erase. In addition to showing interesting properties of NAND flash, this experiment demonstrates the ability of the NANDWorks CLI driver program to run meaningful experiments. While a dedicated C++ program or Lua script could have been written to perform these tasks, the simplicity of driving operations from the command line made it an ideal choice, and the results demonstrate that the CLI was more than sufficient for this task.

Chapter 8. Concluding Remarks

This thesis was motivated by the need for an affordable and accessible platform that could offer direct low-level access to ONFI-compatible NAND flash devices. Although commercial characterization platforms offer impressive performance and capability, they are typically unaffordable and inaccessible for the average student and many researchers. At the other end of the spectrum, more affordable USB-based platforms are more accessible but often offer too much overhead and timing variability for experiments that rely heavily upon precise command timing and latency repeatability. This thesis addressed that gap through the development of the NANDWorks platform, which uses a Raspberry Pi 4 to provide direct access to ONFI-compatible NAND flash devices through software-controlled GPIO bit-banging in Linux user space.

NANDWorks was originally designed as an extension of prior work done using the DE1-SoC platform, but the final product is more than just an adaptation of the original software to the new hardware platform. Rather, the software was developed into something more complete and useful for experimentation, with the added benefit of a layered software stack that distinguishes between hardware access, ONFI sequencing, and user interaction. In addition to the C++ core, the NANDWorks software package includes a command-line driver and a LuaJIT

scripting interface, which make it easier to create experiments without the need to modify the original core software package.

8.1 Summary of Contributions

The first significant contribution of this thesis is the creation and implementation of NANDWorks itself as a low-cost ONFI NAND experimentation platform. The system shows that a single-board computer can drive a raw NAND interface directly from Linux user space while still providing sufficient timing control for a wide range of useful experiments. The choice of the Raspberry Pi 4 as the underlying platform, combined with the memory-mapped GPIO model and the software stack optimized for ONFI communication, helps strike an appropriate balance between cost and the experimental capability.

The second major contribution is the software architecture employed to support this platform. NANDWorks was structured with a hardware abstraction layer, ONFI command and transport logic, a C++ API, a command-line driver, and an embedded scripting system. This layering of the code is important because it allows the system to be both flexible at the low level and simple to extend and use. The addition of the command-line driver and the Lua scripting support is also significant because it transforms the code from a simple codebase that needs to be adapted for each experiment into a more general framework for iterative experimentation. This makes the platform more useful for research and education.

The third key contribution is the validation approach that was employed throughout the development process. Rather than simply using one type of

test, NANDWorks was validated using multiple types of tests, including C++ tests, profiler tools, and pulse-level inspection using a logic analyzer. The test application was designed to verify the basic bring-up, data path correctness, spare area correctness, data integrity, and several edge cases. The profiler tools were used to provide reproducible performance tests for the compiled C++ path as well as the Lua scripting path. This allowed for direct comparison of the performance of both paths. The logic analyzer captures were used as a "ground truth" verification of the signal timing, command ordering, and the presence of operating system interrupts and other sources of jitter. Together, these tests provided overall verification that NANDWorks continued to maintain correct ONFI behavior.

The final major contribution is the utilization of NANDWorks to execute representative experiments on actual NAND devices. In this regard, the experiments presented in Chapter 7 demonstrate that the platform is not restricted to simple communication and read, program, and erase operations, but is also capable of enabling more research-oriented experiments. For instance, the high/low bit page experiment showed that NANDWorks could be employed to infer page-type structure and investigate the effects of program order using a Micron 3D MLC device based on measured read latency and bit error rate. Next, the partial program and partial erase experiments showed that NANDWorks could be used to execute controlled interrupted operations with timing intervals precise enough to produce deterministic and measurable effects in the connected device's behavior. Finally, the Joulescope-based experiments showed that interrupted erase operations using this device configuration consumed less total charge and energy compared to a full

erase operation, thereby showing that NANDWorks can support a workflow that combines low-level NAND operations with other device measurement techniques.

In conclusion, the overall outcome of this thesis indicates that the main goal of NANDWorks was achieved. This is because the platform provides direct access to ONFI-compatible NAND devices using low-cost and readily available hardware, while still being capable of validation, profiling, and representative device-level experimentation. In this way, it helps fill the gap between inaccessible commercial tools and less expensive tools that do not provide the same level of timing control and extensibility.

