2018

Implementing Write Compression in Flash Memory Using Zeckendorf Two-Round Rewriting Codes

Vincent T. Druschke
Michigan Technological University, vtdrusch@mtu.edu

IMPLEMENTING WRITE COMPRESSION IN FLASH MEMORY USING ZECKENDORF TWO-ROUND REWRITING CODES

By
Vincent Druschke

A REPORT
Submitted in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE
In Computer Engineering

MICHIGAN TECHNOLOGICAL UNIVERSITY
2018

This report has been approved in partial fulfillment of the requirements for the Degree of MAS-TER OF SCIENCE in Computer Engineering.

Department of Electrical and Computer Engineering

Report Advisor:	*Dr. Saeid Nooshabadi*
Committee Member:	*Dr. Roger Kieckhafer*
Committee Member:	*Dr. Soner Onder*
Department Chair:	*Dr. Daniel R. Fuhrmann*

Contents

List of Figures

List of Tables

Definitions

block
The smallest unit of a flash memory that can be **erased**.

cell The smallest physical component of a memory from which a meaningful value can be interpreted.

cluster
A fixed-size portion of a file which constitutes the smallest logical unit of file storage.

compression ratio
Typically, the ratio of the amount of information bits stored in a given memory over the number of memory bits occupied. Specifically used here to refer to the ratio of the number of information bits stored over *multiple writes* to a given location in memory, *without* requiring an **erase** operation, over the number of memory bits accessed during those writes.

data bit
A single binary digit representing an arbitrary data value either in part or whole. In particular, refers to the m-1 bit(s) immediately proceeding a **sentinel bit** in a **primary encoding** which can be used to represent additional data in a **secondary encoding**, where m is the *encoding degree*.

encoding
Generally, can be used to refer to any way in which information is represented (for example, base-2 binary, the decimal numeral system, or even the English language can all be thought of as ways of encoding information). In the context of this report, refers to any data represented as either a **primary**, **secondary**, or **padded primary encoding**.

encoding degree
The **series degree** of the **extended Fibonacci series** used when generating a **primary encoding**. Can also be applied to **padded primary encodings** and **secondary encodings**, as this value has implications for the function and interpretation of these encoding types as well.

erase
To restore a **cell** from its **programmed** state back to its original or default value, which can vary according to the design of the memory.

extended Fibonacci series
A recursive series based on the classic Fibonacci series ($A_0 = 1,\ A_1 = 2,\ A_n = A_{n-1} + A_{n-2}$), where each element is based on the sum of the first previous element and the **m**th previous element, where **m** is the series **encoding degree**. This includes the classic Fibonacci series, whose degree is equal to 2.

extension bit
Any bits which proceed the m-1 **data bits** in an **encoding** but appear before the next **separator bit**.

flash memory
A digital memory whose method of storage is an array of FG-MOSFETS arranged into relatively large erase blocks such that the memory can be erased "in a flash."

KS-code
A type of two-round **rewriting code**, named after researchers Klein and Shaoira who first proposed its use, which uses **Zeckendorf sums** to produce a first-round encoding to which additional data can be subsequently written to in the second round of writing.

padded encoding
A **primary encoding** containing the maximum number of **separator bits** possible for a given initial state while still maintaining the properties of a **Zeckendorf sum** and the locations of the separator bits from the initial state.

padding scheme
A method of adding bits to a pre-existing binary string; in this case, particularly refers to the process of adding as many additional s**separator bits** to a **primary encoding** as is possible without violating the properties of a **Zeckendorf sum**.

page The smallest unit of a flash memory that is physically accessible by the read/ write circuit.

primary encoding
A value or set of values encoded according to the binary representation of their **Zeckendorf sums**.

program
To alter a memory's **cells** from their default state, which can vary according to the design of the memory.

rewriting code
A state-based method of representing data such that the data can be altered in-place to represent some value other than its original value, typically while maintaining certain writing properties, such as not requiring bits to be **erased** in order to be updated.

secondary encoding
A **primary encoding** which has had its **data bits** set to represent additional information.

separator bit
A bit with a logical value of '1' in a **primary encoding** or **padded primary encoding**. Denotes the beginning of a set of data bits in a **secondary encoding** (although its value does not contribute to the value of the stored data in this case).

series degree
An integer which represents the number of prior series elements upon which an **extended Fibonacci series** depends.

solid-state memory
Any type of digital memory which does not use moving parts, and is able to represent data values through static electrical circuits alone.

standard encoding
The standard base-2 binary method of representing data values.

Zeckendorf encoding
A binary representation of a **Zeckendorf sum** wherein each bit position represents a member of an **extended Fibonacci series**, rather than a power of 2.

Zeckendorf sum
A summation of a subset of the elements in a given **extended Fibonacci series** in which each element in the series appears at most once and no elements which are within m-1 positions of

each other appear in the sum, where m is the *series degree* of the extended Fibonacci series being summed.

List of Abbreviations

API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
FGMOSFET	Floating-Gate Metal-Oxide-Semiconductor Field-Effect Transistor
FPGA	Field Programmable Gate Array
FTL	Flash Translation Layer
HDL	Hardware Description Language
IP	Intellectual Property
SSD	Solid-State Drive
WOM	Write-Once Memory

Abstract

Flash memory has become increasingly popular as the underlying storage technology for high-performance nonvolatile storage devices. However, while flash offers several benefits over alternative storage media, a number of limitations still exist within the current technology. One such limitation is that programming (altering a bit from its default value) and erasing (returning a bit to its default value) are asymmetric operations in flash memory devices: a flash memory can be programmed arbitrarily, but can only be erased in relatively large batches of storage bits called blocks, with block sizes ranging from 512K up to several megabytes. This creates a situation where relatively small write operations to the drive can potentially require reading out, erasing, and rewriting many times more data than the initial operation would normally require if that write would result in a bit erase operation. Prior work suggests that the performance impact of these costly block erase cycles can be mitigated by using a rewriting code, increasing the number of writes that can be performed on the same location in memory before an erase operation is required. This paper provides an implementation of this rewriting code, both as a software program written in C and as a SystemVerilog FPGA circuit specification, and discusses many of the additional design considerations that would be necessary to integrate such a rewriting code with current file storage techniques.

1. Introduction

Modern computing devices are rapidly adopting **flash memory** as a means of persistent data storage. Offering lower power requirements, smaller form-factors, and superior performance in comparison to "traditional" types of non-volatile memory, all at an increasingly competitive price per unit of storage, flash has become an attractive storage solution for both mobile computing and high-speed file access [1]. Unfortunately, due to the physical properties of the underlying technology and the constraints that these properties impose on flash memory architecture, these benefits also come with certain inherent limitations that can impact the performance and longevity of flash-based memories [2]; while the benefits to using flash technology in memory devices still far outweigh the potential caveats in doing so, circumventing or mitigating the effects of these limitations is nevertheless essential to maximizing its utility. The object of this report is to address one such limiting factor, specifically that flash memories are restricted in their ability to modify data values once written, by providing two novel implementations of a method of data coding. First proposed in prior work [3] as a potential method of circumventing the memory modification restrictions of flash, this report details the motivation, theory, and mechanics of this coding; its implementation and performance in both hardware and software; and its potential application to data storage on a flash-based mass storage device.

1.1 Scope and Outline

The chapters below cover a number of topics relating to the motivation for this work, i.e. the construction and mechanics of flash memories, in addition to detailing the theoretical background of the system of coding being implemented, but the scope of the implementation(s) itself is actually quite narrow. The goal in producing these implementations was simply to adapt the abstract methods of data rendering so described in order to successfully encode and decode some known data; it is important to note that, although the implementations detailed herein are complete in the sense that they are capable of applying the prescribed data transformations which constitute the aforementioned coding method, they do not themselves facilitate its primary function: to store and manage encoded data in flash memory such that memory performance is improved. Instead, these implementations are offered as the first piece of the puzzle in applying these data processes to flash storage devices, providing the necessary shuffling of bits to translate data to its encoded form for storage and back to a usable state once retrieved while leaving the tasks of figuring out where and how to store the encodings in a way that make sense to other devices; while these tasks are fairly straightforward in most systems, applying this coding system complicates things somewhat, as chapter 4 illustrates.

This chapter, chapter 1, explains the construction of flash memory, how it operates, and why such memories are subject to certain limitations. Chapter 2 summarizes the contents of [3] that detail the theoretical basis of the code being implemented, providing a high-level description of its function and purpose. Chapter 3 describes the implementations themselves, providing an overview of their operation from a user's perspective as well as some general insight into the logic by which they operate. As mentioned, chapter 4 discusses the application of this code to storing data, particularly in

flash, and how the results of the encoding process are often at odds with the objective of maximizing memory performance. Finally, Chapter 5 concludes this report with a brief summary of the topics covered, the results and significance of the work, and what yet remains to be done.

1.2 Flash Memory Technology

As opposed to the magnetic storage devices commonly associated with mass data storage applications, flash memories are a form of electronic or **solid-state memory**, meaning that they are able to represent data purely through the use of electronic circuits, with no moving parts. In flash (and other solid-state memories), a **cell** - the most basic physical storage element in a memory from which a value can be meaningfully interpreted - is made up of just a single, specialized field-effect transistor called a floating gate transistor, or FG-MOSFET [4]. Floating gate transistors are built with two electrically isolated gates: a control gate (which functions as a normal transistor gate), and a "floating" gate that is insulated from both the control gate and the transistor substrate by two dielectric layers. By constructing the transistor in this way, charged particles (i.e. electrons) can then be induced onto the floating gate under specific conditions, altering the transistor's response to voltages on the control gate.

1.2.1 Reading and Writing Flash Cells

FG-MOSFETs are charged via two primary mechanisms: hot-carrier injection and Fowler-Nordheim tunneling [4], [5]. To utilize hot-carrier injection, a high positive voltage is applied between the transistor control gate and its source, and a smaller positive voltage is then applied from drain to source. High current flows from the drain to the source, but some of the charge carriers in that current have enough energy to overcome the resistance of the oxide surrounding the floating gate and get trapped. Fowler-Nordheim tunneling takes a slightly different approach: instead of pulling a current through the transistor to induce charge carriers to move through the oxide (which can be destructive to the oxide material), Fowler-Nordheim uses a high positive voltage between the control gate and the transistor substrate to induce a high electric field in the transistor, causing some electrons to "tunnel" through the oxide and again embed themselves in the floating gate. In both cases, charge is removed via tunneling by charging the gate and the substrate with a voltage of opposite polarity to that used charging (i.e. a voltage drop from the substrate to the control gate).

The charges stored in the floating gate of an FGMOSFET modify the functioning of the underlying field effect transistor by increasing the threshold voltage required to cause it to conduct current. A negatively charged floating gate works to counteract the field induced by the positive control gate voltage, causing cells whose floating gates have been charged to conduct less current than they would otherwise when a voltage is applied to their control gates. This property can then be exploited to interpret logic values from the differently charged cells as follows: connect the source and drain terminals of the cell to a read circuit and apply a test voltage, V_{read}; an uncharged cell will readily conduct current from its source to its drain when V_{read} is applied to its control gate, while a transistor that has been written to by having electrons stored on its floating gate will produce a proportionally lower current. A logic value can then be associated with each current level, usually 1 for high current and 0 for low current [5].