8.2 Limitations

Although NANDWorks is a useful and capable platform, it is important to understand what has been accomplished here and what has not. NANDWorks is still, at its core, software-controlled GPIO bit-banging running under Linux user space. This means that, despite great effort to design it properly in order to minimize variability and maximize repeatability, it does not provide the same level of performance as, for instance, an FPGA-based controller, nor is it going to have quite the same level of performance as a dedicated memory test system. NANDWorks is instead a middle ground; it is easier to use and less expensive, but it is still limited by being an operating system-based approach to controlling timing-critical communication.

A second limitation is the scope of experimental coverage. The validation and experiments in this thesis do demonstrate the capability and usefulness of the

technology, but they do not represent an exhaustive study of many vendors for NAND flash memory. What is shown in this thesis is a set of representative devices and experiment types that are sufficient to demonstrate that the technology works and is useful, but this does not mean that all NAND flash devices behave exactly the same, nor does it mean that all features of the tested devices have been fully explored.

The final limitation is that this work is primarily focused on asynchronous ONFI communication and experiments that the existing hardware and software stack support. NANDWorks is not intended to support more advanced interfaces such as NV-DDR, high-level controller emulation, or production-scale memory tests. Furthermore, although this framework has many useful extensions beyond the original ONFI Plus codebase, there is still advanced device behavior and many features that are beyond the scope of what was completed in this thesis.

8.3 Future Work

There are a number of directions for future work, all of which follow naturally from the work presented in this thesis. One of these options is the continued expansion of NANDWorks' device-specific and advanced feature support. Further work in areas such as internal scrambling, read retry, and copy-back commands would add to the overall variety of experiments that can be performed using the tool, making it even more useful for characterizing modern NAND flash devices beyond the basic ONFI commands. This would also move the tool into the realm of a more complete general-purpose NAND flash characterization tool.

Another important direction is the extension of the experimental coverage to a wider set of devices. Extending the application of the NANDWorks platform to a wider set of 2D and 3D NAND devices from different manufacturers, with different page organization, density, bits-per-cell configurations, and physical packaging would allow study of the architectural and reliability-related trends in these devices.

Further improvements could also be made to the timing resolution, observability, and the automation of the platform. Although the current platform already provides profiling tools, logic analyzer validation, and support for Lua scripting, further improvements could be made to include more sophisticated automation capabilities that would make the NANDWorks platform even more useful for structured experimentation. Further optimization of the timing-critical paths of the platform and the exploration of host-side jitter mitigation could provide more confidence in experiment reproducibility, especially for experiments that are highly sensitive to timing.

In conclusion, the work presented in this thesis shows that a low-cost, software-based solution is still viable for providing direct and controllable access to ONFI-compatible NAND devices. While NANDWorks does not attempt to compete with the highest-end commercial or FPGA-based solutions, it provides a foundation on which future NAND experimentation, education, and characterization work may be built.

References

[1] ADVANTEST CORPORATION. Memory test systems. Accessed: Jan. 4, 2026. URL: https://www.advantest.com/en/products/semiconductor-test-system/memory/.

[2] Matchima Buddhanoy, Aleksandar Milenkovic, Sudeep Pasricha, and Biswajit Ray. Page-overwrite data sanitization in 3D NAND flash: Challenges, feasibility, and the PULSE solution. *ACM Transactions on Embedded Computing Systems*, 24(5s), September 2025. doi:10.1145/3761798.

[3] T. Bunker, M. Wei, and S. Swanson. Ming II: A flexible platform for NAND flash-based research. Technical Report CS2012-0978, Department of Computer Science and Engineering, University of California, San Diego, May 2012. Accessed: Jan. 4, 2026. URL: https://cseweb.ucsd.edu/~m3wei/assets/pdf/MingIITechReport.pdf.

[4] Y. Cai, Y. Luo, S. Ghose, E. F. Haratsch, K. Mai, and O. Mutlu. Read disturb errors in MLC NAND flash memory: Characterization, mitigation, and recovery. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 438–449, June 2015. doi:10.1109/DSN.2015.49.