1.2.2 NAND and NOR Flash

There are two common types of flash memory: NAND and NOR, named so for the way in which the operation of the cells of these memories resembles the operation of a NAND or a NOR logic gate, respectively [4]. In both NAND and NOR memories, cells are arranged in a grid, with each "row" defined by a common word line connected to the control gate of each cell in the row, and each "column" sharing a bit line which connects to the drain of one or more of the cells in the column. The key difference between NAND and NOR flash is that, in a NOR flash, each cell in a column connects to the bit line individually and in parallel, while NAND flash allows multiple cells to share a single connection to the bit line by wiring groups of them in series from source to drain, with only the first transistor in the series connecting to the bit line directly. Each cell (or group of cells) then has its remaining source terminal connected to a source line, which completes the read circuit.

To read a given row in a NOR flash, the word line for that row is energized to V_{read} while all other rows on the same bit are set to a lower voltage, V_{off}. If the floating gate of the cell(s) being read are not charged, current flows between the bit line and the source line(s) and the flash memory read circuit will interpret this as a logic 1; charged cells . NOR flash gets its name from the fact that this read operation resembles the operation of a NOR gate: if all gates on a bit line are open, the bit line voltage remains high (as the source line is typically connected to ground), but if a single gate conducts the bit line is pulled low [5].

Reading from a row in NAND is slightly less straightforward than in NOR flash due to the wiring of cells in series. To read from a NAND memory, the word line of the row to be read is again energized to V_{read}, but, in order for the entire series-cell group to conduct from their shared source line to the bit line, the rest of the cells in the group must also conduct. To accomplish this, the word lines which connect to the other cells in the group are set to some voltage, V_{on}, which is greater than V_{read}, forcing these cells to conduct regardless of their charge state. If the cell(s) in the row being read are uncharged, they will once again close when V_{read} is applied and the entire series of cells in each column will conduct from bit line to source line, causing a logic 1 to be interpreted. Otherwise, a charged gate will cause the series to fail to conduct, resulting in a read value of 0. As in a NAND gate, if all gates close the bit line will be pulled to ground, while a 0 on any of the series gates will allow the bit line to remain high [5].

NAND flash contains fewer connecting elements per cell than a comparable NOR flash, giving NAND-based flash designs a significant storage density advantage and making them smaller and cheaper to produce than a NOR memory of equivalent capacity [6]. NAND flash also benefits from faster write speeds compared to NOR flash. These two features make NAND flash the preferred memory type for flash-based file storage, where high capacity and the ability to quickly write data are higher priorities. By contrast, NOR flash offers faster random read times due to the fact that each cell is directly connected to a bit line; NAND flash offers similar *sequential* read performance (i.e. reads to consecutive rows in the same group), but is much slower to read randomly. Because of its ability to perform fast random access reads, is preferred for ROM-type storage and code execution, but is slower to program and erase, and is thus not suited for large-scale file data storage from either a cost or a performance standpoint. Because the focus of this report is on improving write performance for flash storage devices, our implementations are primarily applicable to NAND flash memories, although the principles still technically apply to both.

1.3 Solid State Drives

A secondary storage device that uses flash (particularly NAND flash) memory to store data is called a **solid state drive**, or SSD. The flash memory in an SSD is physically organized into a hierarchy of progressively larger units of storage according to the structural relationships between the cells therein. Flash memory cells are arranged, in order of increasing units of size, as follows: a group of cells connected by a common word line constitutes a **page**; a group of pages embedded in a common substrate constitutes a **block**; groups of blocks which share an independent set of accessing hardware are known as **planes**; and one or more planes connected to a single memory controller, which is responsible for interpreting and executing memory commands [7], [8]. Pages and blocks, in particular, are of special significance to the operation of the SSD due to the way cells are structured in these organizational units.

Because the word line is the component that is used to select a row of cells for reading or writing, all cells on the same word line are accessed at the same time when the word line is asserted. This makes page size a defining feature of the device architecture, as each read and write operation to an SSD must be facilitated at the hardware level in units of pages; not all of the cells of a page will necessarily have their values changed or transmitted back to the accessing device, but an operation on part of a page can occur no faster than the time it takes to access the whole page, so aligning access to page boundaries - reading or writing to one full page instead of two half-pages whenever possible, for example - is necessary for optimal drive performance. Chapter 4 discusses page size and its effects on storage performance in more detail.

All memories must be writable at least once to be given a meaningful value, but in order for a memory to be *re*writable, it must be capable of reversing this process as well. For a memory with some preferred initial state or value, the processes of altering and restoring this value to the cells of such a memory are known respectively as **programming** and **erasing**; in an SSD, programming a cell thus sets its value to 0 while erasing the cell returns its value to 1. Many memories, including magnetic storage devices, can be programmed and erased symmetrically, meaning that programming and erasing are roughly equivalent processes, only differing in the value they assign to a memory's cells [8]. This allows such memories to rewrite previously programmed memory locations in-place, directly programming and erasing cells as necessary without any additional steps. Flash memories, however, are *asymmetric* write devices; their cells can be programmed in pages, but can only be erased an entire block - usually 64 or 128 pages - at a time.

Erasing in blocks means that writes to occupied regions of the device can potentially be much slower than writes to "empty" drive space; if a write to just a single page in a block requires a 1 to be written where a 0 was previously, the SSD then has to make a copy of each page in the block that contains valid data before erasing the entire block, writing the new data to the page that was originally being accessed, then re-writing the saved page data back to all the pages that were copied out beforehand. These additional operations are all performed by the SSD memory controller and happen transparently to the accessing system, so an occasional read-erase-write cycle has fairly minimal impact on drive performance, but frequent writes of this type can severely reduce the average write speed of the device [9]. This block erase property is one of the main limitations of this technology, and is the primary motivation for this report.

1.3.1 SSD Memory Management and the Flash Translation Layer

Fortunately, modern flash storage devices have a number of techniques at their disposal to reduce the occurrences of this worst-case behavior and avoid having to perform costly read-erase-write cycles: nearly all SSDs incorporate some form of address translation, called the flash translation layer (FTL), in order to abstract their physical memory from the logical addresses used to access the device [10] [11]. Essentially, the FTL allows the SSD to maintain a dynamic one-to-one or many-to-one association of logical storage addresses to physical pages in flash. This provides accessing devices with a consistent *logical* interface to the SSD (i.e. data written to address n will always be found at address n) while allowing the SSD memory controller a much greater degree of control over how data is managed physically.

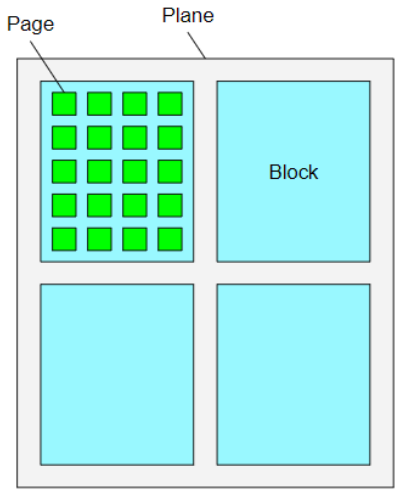


Figure 1.1: A mock-up of the SSD's internal memory structure, with pages in green, blocks in blue, and the plane in gray

One advantage to this approach is that the SSD can process write requests intended to update some data already present at a given logical address by writing the new data to a **fresh** page, one that has not been written to since it was last erased, and updating the page association for that address in the address translation table. This method of writing allows the SSD to avoid having to erase blocks when storing modified data while providing seamless operation from the perspective of the accessing device. The previous page is then marked as **stale** to indicate that it no longer contains useful information and can be safely erased at a later time; one strategy used by SSD memory controllers is to try to group as many of these stale pages as possible into the same block to maximize the efficiency of each erase operation. Unfortunately, as more information is written to the drive and the number of unoccupied pages dwindles, the SSD is left with fewer possible pages in which to store modified data, increasing the likelihood of a read-erase-write cycle occurring. As a method of augmenting the SSDs own storage management abilities to further reduce the occurrence of block erase cycles at write time, a type of data encoding known as a **rewriting code** is proposed to enable restricted in-place modifications to data values in flash [3].

1.3.2 Write-Once Memories and Rewriting Codes

The concept of rewriting codes was originally developed as a method of allowing multiple writes to write-once memories, or WOMs [12]. As the name implies, a WOM is a type of memory in which the act of programming a bit permanently and irreversibly alters the physical structure of the storage mechanism, meaning that this type of memory can usually only be practically written once: unaltered bits can still technically programmed, but programmed bits cannot be erased, making useful modifications to such memory unlikely at best without a methodology for organizing these bits. Rewriting codes are designed to do just that; by encoding information in such a way as to preserve a specific set of bits for future writes, these codes can enable arbitrary repeat writes to a storage medium without the need (or the ability) to erase bits. Apart from allowing limited re-use of physically limited write-once media, this concept can be extended to other memories such as flash; while not technically a WOM in and of itself, flash can be considered a type of quasi-WOM due to the relative difficulty of erasure on such a device.

To preserve a write-once memory for future writes, previously developed codes have taken advantage of sparse encoding techniques to spread out the information contained in a given value over a larger-than-normal memory area while ensuring that a certain number of the bits therein remain un-programmed [12], [13]; these sparsely encoded initial values can then be modified by writing to the preserved memory bits, thus "rewriting" that value in memory. Because they occupy more space in memory, rewriting codes generally present a trade-off between total storage volume (i.e. the amount of information that can be represented by the memory at a given instant) and number of possible rewrites for a given memory. At first glance, this would seem no better than just artificially limiting the amount of data that can be written to the memory at one time, thus ensuring that some memory regions will be available for additional writing without the need for a complex data coding process. However, apart from securing some number of additional writes by virtue of the way in which it represents information, the real benefit of a rewriting code is that it can actually allow more information to be written to the memory cumulatively over the total number of writes supported than would have been otherwise possible; even though each encoding, in and of itself, requires more memory to represent the same value, the total number of data bits it is able to represent over all writes should, ideally, be greater than the number of memory bits needed to store that encoding.

Because of this property, a rewriting code can be thought of as a type of compression algorithm, but instead of deriving its compression performance from the reduction of a single data object's memory footprint as much as possible for a single write, a rewriting code, as its name implies, attempt to maximize the utility of the memory over *multiple* writes. One obvious drawback to this approach is that, in many of these rewriting codes, the rewriting process is destructive to the existing data; whereas writing two data sets encoded using a traditional approach to data compression allows for the retrieval and decoding of both sets once written, a rewriting code typically only supports the retrieval of the most recently encoded values [12], [13]. However, it is also possible to use rewriting codes as "compression boosters," applying them in conjunction with other methods of data encoding to achieve a combined compression effect [3].

2. The KS-Code

Prior literature suggests the use of a particular rewriting code [3], which this report refers to as the **KS-code**, based on a method of representing integral values as a particular summation of the members of the Fibonacci series known a **Zeckendorf sum**. These sums hold certain properties that, when represented in binary, guarantee a particular arrangement for the represented data which can then be exploited to allow a second write "into" a previously stored value.

2.1 Primary Encodings

While there are infinitely many ways of arriving at a given integer by summing members of the Fibonacci series, two properties must be obtained in order for such a sum to be considered a Zeckendorf sum: first, no elements in the series may be repeated - each Fibonacci number can appear at most once in the sum; second, the sum must not contain any two *consecutive* Fibonacci numbers, as Zeckendorf sums are *minimal* sums of the Fibonacci series, and any two consecutive Fibonacci numbers can be represented more minimally by replacing them with the value of their sum, which is also a Fibonacci number by definition [14]. As an example, consider the number 29: while it is natural to think of this number as the sum of the numbers 20 and 9 in base-10, or even $2^4 + 2^3 + 2^2 + 2^0$ in (base-2) binary, we can also represent this value, using Fibonacci numbers, as the sum of 21 and 8, which are the 8^{th} and 6^{th} numbers in the Fibonacci series, respectively. Zeckendorfs theorem posits that any integer can be uniquely represented as the sum of one or more non-consecutive members of the Fibonacci series [14], and it is this property of non-consecutiveness that forms the basis of the KS rewriting code.

Of course, when storing these encodings on a computer, there are still only two digital values with which to represent them, so how is this done efficiently? Fortunately, because Zeckendorf sums contain no repeat elements (i.e. it will never be necessary to count more than 1 of each Fibonacci number in the sum), these sums can be encoded as a simple binary sequence, where a 1 represents the presence of a given Fibonacci number in the sum, and 0 the absence. After all, the typical base-2 binary encoding, or **standard encoding**, merely represents the sum of a series of consecutive powers of two; by mapping each index of a string of binary digits to a member of the Fibonacci series rather than a power of two, a Zeckendorf sum can be represented in the binary numeral system, producing a **Zeckendorf encoding** [14]. Returning to the above example, the Zeckendorf encoding of 29 can be produced by examining the first 9 members of the Fibonacci series, which are (in descending order): $21, 13, 8, 5, 3, 2, 1, 1$, and 0. Rewriting 29 as a Zeckendorf encoding, with its Zeckendorf sum members being 21 and 8, produces $21 * 1 + 13 * 0 + 8 * 1 + 5 * 0 + 2 * 0 + 1 * 0$ (omitting the first two elements in the series, as the repeated 1 is unnecessary and the value of 0 is implied by an empty sum), or just 101000. More generally, it is possible to represent any Zeckendorf sum Z as the sum-product of a series of n bits, b, and the Fibonacci series F (again, omitting the first 0 and 1), or:

$$Z = \sum_{i=0}^{n-1} F_i b_i = F_{n-1}b_{n-1} + F_{n-2}b_{n-2} + \ldots + F_0 b_0$$
$$F_0 = 1,\ F_1 = 2,\ F_i = F_{i-1} + F_{i-2}\ for\ i \geq 2 \tag{2.1}$$

The Zeckendorf sum of a given (non-negative) integer value can be calculated in linear time with respect to the size of the input value using a greedy algorithm [14]: for any number in the Fibonacci series that is less-than or equal to the input value, starting with the largest Fibonacci number that satisfies this condition, add that series element to the sum and subtract its value from the value of the input, then move on to the next smaller Fibonacci number and repeat this process with the remainder of the input value until that remainder becomes 0. Translating data from its standard form to this new Zeckendorf form constitutes the encoding portion of the first round of the KS-code, the product of which will be referred to as a **primary encoding**. A primary encoding can then be decoded by simply calculating the unsigned, base-2 integer sum of the series elements in the encoding.

2.2 Secondary Encodings

As the previous section alludes, Zeckendorf sums posses an important property for this rewriting code: each of these sums must, by definition, be composed of non-sequential members of the Fibonacci series. Applied to the binary encoded forms of such sums, this property implies that any two 1-bits in the binary string of such an encoding must be separated by at least one 0-bit; stated another way, it can be assumed that each 1 in an encoding will be followed by at least one 0 (with the notable exception of the last bit in the series). These leading 1s are referred to as **separator bits**, with the bit immediately following each separator being designated a **data bit** [3]. Note the significance of these so-called data bits: the primary encoding guarantees that each 1 is followed by a 0, so the values of these data bit in the primary encoding are initially implicit. As they have a fixed value of 0 and thus do not represent any value in the encoding, their only function in this case is to act as placeholders to denote the position of the "skipped" elements in the encoded series' summation. Once a primary encoding has been produced, the actual values of the data bits therein are essentially irrelevant to the interpretation of the Zeckendorf summation's value (as they can be inferred as 0's during the decoding of the encoded sum and skipped entirely), meaning that it is possible to alter the values of these bits without actually changing the interpretation of the initial encoding. Thus, for each separator bit that appears in a binary-encoded Zeckendorf sum, an additional bit of data can be stored by setting the corresponding data bit to the desired value. The process of writing to the data bits in a primary encoding constitutes the second round of the rewriting code, with the "re-written" primary encoding becoming a **secondary encoding**. Both data sets can still be retrieved from the encoding after a second-round write by scanning through the resulting secondary encoding and identifying the separator-data bit pairs; the separator bit can still be interpreted as a member of the Zeckendorf sum used to encode the primary data value, and the data bit represents one bit of the secondary data value. More to the point, because each data bit has the same value initially, it is possible to perform this secondary write operation without causing any bits to be erased.

Unfortunately, Zeckendorfs theorem only implies *at least* one 0-bit after each sentinel, meaning that no assumptions can be made about any possible additional bits based solely on the encoding itself, and it is for this reason that only one additional bit of data can be stored in the secondary

encoding for each separator bit. More specifically, each separator-data bit pair in a primary encoding may or may not be followed by one or more 0-bits; if we were to use these bits for secondary data storage, setting any of these bits to a value of 1 would make them indistinguishable from another separator bit during decoding (as it can only be assumed that the *one* bit after each separator will not be another separator). These additional bits are called **extension bits**, and a source of performance loss for this rewriting code. Mitigating the deleterious effects of these extension bits on the overall compression performance of the encoding is key to maximizing the effectiveness of the rewriting code.

2.2.1 Padding

While the ability to store multiple data values in a single Zeckendorf encoding can be potentially beneficial, this method of storage is highly inconsistent from a compression performance perspective: the total number of data bits which can be stored within the secondary encoding is entirely dependent on the data that was initially used to create the primary encoding. Again, secondary data can *only* be stored in the single bit immediately following each separator 1-bit in the encoding, so an encoding in which the Zeckendorf sum of the input value contains fewer elements (and thus fewer 1's) will have very little potential for secondary storage. In general, a binary-encoded Zeckendorf sum of length n can contain anywhere between 0 and $\frac{n}{2}$ separator bits, with an encoding containing no 1's being the worst case (i.e. an encoded value of 0), and a pattern of alternating 1's and 0's (which corresponds to the largest encoded value possible for a given number of encoding bits) representing the optimal case for secondary encoding. To mitigate this uncertainty problem and minimize the number of extension bits present in the encoding, a **padding scheme** is added to the rewriting process to "insert" as many 1-bits into the encoding as possible (without violating the sparseness property) in order to maximize the number of data bits available for the secondary encoding [3]. Padding a primary encoding raises the lower bound on the number of data bits in the encoding from 0 to $\frac{n}{3}$ for an n bit encoding: sets of three or more consecutive 0's in a bit string can accept an additional 1 without creating any adjacent 1's, but any pattern which contains just two consecutive 0's cannot be padded further without violating the all-important sparseness property, resulting in one separator bit (and thus one data bit) for every *three* bits in the worst case. The padding process is also irreversible, meaning whatever data value that was initially used to produce the primary encoding will be destroyed. As such, padding is only used to condition data which is no longer needed in order to maximize the utility of the data for storage in the secondary writing process. An illustration of the padding and secondary encoding process is shown in figure 2.1.

1	0	0	0	0	0	0	1	0	0	1	0
1	0	$\underline{1}$	0	$\underline{1}$	0	$\boxed{0}$	1	0	$\boxed{0}$	1	0
1	**1**	$\underline{1}$	**0**	$\underline{1}$	**0**	$\boxed{0}$	1	**1**	$\boxed{0}$	1	**1**

Figure 2.1: A Zeckendorf encoding ($F_{11} + F_4 + F_1 = 243$) (top) is padded (center), with additional separator bits underlined and extension bits boxed; the encoding is then overwritten with the number 19 (bottom), shown in bold.

2.3 Extended Fibonacci Series

While the description and examples of the KS-code thus far have focused on summations of the classic Fibonacci series, the principles of the rewriting code described can be applied to any method of encoding with similar properties. Recall the Fibonacci series F, recursively defined as $F_i = F_{i-1} + F_{i-2}$ with base cases $F_1 = 2$ and $F_0 = 1$. By modifying this recursion, it is possible to create a related series which, when used to encode data in the same fashion as before (i.e. binary-encoded series sums), produce binary strings with variations of the sparseness property described above. In fact, the Fibonacci series belongs to an infinite set of recursive series whose members can be described generally as any series A with a **series degree** of m, denoted as $A^{(m)}$, such that $A_i^{(m)} = A_{i-1}^{(m)} + A_{i-m}^{(m)}$, with base cases $A_i^{(m)} = i + 1$ for $0 \leq i < m$ [15]. This allows us to produce alternate primary encodings, based on any one of these $A^{(m)}$ series for some fixed degree $m \geq 2$, [1] which posses the property that each 1-bit in the encoding will be followed by at least $m - 1$ 0s (again with the exception of separator bits appearing in the final $m - 1$ bits of the encoding). All such encodings remain compatible with the padding and secondary read/ write processes as applied to the Fibonacci (or $A^{(2)}$) encoding scheme, but with additional data bits read and written after each separator. These extended Zeckendorf encodings can thus be used to fine-tune the overall compression algorithm by leveraging higher order encoding series to modify the sparseness properties of the primary encoding, which may prove to be advantageous for encoding different types of data or for different usage patterns.

2.4 Compression Performance

Like other rewriting codes, data encoded in the initial round of the KS-code is expanded, requiring more memory bits to represent than it would in its standard form; over both rounds of writing, the goal is to cumulatively write more bits of *information* than the number of memory bits written to, with the **compression ratio** of the encoding defined as the ratio of the total amount of information written over the number of memory bits used. In the example above, the number 29, which is a 5-bit value in its standard form, is represented as a 6-bit binary Zeckendorf sum, 101000, for a round-one expansion factor of $\frac{6}{5} = 1.2$; this value can then be padded with an additional separator bit, to 101010, for a total of 3 secondary storage bits and an overall compression ratio once overwritten of $\frac{5+3}{6} = 1.333$. For a sufficiently large number of bits encoded, n, this expansion can be approximated by $\frac{n}{\log_2(\phi)} \approx 1.44n$, or an expansion of roughly 44% over the standard encoding [3]. However, like other rewriting codes, this particular method of representing data makes up for the reduction in immediate storage capacity with improvements to the overall "write capacity" of the memory: the first write will store n bits of information in $1.44n$ bits of memory as a primary encoding, then a second write will allow re-use of up to half of these memory bits, for another $\frac{1.44n}{2}$ information bits stored, resulting in a best case compression ratio of $\frac{n+0.72n}{1.44n} = \frac{1.72}{1.44} \approx 1.194$. Even in the worst case, where only one in *three* bits can store secondary data, it is still possible to store $n + \frac{1.44n}{3}$ bits of information across the same $1.44n$ storage bits, yielding a compression ratio of $\frac{n+0.48n}{1.44n} = \frac{1.48}{1.44}$, or approximately 1.028. Thus, even in the worst case, the KS-code (with degree 2) still manages to store slightly more information bits over two writes than the number of storage bits that were used to represent that information.

[1] An interesting observation: the series $A^{(1)}$ actually represents the standard binary series, $1, 2, 4, 8, 16...$, and so data represented by the standard binary encoding could be said to represent a particular sum of this variant of $A^{(m)}$.