[5] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Cristal, O. S. Unsal, and K. Mai. Error analysis and retention-aware error management for NAND flash memory. *Intel Technology Journal*, 17(1):140–162, 2013. Accessed: Jan. 4, 2026. URL: https://www.intel.com/content/dam/www/public/us/en/documents/research/2013-vol17-iss-1-intel-technology-journal.pdf.

[6] A. Goda. Recent progress on 3D NAND flash technologies. *Electronics*, 10(24), December 2021. doi:10.3390/electronics10243156.

[7] J. Kwak, S. Lee, K. Park, J. Jeong, and Y. H. Song. Cosmos+ OpenSSD: Rapid prototype for flash storage systems. *ACM Transactions on Storage*, 16(3), July 2020. doi:10.1145/3385073.

[8] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu. Enabling accurate and practical online flash channel modeling for modern MLC NAND flash memory. *IEEE Journal on Selected Areas in Communications*, 34(9):2294–2311, 2016. doi:10.1109/JSAC.2016.2603608.

[9] M. McCauley. bcm2835: C library for broadcom BCM 2835 as used in raspberry pi. Accessed: Jan. 4, 2026. URL: https://www.airspayce.com/mikem/bcm2835/.

[10] Micron Technology, Inc. TN-29-19: NAND flash 101: An introduction to NAND flash and how to design it in to your next product. Technical Note TN-29-19, Micron Technology, Inc., April 2010. Accessed: Jan. 4, 2026. URL: https://user.eng.umd.edu/~blj/CS-590.26/micron-tn2919.pdf.

[11] NplusT Srl. Nanocycler. Accessed: Jan. 4, 2026. URL: https://www.n-plus-t.com/products/nanocycler/.

[12] Jeong Wook Oh. Reverse engineering flash memory for fun and benefit, August 2014. Accessed: Jan. 4, 2026. URL: https://www.blackhat.com/docs/us-14/materials/us-14-Oh-Reverse-Engineering-Flash-Memory-For-Fun-And-Benefit-WP.pdf.

[13] ONFI Workgroup. Open NAND flash interface specification, February 2024. Specification version 5.2. URL: https://onfi.org/files/ONFI_5_2_Rev1.0.pdf.

[14] M. Pall. Luajit. Accessed: Dec. 24, 2025. URL: https://luajit.org/luajit.html.

[15] Prawar Poudel. *Exploiting Physical Properties of Flash Memories for Enhancing Security and Energy Efficiency of Embedded Systems*. Ph.d. dissertation, The University of Alabama in Huntsville, Huntsville, Alabama, 2021. URL: https://alexmilenkovich.github.io/publications/files/prawar.poudel.dissertation.pdf.

[16] Md Raquibuzzaman, Matchima Buddhanoy, Aleksandar Milenkovic, and Biswajit Ray. Instant data sanitization on multi-level-cell NAND flash memory. In *SYSTOR '22: Proceedings of the 15th ACM International Conference on*

Systems and Storage, pages 85–95. ACM, June 2022. doi:10.1145/3534056.3534941.

[17] Md Raquibuzzaman, Aleksandar Milenkovic, and Biswajit Ray. EXPRESS: Exploiting energy–accuracy tradeoffs in 3D NAND flash memory for energy-efficient storage. *Electronics*, 11(3), January 2022. doi:10.3390/electronics11030424.

[18] Md Raquibuzzaman, Aleksandar Milenkovic, and Biswajit Ray. Hide-and-seek: Hiding secrets in threshold voltage distributions of NAND flash memory cells. In *HotStorage '23: Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, pages 80–86. ACM, July 2023. doi:10.1145/3599691.3603415.

[19] Md Raquibuzzaman, Aleksandar Milenkovic, and Biswajit Ray. Intrablock wear leveling to counter layer-to-layer endurance variation of 3-D NAND flash memory. *IEEE Transactions on Electron Devices*, 70(1):70–75, January 2023. doi:10.1109/TED.2022.3224420.

[20] Raspberry Pi Ltd. BCM2711 ARM Peripherals, January 2022. Build date: Jan. 18, 2022; Accessed: Mar. 6, 2026. URL: https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf.

[21] Raspberry Pi Ltd. Raspberry Pi 4 Model B. Product Brief, December 2025. Published: Dec. 2025; Accessed: Mar. 6, 2026. URL: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf.