Using a series of a higher degree incurs a higher initial storage space penalty: as degree increases, the resulting series tends to grow more slowly, requiring more bits to represent a given value. This data expansion can generally be described as a function of degree m according to $n \log_{\phi_m}(2)$, where n is the number of bits to be encoded and m represents the dominant root of the characteristic equation of the series $Am, xm - xm - 1 - 1 = 0$; as m increases, ϕ_m asymptotically approaches 1, so the expansion factor, $\log_{\phi_m}(2)$, becomes increasingly large [3]. However, while the initial storage penalty is greater when compared to a lower encoding degree, a higher degree series can potentially offer better write compression overall due to the presence of additional secondary data bits; table 2.1 lists the minimum and maximum theoretical compression ratios for the first nine degrees of series encodings (starting with the Fibonacci series, $A^{(2)}$), as well as their overall primary expansion factors.

Table 2.1
Data Expansion Factor and Theoretical Min/ Max Compression for Encoding Degrees 2 to 10 Inclusive

Degree	Expansion Factor	Min CR	Max CR
2	1.440	1.0217	1.1957
3	1.813	0.9483	1.2069
4	2.151	0.8971	1.2206
5	2.465	0.8462	1.2051
6	2.763	0.8182	1.1932
7	3.047	0.7835	1.1856
8	3.321	0.7642	1.1698
9	3.587	0.7456	1.1667
10	3.846	0.7317	1.1545

An interesting pattern to note: minimum compression ratios go down as series increases degree due to the greater possible number of extension bits as a proportion of the total number of bits in the encoding, but maximum compression ratio increases up to a point, peaking at approximately 1.221 for an encoding degree of 4, before the number of bits needed to represent the encoding begins to outgrow the amount of information being represented. For lower encoding degrees, this means that there is a trade-off between maximizing potential secondary storage in the best case and mitigating storage waste in the worst case. Our analysis is restricted to encoding degrees 2 to 10, inclusive, as it would appear that higher order series, at least in theory, offer no compression benefit, as a steady decline in both minimum and maximum performance is observed for degrees greater than 4. It is certainly possible that some use case exists for higher degree encodings in practice due to some as-yet unforeseen property of the data, but, in general, higher degree encodings seem to yield diminishing returns on potential compression performance.

3. Implementation

We developed two implementations to facilitate the encoding and decoding of data according to the KS-code: one as a set of software functions, the other a hardware logic circuit design. We wrote the software first as a "proof of concept," allowing us to become familiar with the KS-code and its mechanisms and to develop the initial sketches for a general set of coding algorithms. Later, the software became useful as a method of testing file data properties, as several test batches could be performed fairly quickly on various files to determine how the code fared in terms of compression performance for various files. This source code is provided here with a header file in addition to main execution logic, and can thus be compiled as either a standalone utility or as a software library.

The hardware shares many of the logical design cues from the software, and is based on the same general algorithms, relying on sequential, clocked logic to drive the state-based calculations that make up the KS-code. Although we recognize that hardware *cannot* be written as software, and despite the fact that the specific implementations do differ in some key aspects, many of the coding processes are nevertheless highly state dependent, lending themselves much more readily to sequential designs. In hardware, each coding function is implemented as a separate module, with the total set of modules being packaged together to form a single encoding unit which is capable of performing the full suite of coding tasks. We developed these modules using the SystemVerilog HDL, which afforded a convenient means by which to define and test a circuit specification that could then be distributed and synthesized for any platform, be it an FPGA or some ASIC design. Source code for both designs is available in appendix A.

3.1 Algorithms

This section provides pseudo-code representations of the algorithms developed in the course of this implementation, illustrating the basic logic used for each coding process. The hardware and software implementations both share this same logic design, and so these algorithms can be thought of as the "blueprints" for their design.[1] Each algorithm is described in terms of the function it performs within the context of the coding scheme, with a separate name given to describe each of the five fundamental operations thereof: creating a primary encoding; decoding a primary encoding; padding a primary encoding; creating a secondary encoding; and decoding a secondary encoding. These operations or processes are respectively referred to as "encoding", "decoding", "padding", "overwriting", and "extracting". The first three of these are self-explanatory; overwriting and extracting, however, are new terms used to distinguish the encoding and decoding processes used in the second round of this rewriting code's operation from those in the first, and are in a sense more descriptive as to what task these processes perform.

[1] Specific implementation details, unless relevant to the content of this discussion, are omitted here for brevity and clarity. Implementation source code can be found in appendix A for complete details on how each of these algorithms are applied in their respective implementations.

Each of these algorithms relies on bit-wise access to at least one if its inputs or outputs, so it is important to note that such access in each of these algorithms is exclusively performed from most to least significant bit. Any access to a bit at a particular index should interpreted such that index 0 is the most significant, 1 is the next most significant, and so on.

3.1.1 Encoding Algorithm

```
Encode(DATA_SET, DEG):
    SERIES := {extended Fibonacci series of degree DEG}
    ENCODE_SET := {}

    while DATA_SET is not empty
      DATA := next element in DATA_SET
      ENCODING := 0 // Size in bits equal to length of SERIES

      for each ELEMENT in SERIES in descending order
        if DATA >= ELEMENT
          subtract ELEMENT from DATA
          set corresponding bit in ENCODING to 1
        endif
      endfor

      append ENCODING to ENCODE_SET
    endwhile

    return ENCODE_SET
```

For each discrete data element, DATA, in a set of data elements, DATA_SET, iteratively compare that element's value - interpreted as an unsigned integer - to the members of the extended Fibonacci series, SERIES, generated by degree, DEG, in descending order. If the value of the DATA is greater than or equal to that of the current ELEMENT in the SERIES, decrement the DATA value by the value of the ELEMENT and set the bit corresponding to ELEMENT in the ENCODING to 1; ENCODING bits correspond to SERIES elements such that the most significant bit in the ENCODING represents the largest element in the SERIES, the second most significant bit represents the second element, and so on. Continue this process until all SERIES elements have been considered, then append the resulting ENCODING to a list of encodings, ENCODE_SET, and begin the process again with the next element in DATA_SET until all elements in the set have been processed.

3.1.2 Decoding Algorithm

```
Decode(ENCODE_SET, DEG):
    SERIES := {extended Fibonacci series of degree DEG}
    DATA_SET := {}

    while ENCODE_SET is not empty
      ENCODING := next element in ENCODE_SET
      DATA := 0
```

```
    for each BIT in ENCODING
      if BIT == 1
        add corresponding element of SERIES to DATA
      endif
    endfor

    append DATA to DATA_SET
  endwhile

  return DATA
```

For each discrete encoding, ENCODING, in a set of encodings, ENCODE_SET, iteratively examine each BIT in the ENCODING, starting with the most significant bit. If the BIT has a value of 1, add the value of the corresponding element in the Fibonacci series, SERIES, and generated by degree, DEG, to the DATA value being calculated. Continue until all bits in the ENCODING have been examined, then append the resulting DATA value to a list of data values, DATA_SET, and begin the process again with the next encoding in ENCODE_SET until all elements in the set have been processed.

3.1.3 Padding Algorithm

```
Pad(ENCODE_SET, DEG):
  PADDED_SET := {}

  while ENCODE_SET is not empty
    ENCODING := next element in ENCODE_SET
    ZERO_COUNT := DEG - 1

    for each BIT in ENCODING
      if BIT == 0
        add 1 to ZERO_COUNT
      else
        ZERO_COUNT := 0
      endif

      if ZERO_COUNT == (2*DEG - 1)
        set the bit in ENCODING at DEG-1 positions prior to the ↩
            current BIT to 1
        subtract DEG from ZERO_COUNT
      endif
    endfor

    if ZERO_COUNT >= DEG
      set bit in ENCODING at ZERO_COUNT - DEG positions from the ↩
          end of the ENCODING to 1
    endif

    append ENCODING to PADDED_SET
```

```
    endwhile

    return PADDED_SET
```

For each discrete encoding, ENCODING, in a set of encodings, ENCODE_SET, iteratively examine each BIT in the ENCODING, starting with the most significant bit. Count the number of consecutive bits in the encoding by incrementing a counter for every BIT value of 0, and reset the counter if the BIT value is 1. The counter is initialized to DEG-1 to allow padding a 1 within the first DEG bits of the encoding, as the bits prior to each individual encoding have no effect on their correctness. If the count of consecutive 0 bits reaches DEG*2 - 1, pad this run of 0's by setting the bit located DEG-1 positions prior to the current BIT to 1, splitting the current string of 0's into two strings of DEG-1 0's separated by a 1, then set the counter to DEG-1 to reflect the reduction in the number of consecutive 0 bits. Continue until all bits in the ENCODING have been examined. Once the end of the encoding is reached, if the remaining count of consecutive 0's is still greater than or equal to DEG, place an additional 1 in the bit whose index matches this count, minus the degree DEG; this can be done due to the fact that there does not necessarily have to be a full DEG-1 0's after the final 1 bit in the encoding. Append the ENCODING to a list of padded encodings, PADDED_SET, and begin the process again with the next encoding in ENCODE_SET until all elements in the set have been processed.

3.1.4 Overwriting Algorithm

```
Overwrite(ENCODE_SET, DATA_SET, DEG):
    OVER_SET := {}

    while ENCODE_SET is not empty
      ENCODING := next element in ENCODE_SET
      WRITE_BITS := 0
      DATA_IDX := 0

      for each BIT in ENCODING
        if DATA_IDX == 0
          if DATA_SET not empty
            DATA := next element in DATA_SET
          else
            append ENCODING to OVER_SET
            return OVER_SET
          endif
        endif

        if WRITE_BITS > 0
          set BIT to value of bit in DATA at DATA_IDX
          decrement WRITE_BITS
          increment DATA_IDX, wrapping to 0 on the last bit
        else if BIT == 1
          WRITE_BITS := DEG - 1
        endif
      endfor
```

```
    append ENCODING to OVER_SET
  endwhile

  return OVER_SET
```

For each discrete encoding, ENCODING, in a set of encodings, ENCODE_SET, iteratively examine each BIT in the ENCODING, starting with the most significant bit. Retrieve a data element, DATA, from a data set, DATA_SET, on the first iteration and any subsequent iterations where the current DATA has been completely encoded. If the BIT value is 1, set a counter to one less than the degree, DEG, of the ENCODING. On subsequent iterations, if this counter is greater than zero, set the value of the current BIT in the ENCODING to one of the bits in DATA, starting with the most significant bit and continuing in order of decreasing significance in later iterations until the DATA has been completely encoded to the prior ENCODING. Decrement the counter after each bit written. Once every BIT in ENCODE has been examined, append ENCODE to a list of overwritten encodings, OVER_SET, and begin the process again with the next element in ENCODE_SET until all elements have been processed from either ENCODE_SET or DATA_SET.