[22] Sadman Sakib, Aleksandar Milenkovic, Md Tauhidur Rahman, and Biswajit Ray. An aging-resistant NAND flash memory physical unclonable function. *IEEE Transactions on Electron Devices*, 67(3):937–943, March 2020. doi:10.1109/TED.2020.2968272.

[23] Sadman Sakib, Aleksandar Milenkovic, and Biswajit Ray. Flash watermark: An anticounterfeiting technique for NAND flash memories. *IEEE Transactions on Electron Devices*, 67(10):4172–4177, October 2020. doi:10.1109/TED.2020.3015451.

[24] Sadman Sakib, Aleksandar Milenkovic, and Biswajit Ray. Flash-dna: Identifying nand flash memory origins using intrinsic array properties. *IEEE*

Transactions on Electron Devices, 68(8):3794–3800, August 2021. doi:10.1109/TED.2021.3087454.

[25] Saleae. Logic pro 16 data sheet, 2026. Accessed: Mar. 5, 2026. URL: https://downloads.saleae.com/specs/logic_pro_16_data_sheet.pdf.

[26] SparkFun Electronics. Introduction to the raspberry pi GPIO and physical computing: GPIO pins overview. Accessed: Mar. 5, 2026. URL: https://learn.sparkfun.com/tutorials/introduction-to-the-raspberry-pi-gpio-and-physical-computing/gpio-pins-overview.

[27] Q. Xiong, F. Wu, Z. Lu, Y. Zhu, Y. Zhou, Y. Chu, C. Xie, and P. Huang. Characterizing 3D floating gate NAND flash. In *Proceedings of the ACM SIGMETRICS*, Urbana-Champaign, IL, USA, June 2017. doi:10.1145/3078505.3078550.

Appendix A

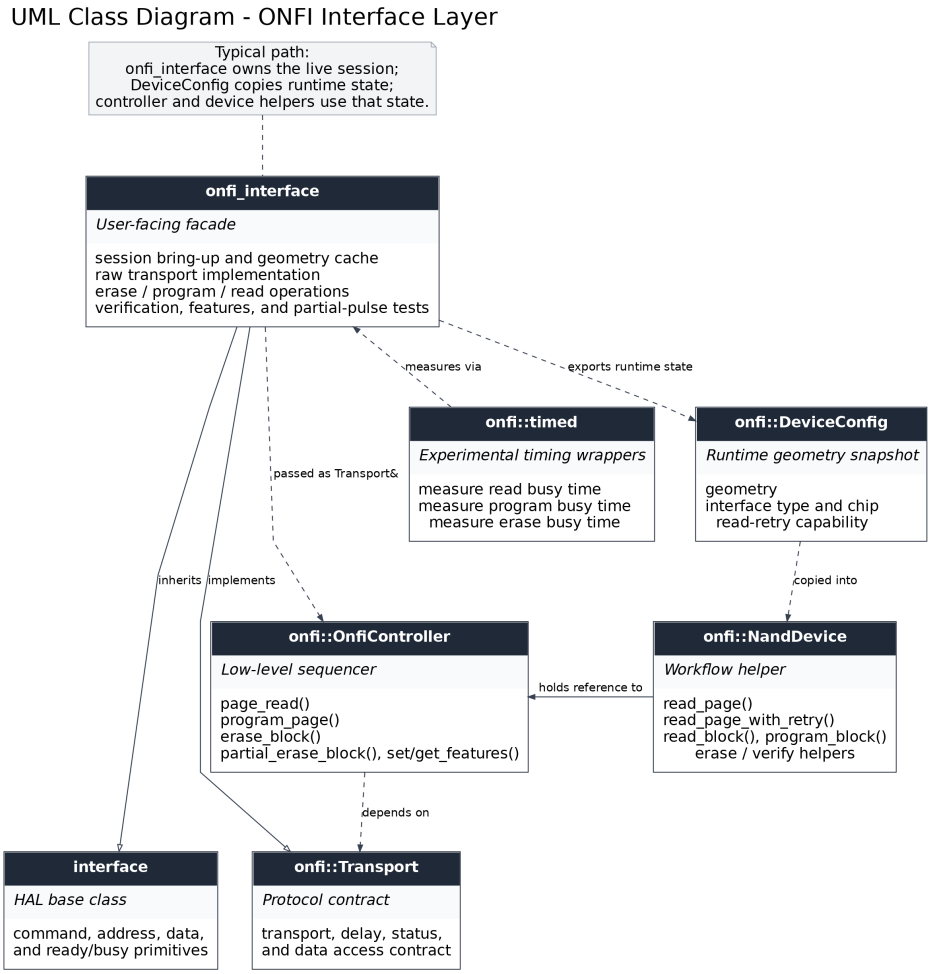


Figure A.1: Structural overview of the ONFI interface layer, showing how onfi_interface bridges the hardware abstraction layer, transport contract, controller, and higher-level device workflows.

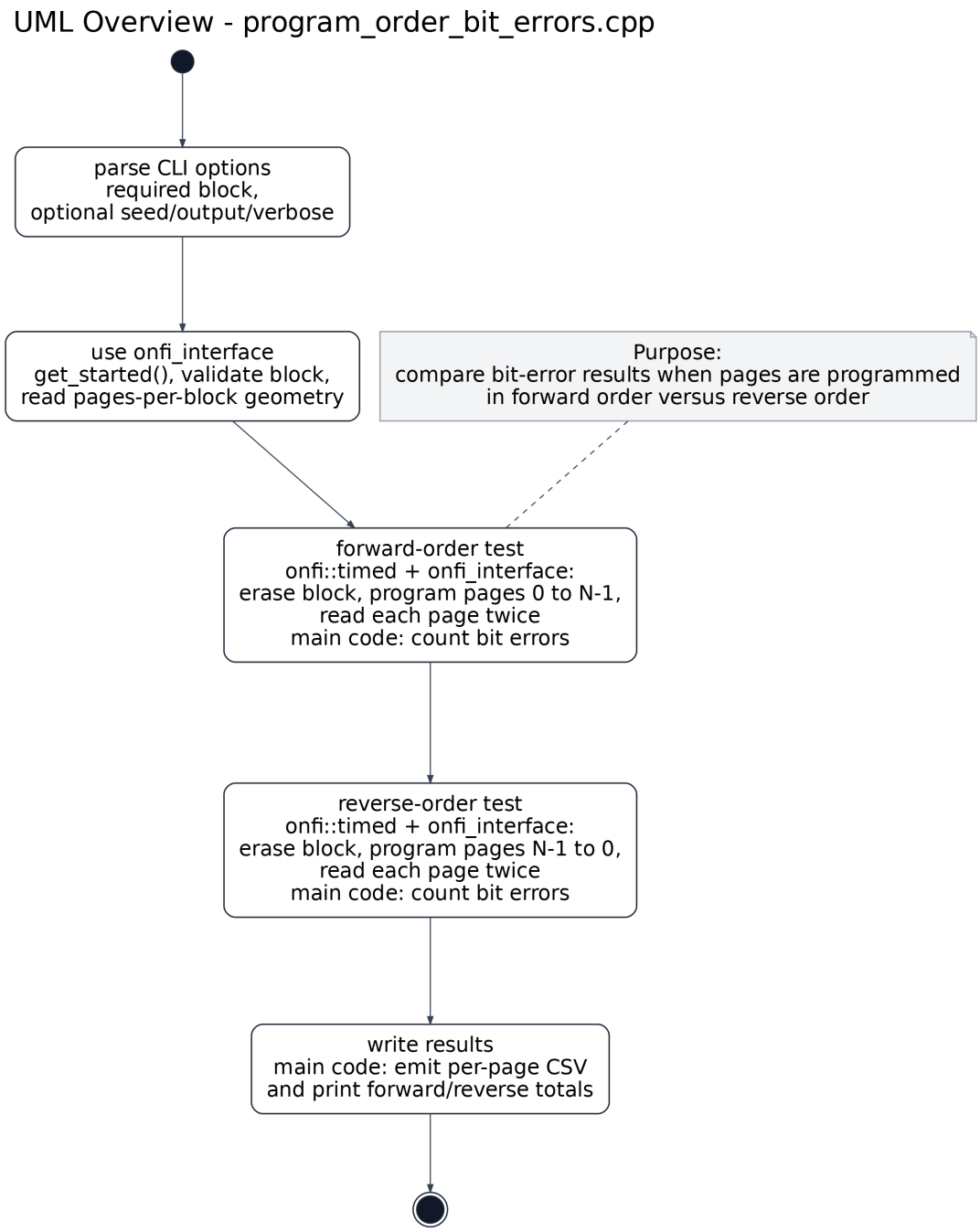


Figure A.2: Overview of program_order_bit_errors.cpp, showing forward-order and reverse-order page programming with timed ONFI operations and per-page bit-error comparison.