3.1.5 Extracting Algorithm

```
Extract(OVER_SET, DEG):
  DATA_SET := {}

  while OVER_SET is not empty
    OVER := next element in OVER_SET
    READ_BITS := 0
    DATA_IDX := 0
    DATA := 0

    for each BIT in OVER
      if READ_BITS > 0
        if BIT == 1
          set bit in DATA at DATA_IDX to 1
        endif

        decrement READ_BITS
        increment DATA_IDX, wrapping to 0 on the last bit
        if DATA_IDX == 0
          append DATA to DATA_SET
          clear value of DATA
        endif
      else if BIT == 1
        READ_BITS := DEG - 1
      endif
    endfor
  endwhile

  return DATA_SET
```

For each discrete encoding, OVER, in a set of encodings, OVER_SET, iteratively examine each BIT in OVER, starting with the most significant bit. If the BIT value is equal to 1, set a counter to DEG-1. On subsequent iterations, if this counter is greater than 0, copy the value of BIT to a decode value DATA, starting with the most significant bit, and decrement the counter. As long as the counter is greater than 0, the value of bit is ignored for the purposes of setting this counter to avoid misinterpreting data bits as separator bits in the secondary encoding. Once a DATA element has been filled with bits extracted from OVER, append it to a list of data elements, DATA_SET. Once the end of an encoding, OVER, has been reached, retrieve the next encoding from OVER_SET and continue extracting until all elements have been processed.

3.2 Encoding Unit Size and Performance Analysis

In order to process large data sets, rather than trying to encode the data "all at once," each algorithm only operates on a portion of the total data set at one time, completing and storing the result of that operation before moving on to the next. This is a common strategy for dealing with large data sets in other computing domains, but is of particular consequence in its application to encoding data, as the exact size of the data chunks used can impact the compression performance of the encodings. When a large data set or file is encoded using one of these algorithms, the complete encoding of the data set will in fact be a composition of many smaller, equal-sized individual encodings. This size has to be agreed upon by each of the algorithms - as data that has been encoded according to a certain size can only be decoded by that same size - and is implementation specific. We describe this parameter as the **encoding unit size** of the implementation; this unit is specifically defined as the standard encoding size of the individual data values given to the primary encoder. Whatever this unit of data is, its size will determine the size of the primary encodings produced by the implementation, which in turn dictates the size of the input and output for each other function.

The specific size of the encoding unit is an implementation detail driven in part by hardware constraints: because the 'encode' algorithm works by interpreting and performing arithmetic on data as a single integral value, calculating a primary encoding from even a modestly large file would quickly become computationally intractable without some delineation of the input data due to the amount of memory and the number of calculations needed to represent and perform arithmetic on a scalar representation of the entire data-set. By reducing this single calculation into many smaller, more manageable operations and concatenating the results, memory and processing requirements are reduced dramatically, with only a negligible penalty to overall compression performance when compared to the ideal case. An encoding unit of 32 bits (4 bytes) is used in both of our implementations by default and is assumed for all practical analyses unless otherwise stated, with the rationale for this decision (and potential alternatives) discussed below.

3.2.1 Data Representation and Coding Performance

One of the most immediate reasons for choosing a 32-bit encoding unit is simply because of its ubiquity: as of the publication of this paper, the most common (although by no means only) size for an integer data type in fixed-width typing systems is 32 bits. Still, most modern general-purpose computing platforms have now adopted 64-bit architectures, and so these systems could also easily accommodate an encoding unit size of 64 bits or 8 bytes; their **word size**, the fundamental data unit with which a processor is designed to operate, is large enough to accommodate such calculations

natively. Selecting a larger encoding unit than the arithmetic data width supported by the target computing platform would be inadvisable, as the computational overhead associated with so-called bignum or arbitrary-precision arithmetic can be significant, but an encoding unit size *smaller* than the maximum operational width of the architecture has no significant impact on computational performance, assuming that all arithmetic operations (i.e. comparison, addition, subtraction) take the same amount of processor time to complete regardless of the width of the operation. The reason for this is that, whether it is encoding two 32-bit data values or one 64-bit value, the processor will still end up performing roughly the same number of calculations to arrive at the result(s): an encoding of degree 2 requires 46 series elements, or 46 bits, to represent all the values of a 32-bit number, while a 64-bit number occupies exactly twice that amount - 92 bits - once encoded. In both cases, these two different encoding units are expanded by the same constant factor, and each requires the same number of calculations per encoding bit produced (one comparison and one addition or subtraction).

3.2.2 Encoding Unit Compression Performance

Different size inputs to the primary encoding algorithm can result in different numbers of "round-off" bits, though, when comparing the theoretical and practical minimum and maximum compression ratios for the resulting encoded data. Again, consider our 32-bit encoding unit encoded using a series of degree 2 for a resulting encoding of 46 bits; as we demonstrated in Chapter 1, the theoretical best-case secondary storage capacity for an n-bit encoding is $\frac{n}{2}$, and is $\frac{n}{3}$ in the worst case. For our example of a 46-bit encoding, $\frac{46}{2} = 23$ happens to be an optimal maximum number of possible secondary storage bits, but $\frac{46}{3} = 15.333$ is a sub-optimal minimum for this encoding resolution due to the fact that fractional bits do not exist in reality, and so we "lose" one-third of a bit in our worst-case secondary storage outcome. Another way to look at this is that, if our goal is only to guarantee that we have *at least* 15 bits of secondary storage, we could just as easily do so with a 45-bit encoding ($\frac{46}{3} = 15$); in the worst case, our 46-bit encoding occupies an additional bit in memory while only providing the same number of secondary storage bits. The overall effect of this is that our worst-case compression ratio for this encoding unit size is now $\frac{32+15}{46} = 1.022$, slightly lower than our theoretical worst case ratio of 1.028.

The magnitude of this error varies with encoding unit size and encoding degree in these best/worst case compression analyses, but even though one encoding unit size may be well-suited for one particular encoding degree, it may perform poorly with others; moreover, using odd-numbered or non-byte-aligned encoding units introduces additional complexity when applied to byte aligned systems. In general, longer encodings can offer modest benefits to average compression ratios simply due to the fact that this fractional loss represents a lower overall proportion of the number of bits stored. Mathematically, we can describe the lower bound of the ratio of actual to theoretical compression performance as $\frac{n+r_a}{en} \div \frac{n+r_t}{en} \approx \frac{n+r_t-1}{n+r_t} = 1 - \frac{1}{n+r_t}$, where n is the encoding unit size, e is the expansion factor on the encoded data for a given encoding series, and r_a and r_t are the actual and theoretical round-two storage capacities of the resulting encoded word (maximum or minimum), respectively; we simplify the relationship between r_a and r_t as $r_a = r_t - 1$ to calculate a lower bound on compression loss because the actual number of secondary storage bits will never vary from the theoretical by more than 1 bit.

From this equation we see that, as n increases (and r increases as a function thereof), $\frac{1}{n+r_t}$ becomes vanishingly small and so the lower bound on the ratio of actual to theoretical performance approaches 1. In practice, this effect is marginal, as an encoding unit size of 32 bits already returns

a lower bound loss of worst-case performance of $\frac{1}{32+15.333} = 0.979$ for degree 2 encodings, and in practice this is even lower. If we consider the next power-of-two data size: a 64-bit encoding unit, expanded to 92-bits once encoded, gives us a worst case compression performance of $\lfloor \frac{92}{3} \rfloor = 30$, which is in practice no better than the performance of our 32-bit encodings. We would have to choose the next power of two, 128, in order to obtain a more favorable outcome, with encodings of 184 bits yielding a worst-case round two storage capacity of $\lfloor \frac{184}{3} \rfloor = 61$, or one extra bit of storage for every 16 bytes in the worst case.

3.2.3 Storage Performance

Before delving into any particular method of producing encodings, it would be good to have some idea of the theoretical performance benefit of the coding method itself in the ideal case. [10] provides a theoretical framework for describing flash performance, whereby the *cost* of each read, in terms of amount of time necessary to perform the operation, is calculated by $C_{read} = xT_r$, where T_r is the time it takes to read a page and x is the number of pages read. Similarly, the cost of writing can be summarized as $C_{write} = p_i y T_w + p_o(xT_r + yT_w) + p_e(T_e + yT_w + T_c)$, where T_r is again the time it takes to read a page, x is the number of pages which need to be read for the operation (as in reading a page to modify its value), T_w is the time it takes to write a page, y is the number of pages written to, T_e is the time it takes to erase a block (assuming only one erase needed per write), and T_c is the time needed to copy the current data from a block being erased, which will be proportional to $n(T_r + T_w)$ for the number of pages, n, that contain valid data, bound by $0 \leq n \leq b - 1$ for a block containing b pages. The values p_i, p_o, and p_e then represent the probabilities that a write will be either performed in-place, that it will require some x reads before writing (such as during an unaligned page access), or that it will require a full block-erase cycle, respectively.

As the encoding process expands the data being stored, effectively "watering down" the bytes being written to the storage device, bandwidth will suffer accordingly; assuming, for example, an SSD write speed of 500MiB/s for standard encodings, the effective bit rate for primary encodings of degree 2, which occupy roughly 44% more memory, will be reduced to $\frac{500*32}{46} = 347.8$MiB/s in the ideal case, since each encoded bit takes the same amount of time to read and write, but represents proportionally fewer bits of the actual information being stored (if memory is *not* accessed efficiently due to the encoding process, a topic discussed in chapter4, this reduction could be even higher). In terms of the drive access cost calculations, this is modeled by applying an expansion factor, e, to T_r and T_w, as the time to read and write each unit of *information* is increased by whatever the average expansion of the data is for each operation across both rounds of writing. Adding this factor to our model, the cost calculations become:

$$C_{KSread} = xeT_r \tag{3.1}$$

$$\begin{aligned} C_{KSwrite} &= p_i y e T_w + p_o(xeT_r + yeT_w) + p_e(T_e + xeT_w + T_c) \\ &= e(p_i y T_w + p_o(xT_r + yT_w)) + p_e(T_e + xeT_w + T_c) \end{aligned} \tag{3.2}$$

Note that the only terms unaffected by this expansion are T_e and T_c: because they are both based on the time taken to operate on the fixed size of a flash block, they are unaffected by changes

in the amount of information represented by those operations.[2] Equation 3.2 thus demonstrates that, in order for the cost of writing to the SSD, C_{write}, to be lower using encoded data, p_e has to be reduced by some factor proportional to the additional cost of one of these read-erase-write cycles. Assuming some average cost for each "type" of write, C_i, C_o, and C_e, these values can be substituted for each probability term in 3.2, so that $C_{write} = C_i + C_o + C_e$ and $C_{KSwrite} = e(C_i + C_o) + rC_e$ (assuming no effect on C_i and C_o other than data expansion[3]), where r represents the reduction in C_e due to its probability of occurrence being reduced by the coding method. Rewritten as a series of inequalities:

$$\begin{aligned} C_{write} &> C_{KSwrite}, \\ C_i + C_o + C_e &> e(C_i + C_o) + rC_e, \\ 0 &> (e-1)(C_i + C_o) + (r-1)C_e, \\ (1-r)C_e &> (e-1)(C_i + C_o) \end{aligned} \tag{3.3}$$

In other words, the *reduction* in the weighted-average read-erase-write time, $(1-r)C_e$, must be greater than the corresponding *increase* in write and read-write times, $(e-1)(C_i + C_o)$ for encoded vs. un-encoded data in order for the encoding to be effective. This analysis will be applied again later when considering access times for particular encoding instances, using definitive values for e and r.

3.2.4 Parallelism

One final point to consider when choosing an encoding resolution is the potential for parallel processing of data. On an individual basis, each of these algorithms process data sequentially, examining and producing a decision on each bit of the encoding being created/ decoded before moving on to the next. However, separate encoding and decoding processes are almost entirely independent of each other, so this strategy of breaking the data into many individual encodings can work in our favor when parallelizing the encoding and decoding processes: the smaller each individual encoding is, the more of them can be processed in parallel. In particular, the primary encoding, decoding, and padding processes are all trivially easy to parallelize, as each instance of one of these processes is mutually independent of any other instance of the same process.