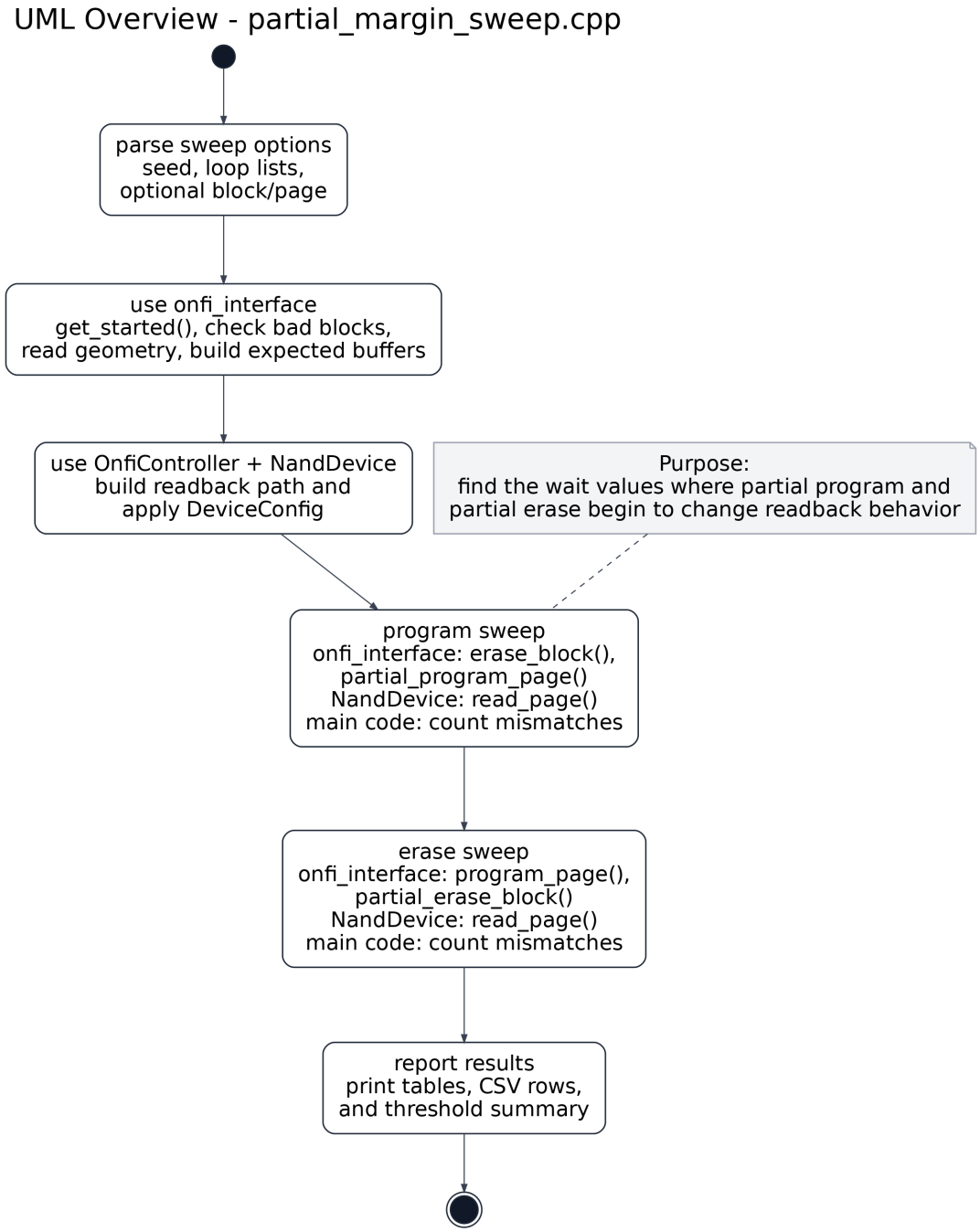


Figure A.3: Overview of partial_margin_sweep.cpp, showing how onfi_interface applies partial program and partial erase operations while NandDevice reads data back for mismatch analysis.