Parallelism for secondary file encoding is somewhat less straightforward, but still possible given some additional information about the set of primary encodings being overwritten. The 'overwrite' algorithm above works by opportunistically filling the bits of the input primary encodings, writing one bit of the input data stream for each 'data bit' found with no guarantees as to how much input data will be written from one encoding to the next. This makes it difficult to predict how the input data will align with the primary encodings that they are encoded to; ideally, we would like to supply the 'overwrite' algorithm with just enough data at a time to fill each individual primary encoding so that all of the available secondary storage bits are utilized and there is no overlap between

[2] Technically, it will probably be more likely that T_c is large (more pages per erased block require copying) due to the larger number of pages occupied with each write, but the SSD still only needs to copy a whole number of pages up to the size of the block, minus the pages being newly written, regardless of whether the data therein is encoded or not.

[3] Again, it will probably be the case that C_o becomes somewhat higher due to the increase in the number of operations which require a read as well as a write in order to overwrite the stored encodings.

the data stored in the resulting secondary encodings. However, without some foreknowledge as to the exact number of 'data bits' each primary encoding can accept, it is impossible to know which data bits to start writing into the next primary encoding without knowing the results of the previous encoding. The solution to this problem is to add a pre-processing step to the input primary encodings when overwriting them: in parallel, count the number of sentinel bits in each primary encoding to predetermine the number of data bits that each encoding will store, then align the data to be encoded as necessary. Extracting and re-aligning the encoded data can be done in parallel without the use of a pre-processing phase by simply reading out each encoded bit string and concatenating the results.

The potential benefit to choosing a relatively small encoding unit size can thus be quite substantial, as a reduction of the encoding unit size by half can theoretically double the processing speed of a given data set. Of course, the degree to which a smaller encoding unit is useful is limited by factors such as the potential space and compression inefficiency of extremely small encodings, and hardware resource availability is a limiting factor in any parallel computing problem. We predict that parallelism will be an important consideration for any application of this coding method due to the need to minimize encoding/ decoding overhead in order to extract any meaningful performance benefits from this method of data storage; our implementations themselves do not directly provide support parallel processing of data, but the hardware circuit design in particular was created with this objective in mind, and is modularized to support hardware repetition for parallel applications.

3.3 Software Implementation

The initial implementation of this code was written in C primarily as a way of prototyping the logic for the encoding and decoding processes, as well as to test and collect data on average file compression characteristics. Source code was compiled using the GCC compiler and tested on a distribution of the Linux operating system. The program accepts either four or five parameters, depending on the mode of operation: first, the operation or "sub-command" to be performed; second, the encoding series degree; third, a path to a file to obtain input data from; fourth, a path to write output data to; and last, a second input file path, used when creating a secondary encoding to specify the file to be encoded (the first input file in this case is used to obtain the previously encoded data that the second file's data will be written into). Given a valid set of arguments, the program will attempt to open the given input and output file(s) and, if successful, will begin reading and processing the provided file data according to the operation specified, writing the results to the specified output file. If the output file path supplied to the program references a file that already exists, its contents will be replaced with the results of the current operation, otherwise a new file is created at that path (assuming it is valid).

There are 5 supported operations, corresponding to the five coding processes identified in the last section - encode, decode, overwrite, extract, and pad - which, again, implement the primary encoding and decoding processes, the secondary encoding and decoding processes, and the data padding scheme, respectively. All operations require an output file and at least one valid input file to be provided, as well as an encoding degree; the degree parameter is needed, not just in primary encoding and decoding but in all operations, due to the relationship that the encoding degree has with the number of bits that can be read/ written/ padded after each separator bit during the processing of these respective tasks. Valid encoding degrees are in the range of 2 to 10, inclusive; this constraint is partially to limit the size of the output data and the calculated encoding series, as well as to simplify certain operations by allowing assumptions to be made about this parameter.

This is considered a reasonable limitation, as our analysis only focuses on degree values in this range, with higher degree values being of questionable utility due to their size and poor compression performance characteristics.

All encoded output files produced by this program (i.e. as a result of encoding, padding, or overwriting) are formatted with an 8 byte header prior to receiving encoded data in order to both identify these files as having been produced by this particular coding implementation and to store file meta-data, including the state of the encoding (primary, padded, or secondary) and the number of secondary data bits it contains. Any encoded files which are then provided as inputs for other operations can be evaluated and rejected if they do not contain the proper formatting. This, in part, protects the user from accidentally supplying the wrong type of encoding for each operation and helps to ensure the correctness of the results.

3.3.1 Software Implementation Notes

One important caveat when generating secondary encodings: depending on the relative sizes of the two input files and the number of storage bits available in the encodings of the primary input file, and unless the size of the secondary input file matches *exactly* with the number of data bits in the primary file, the algorithm may reach the end of either input file before completely processing the other. In a situation where the secondary input file contains fewer bits than the number of data bits present in the primary input file, the program can simply write out the final output byte (assuming there was a byte encoding in progress before the end-of-file was reached) and terminate early. This scenario is preferred, as the ultimate goal of the secondary encoding process in this case is to simply represent the data provided by the secondary input file in the output encoding, which is accomplished whether or not the primary input's data bits have been fully utilized. A more difficult issue arises when the end of the *primary* file is reached first, leaving a portion of the secondary input file data absent in the output file; there are more robust approaches to handling this situation, but the policy of this implementation is to simply report a warning to the terminal and exit early, leaving the rest of the secondary file data bits absent in the output encoding. While this solution is acceptable for our software implementation, in which we are simply interested in testing our encoding/ decoding algorithms, but is obviously untenable for use in an actual file I/O system due to the potential loss of file data. This problem, and possible solutions thereto, will be revisited in Chapter 4.

Also note that the contents of the provided input file(s) are never themselves altered: the program only ever *writes* manipulated input data (if applicable) to the specified output file, a useful design choice during prototyping as it both prevents loss of data due to an irreversible misapplication of some coding process and enables validation of complete process cycle results by preserving the original input to the encoder, to which the decoded result can be compared.

3.3.2 Average File Compression Ratios

After completing the development of this application, we ran the application on a set of sample files of various types in order to obtain compression performance averages for different kinds of data using our particular implementation. Table 3.1 lists the mean compression ratio achieved across a small selection of five representative files of each type for varying encoding degree. The

first row shows results for compression performance testing done on a series of text files containing pseudo-random contents generated by a standard system PRNG. This set of results can in some ways be considered the "baseline" result, as it shows how the encoding is likely to perform assuming uniform distribution of data values. Indeed, this is essentially the observed result: almost none of the performance averages differ to a statistically significant degree. Moreover, the variance between each file for each degree of encoding tested was observed to be negligibly low in each case (typically $<< 10^{-3}$).

Table 3.1
Average Write Compression for Varying File Type

File Type/ Degree	2	3	4	5	6	7	8	9	10
.txt	1.15	1.14	1.11	1.08	1.06	1.04	1.03	1.01	1
.pdf	1.15	1.14	1.11	1.09	1.07	1.05	1.04	1.03	1.02
.mp3	1.15	1.14	1.12	1.1	1.08	1.06	1.05	1.04	1.03
.jpg	1.15	1.13	1.11	1.09	1.07	1.05	1.04	1.03	1.02
.docx	1.15	1.2	1.12	1.1	1.08	1.06	1.05	1.04	1.03

Table 3.2
Write Compression Sample Variance

File Type/ Degree	2	3	4	5	6	7	8	9	10
.txt	0	0	0	0	0	0	0	0	0
.pdf	$3 \cdot 10^{-7}$	$6.8 \cdot 10^{-6}$	$6.7 \cdot 10^{-6}$	$1.97 \cdot 10^{-5}$	$8.7 \cdot 10^{-6}$	$3.68 \cdot 10^{-5}$	$6.03 \cdot 10^{-5}$	$4.3 \cdot 10^{-6}$	$4.7 \cdot 10^{-6}$
.mp3	$3.5 \cdot 10^{-6}$	$2.13 \cdot 10^{-5}$	$1.78 \cdot 10^{-5}$	$7.33 \cdot 10^{-5}$	$7.27 \cdot 10^{-5}$	$8.35 \cdot 10^{-5}$	$6.52 \cdot 10^{-5}$	$1.03 \cdot 10^{-4}$	$7.27 \cdot 10^{-5}$
.jpg	$2 \cdot 10^{-7}$	$3 \cdot 10^{-7}$	$2 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$2 \cdot 10^{-7}$	$6.3 \cdot 10^{-6}$	$7 \cdot 10^{-7}$	$1.3 \cdot 10^{-6}$	$1.8 \cdot 10^{-6}$
.docx	$2 \cdot 10^{-6}$	$1.6 \cdot 10^{-2}$	$1.02 \cdot 10^{-5}$	$1.33 \cdot 10^{-5}$	$1.32 \cdot 10^{-5}$	$1.85 \cdot 10^{-5}$	$1.72 \cdot 10^{-5}$	$2.32 \cdot 10^{-5}$	$1.77 \cdot 10^{-5}$

While the sample size that this data was drawn from is admittedly very small, the unerring consistency of the results should provide at least a well educated guess as to the compression performance that can be expected in a typical application, barring any unusual properties of the data. It would appear, for instance, that higher degree encodings (up to the maximum *theoretical* compression ratio in degree 4) do not, on average, offer any performance benefit inherently. Other factors may still affect the decision to use one encoding degree or another, and compression performance is still similar for the first few degrees, so any of these could potentially be a viable option.

3.3.3 Average Compression Performance Analysis

Using these average-case compression numbers, some additional analysis can be applied to the theoretical disk performance effects of using the KS-code. Returning to 3.3, the average cost increase for reading and writing can be calculated by taking a simple average of the data expansion in both rounds of writing (assuming equal probability of accessing a primary or secondary encoding). For an encoding degree of 2, with an average compression ratio of 1.15, two optimally efficient accesses to n bits of memory will affect $\frac{n}{1.44}$ bits of data in the first write and a further $1.15n - \frac{n}{1.44} = 0.456n$ bits in the second, but the drive operation on n bits still takes the same amount of time regardless of whether data is encoded and only stores or returns a fraction of the information bits in the encoded case. Accessing n bits of memory over two rounds of coding thus allows $1.15n$ total bits of information to be read or written, yielding an average storage bandwidth utilization of just $\frac{1.15n}{2n} = 0.575$, and so the cost increase of the access will be the reciprocal of this, $e = \frac{1}{0.575} = 1.74$.