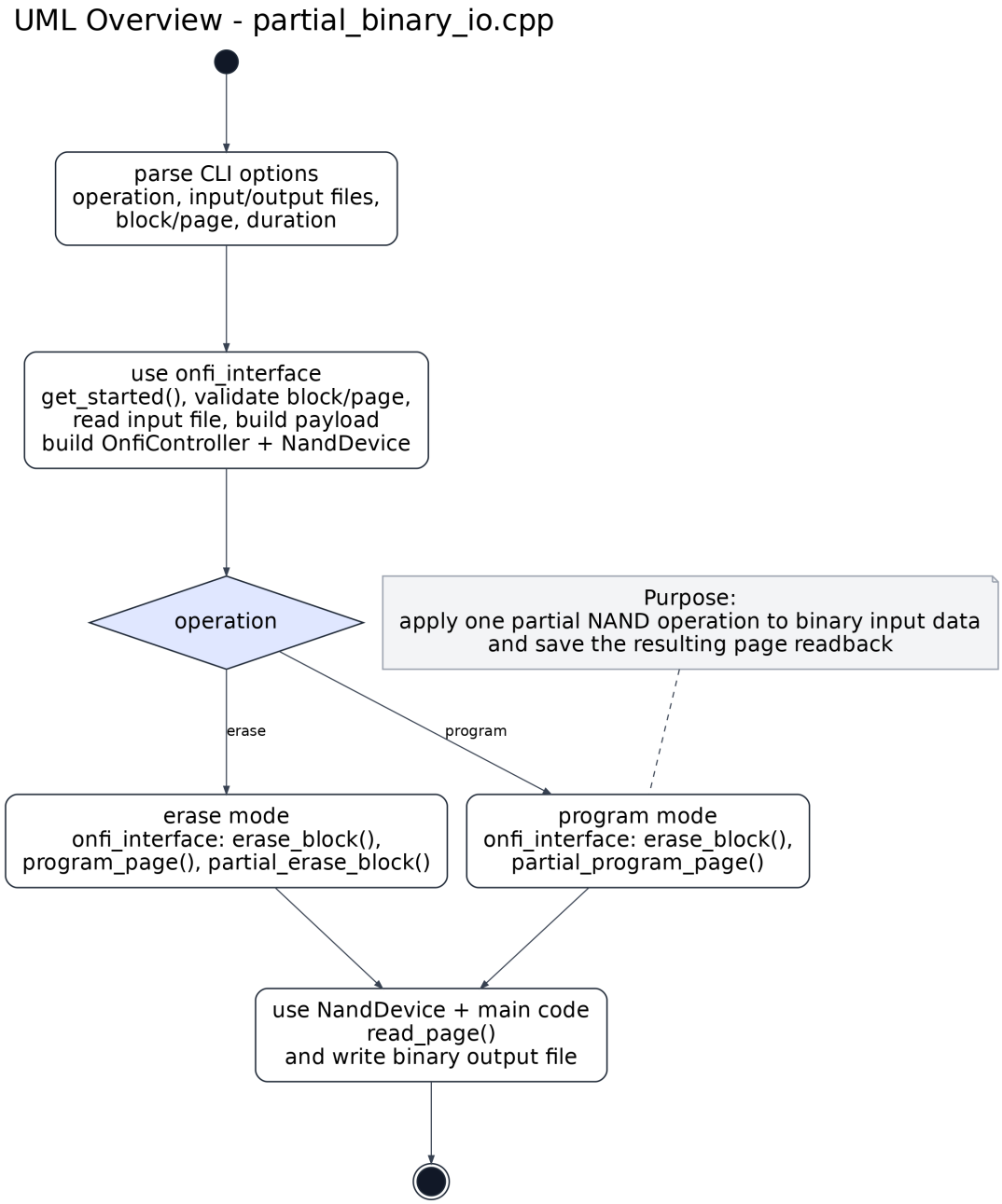


Figure A.4: Overview of partial_binary_io.cpp, showing how binary input data is written with a partial NAND operation and then read back to an output file.