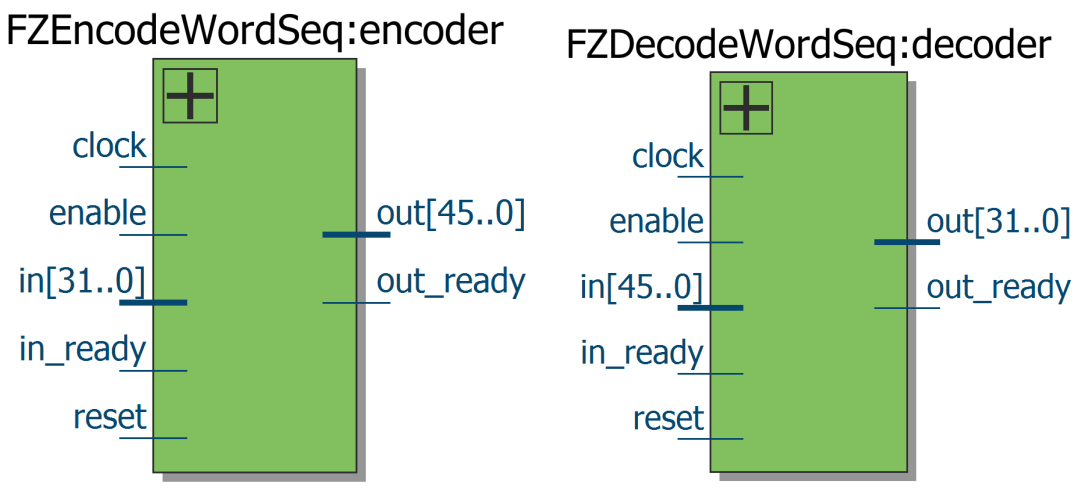
Taking this analysis a step further, it is possible to provide some reasonable estimate as to the relative costs of T_r, T_w, and T_e. For a representative example of the performance of a flash memory module, [16] lists the read, write, and erase times for an 8GiB flash chip which differ by about one order of magnitude each when compared to the next fastest operation, with reads being the fastest, taking at most $25\mu s$, and program/ erase operations typically requiring approximately $220\mu s$ and $1.5ms$, respectively. Substituting values of $T_r = 1$, $T_w = 10$, and $T_e = 100$ then results in $C_{write} = p_i 10 + p_o(1 + 10) + p_e(100 + 10 + T_c)$, assuming an access of a single page. T_c can be estimated by $\frac{b}{2}(1 + 10)$, where b is again the number of pages per block, based on the assumption that a block copy will require, on average, half of the block pages to be copied and rewritten after an erase. For a block size of 64, this puts the value of T_c at roughly 352. Finally, reducing everything so far and applying the inequality in equation 3.3, the overall performance effect of the encoding is approximated by $(1 - r)(p_e 462) > 0.74(p_i 10 + p_o 11)$. It is difficult to make any further conclusions about the efficacy of this method of encoding without additional information on the frequency of each type of write for a given SSD, but if r is estimated at $\frac{1}{1.15} = 0.87$, given that each write should theoretically be compressed by 15% according to the average compression ratio and assuming that this results in a linear reduction in the probability of a block erase occurring, the inequality $0.13(p_e 462) > 0.74(p_i 10 + p_o 11)$ is obtained. To further simplify, we assume $p_i \approx p_o$, since every secondary write (half of all writes, in theory) will be of this read-write type. Substituting p_i and p_o with $(1 - p_e)/2$ gives a final performance inequality of:

$$\begin{aligned} 0.13(p_e 462) &> 0.74((1 - p_e)(10 + 11)), \\ 0.13(p_e 462) &> 15.54 - p_e 15.54, \\ 0.13(p_e 462) + p_e 15.54 &> 15.54, \\ p_e 60 + p_e 15.54 &> 15.54, \\ p_e &> 0.206 \end{aligned} \tag{3.4}$$

Inequality 3.4 would suggest that over 20% of writes would need to require a block erase before the benefits of using KS-coding outweigh the costs, which is relatively high for a drive with ample space to accommodate writing, but may improve access speeds for drives with relatively high utilization. It could also potentially be the case that p_e does not vary linearly with the compression ratio; if a relatively small reduction in the number of memory bytes used for both rounds of writing reduced the occurrence of a block erase to a much greater degree, then the encoding would be much more effective.

3.4 Hardware Implementation

We developed our hardware implementation as a series of SystemVerilog HDL modules, defining a set of logic circuits to perform each of the various data transformation activities required to implement the KS-code, as well as an interface circuit to incorporate all of these functions into a single package with a common I/O interface. The HDL code provided herein can thus be used to synthesize a functional hardware circuit - what is known as an **IP core** - to provide the specific data manipulation capabilities of this rewriting code within a larger device-level design. As this implementation is not a complete I/O device but is instead designed to be incorporated into an existing circuit that can provide data and control logic to the module, this section focuses on the operation of the



(a) Encoder (b) Decoder

Figure 3.1: Block diagrams of the encode and decode modules

implementing module and its sub-components from the perspective of this hypothetical "control device."

The hardware supports the same 5 basic functions as does the software (encode, decode, pad, overwrite, and extract), with each function being implemented as a modular sub-circuit within the overall design. Common inputs to each sub-circuit include clock, enable, and asynchronous reset pins. Each sub-circuit also features an output bus and one or more input buses which carry data bits to and from the sub-circuits in parallel, as well as bus status pins which allow I/O synchronization between the sub-circuit and the device. Data buses are sized to match either the configured encoding unit size of the module (32 bits by default) or the corresponding encoding output size for the given encoding unit and encoding degree, and all circuit I/O generally conforms to one of these two data sizes; when describing module or sub-module functionality, we will refer to the data on these bus types respectively as either **data words** or **encoded words**, with the word size of the module defined as the former of the two. This section describes the operation and I/O characteristics of each functional module, as well as their integration into the top-level packaging module, before providing a theoretical performance analysis and discussion of device-level integration considerations.

3.4.1 Primary Encodings and Padding

The encode, decode, and pad circuits all exhibit essentially the same I/O pattern, save for the size and exact nature of their input and output data, and are in fact described by the same finite state machine, shown in 3.3. From an initial (i.e. reset) state, the control device can initiate a module task by asserting a value to its input bus and pulsing the input `in_ready` pin high for one clock cycle; during this cycle, the module latches in the input bus value and stores it internally, leaving the device free to alter the bus value without disrupting the module on subsequent cycles. The module completes its task a set number of cycles later, asserting its output `out_ready` pin high on the same cycle that it performs the last calculation for the current task to signal to the device that the output

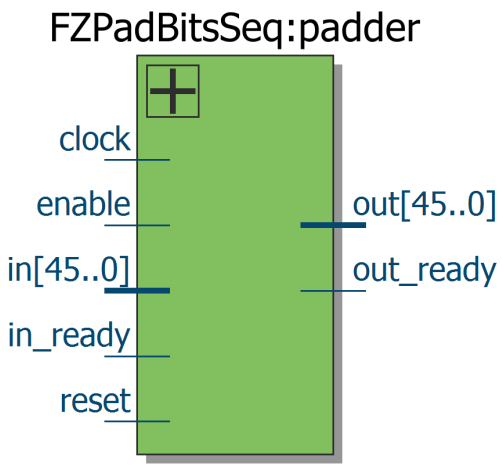


Figure 3.2: A block diagram of the pad module

bus value is now valid. The control circuit detects this signal one cycle later and can then examine the output value and provide a new input to the module if another operation of the same type is required.

Note that, once an operation has started, each module's policy is to *ignore* any further activity on the input bus or `in_ready` pin to avoid disruptions to the current operation. This design prevents data errors by ensuring that the module performs each task in its entirety before it can signal a valid output word, and also introduces the possibility for additional module-level concurrency when packaging the functional circuits together, as 3.2.3 discusses. As mentioned above, each of these processes require a fixed number of cycles to produce a result: the total time each module takes, in clock cycles, from latching input to signaling output-ready is equal to one greater than the size of an encoded word; mathematically, this value is approximated by $1 + e * n$, where e is the expansion factor of the chosen encoding degree and n is the width of the data on the input bus.

3.4.2 Secondary Encodings

In contrast to the previous three circuits, the overwrite and extract implementations both require some form of asynchronous input (i.e. input not concurrent with the completion of a task), making their operation somewhat more complex. The overwrite circuit, tasked with writing new data words into existing encoded words, must manage two inputs coming from two separate I/O streams; this raises a related issue to the one described in 3.2.2: due to the size disparity between the primary and secondary input data words and the uncertainty in the number of data bits available in each primary encoding, it is possible that the module may exhaust the input data word before completely processing the encoded word, or vice versa. 3.5 demonstrates the finite state machine model of this circuit, with an additional state corresponding to the case wherein the module has received an encoded word as input but has not yet received a data word.

Starting a new module task follows the same process as in the prior circuits, but with a second input bus and input valid signal: from the initial circuit state, the control device provides input(s) to

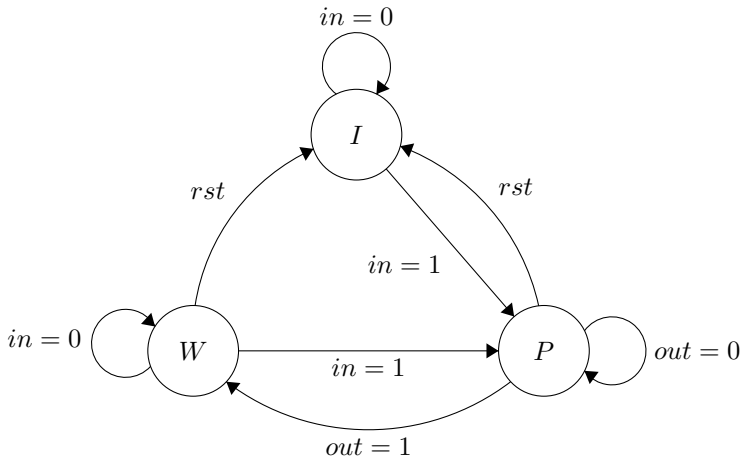
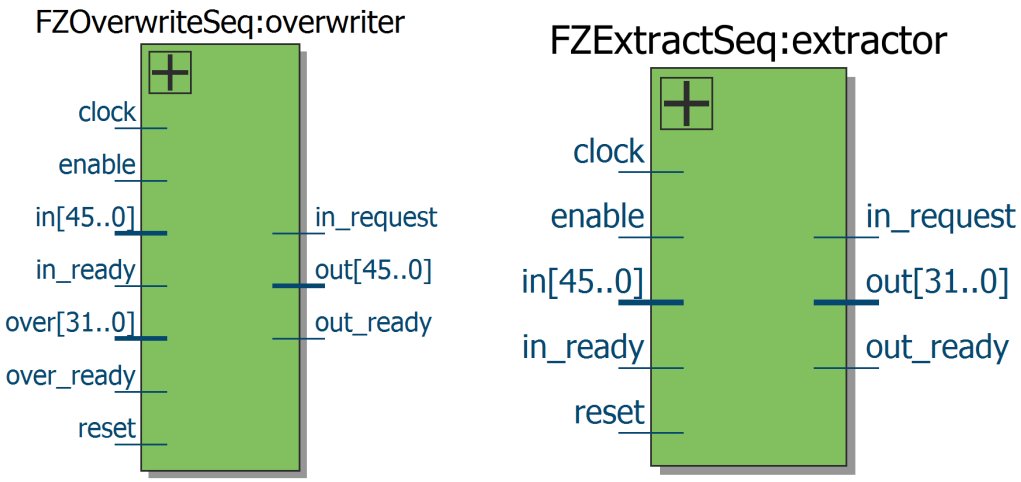


Figure 3.3: State machine for encode, decode, and pad modules. From initial state I, module transitions to P to process input once it becomes valid. From P, transitions to W once output is complete and waits there for additional input. Resetting returns the module to I.



(a) Overwriter

(b) Extractor

Figure 3.4: Block diagrams of the overwrite and extract modules

the circuit, signaling that both inputs are ready by asserting both the `in_ready` and `over_ready` pins high for at least one cycle. Once the inputs are latched in next cycle, the circuit begins performing its overwriting task and the input bus values can be safely modified. On the final cycle of the task ($1 + e * n$ cycles after being initiated), the circuit signals that its output is ready and will hold this signal and the output bus value until the control device initiates a new encoding by providing another encoded word to fill.

After this initial overwrite task, the operation of the circuit changes slightly in order to preserve data word bits left unwritten in the first task. In most of the implemented modules, signaling that

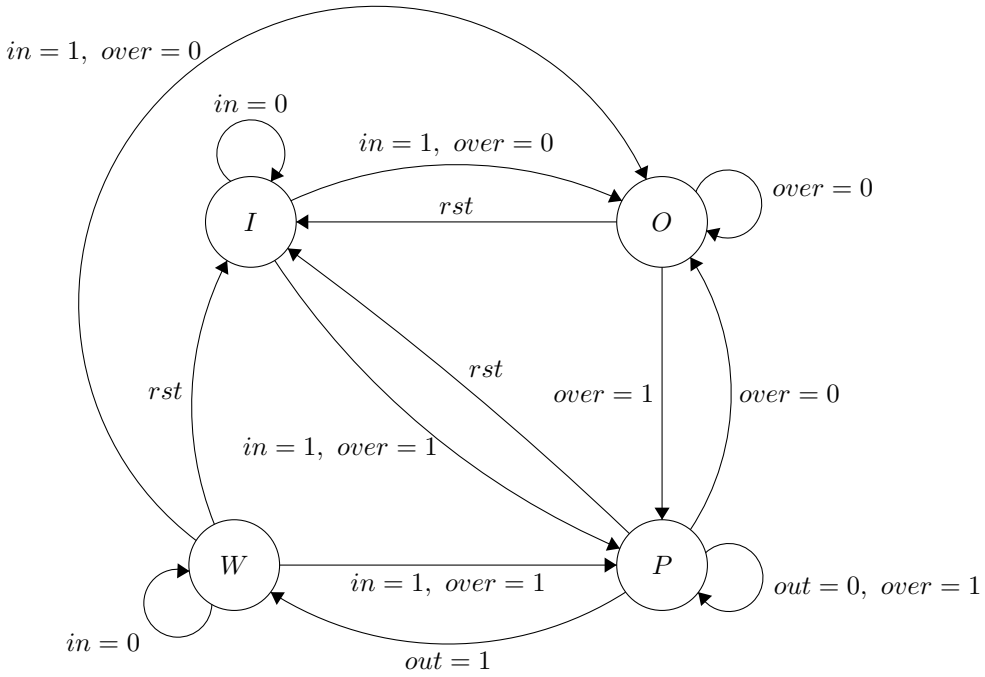


Figure 3.5: State machine for the overwrite module. From initial state I, module transitions either to O while awaiting secondary input, if data to overwrite has not been provided, or to P to process the encoding. From P, transitions are either to W when an encoding is complete or to O if data is exhausted beforehand. Resetting returns the module to I.

an output word is complete also implies that an input word has been fully processed and can be replaced with a new value, so it is not necessary to provide ports to signal that additional data is needed for these inputs; the overwrite module itself exhibits this behavior as well, using the output ready signal as a cue to the controller that it requires an additional encoded word input. In the case of the overwrite module's data word input, however, because of the variation in encoded word secondary storage capacity, it is unlikely that the module will finish writing every bit in the data word at the same time as it completes writing to an encoded word. In fact, the module may finish writing its current data word on any cycle of a given task, and so it utilizes a secondary output signal, `in_req`, in order to notify the control device that it has exhausted its data word input and request additional data from the device. The controller can then respond to this signal by either providing another data word to the circuit or just reading out the current, partially processed contents of the output buffer (as this signal could mean that the last data word in a file or data set has just been processed, and that there are no more to follow).

Finally, the operation of the extract module also comes with some exceptions as compared to the previous modules, again due to the variation in the number of data bits present in each encoded word. Each module that this section enumerates is designed to wait for some input to begin a task, then to continue that task until it produces a complete output, such that the result of the operation fills a full word of the appropriate type for that module; for the extract module, however, this means producing a fixed number of output bits from a variable number of data bits in each input, processing as many input encoded words as is necessary to produce a full data word. The extract module again uses a data exhaustion/ data request signal to explicitly notify the controller

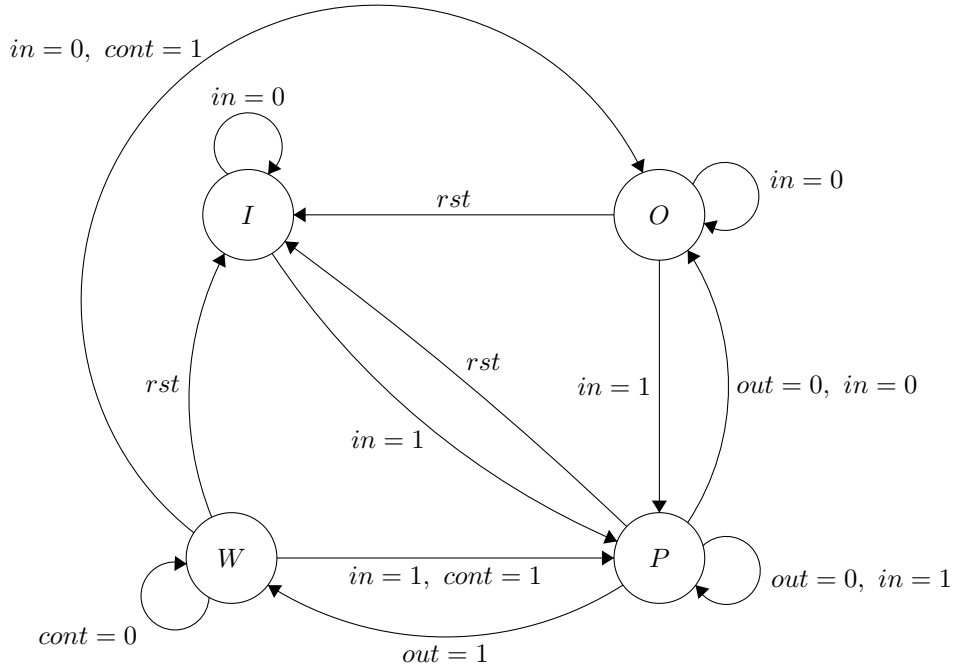


Figure 3.6: State machine for extract module. From initial state I, module transitions to P when input data is available for processing. From P, transitions are either to W when a decoding is complete or to O if input is exhausted beforehand. Resetting returns the module to I.

every time it consumes an input encoded word, making this the only module with no synchronicity between any of its inputs or outputs.

The finite state machine for the extract module is shown in 3.6. Once the circuit is in its initial state, the control device again supplies input to the module data bus and signals that the input is ready. After latching in the input encoded word, the extract circuit continues processing until it either completes a full data word or runs out of encoding bits to examine. If the extract module indicates that it has run out of encoded data to process, the controller can again either provide additional data and allow the module to continue its task, or simply read the partial output data. Once a full output word is complete, the module again halts and waits for the control device to signal that it should continue via the `in_ready` pin; in this case, if the module still contains unprocessed input data (which is likely to be the case), input is *not* latched in from the input bus, and the module continues extracting its current data word. The extract module is still guaranteed to completely process its *input* in $1+e*n$ cycles, but may require anywhere in the range of $1+\frac{n}{\frac{r}{e*n}},\ r_min \leq r \leq r_max$ cycles to calculate a complete output, where r is the average number of data bits per encoded word and is bounded by r_min and r_max, making this the only hardware task not guaranteed to complete within a uniform time period.

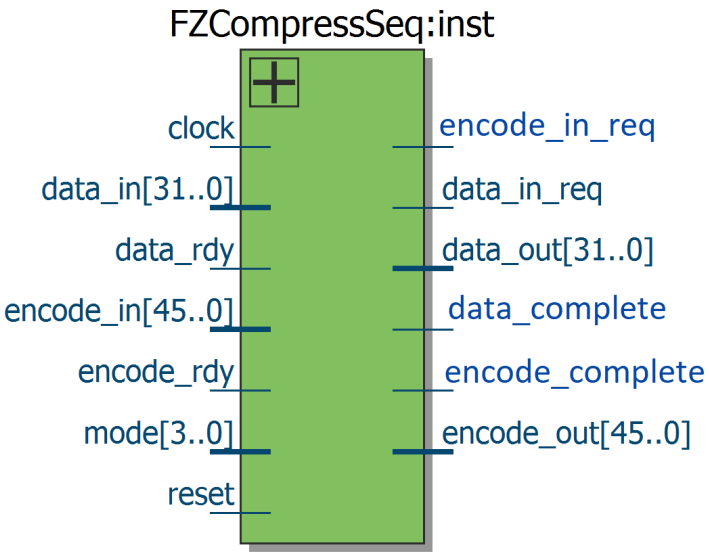


Figure 3.7: Block diagram for the packaging module

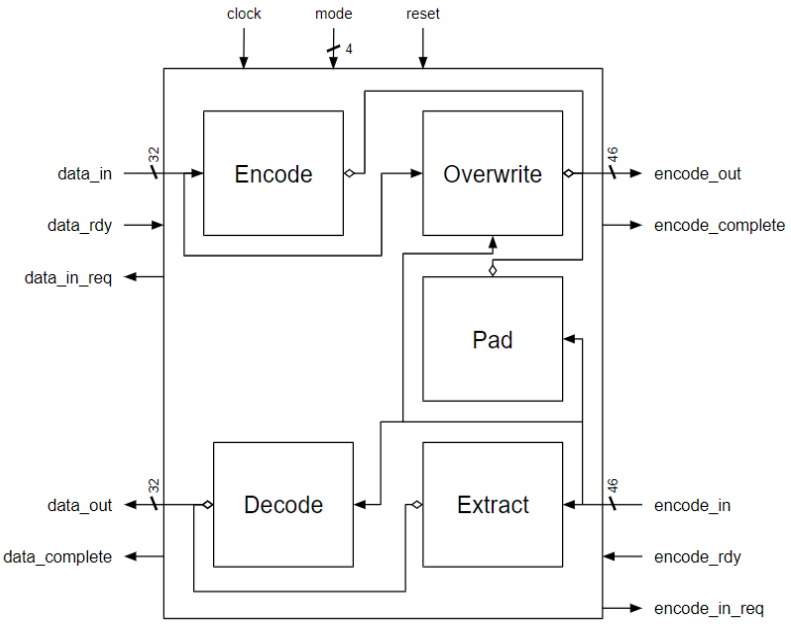


Figure 3.8: I/O flow diagram for each module within the packaging module

3.4.3 Packaging Module

The purpose of the packaging module, the top-level module in this implementation, is primarily to provide a common interface for each of the aforementioned sub-modules, with very little logic actually

present in the packaging module itself. Inputs to the module include clock, reset, and mode select pins, as well as two input buses: one to accept data words to be encoded, and one to accept encoded words to be decoded or padded. A complimentary pair of output buses provide a complete data pathway through the circuit, again carrying both data words and encoded words out of the module on their respective ports. Two input pins, one for each input bus, carry input readiness signals to the various sub-modules, and another pair of output pins likewise provide exterior visibility of output readiness signals for each output bus. Finally, two additional output pins are used to signal asynchronous input requests for each input bus.

The 4-pin mode select port allows for up to 16 different operating modes, only 11 of which are currently implemented. Five of these modes are, somewhat predictably, for enabling each sub-module individually, plus one for a no-op mode (all modules disabled). It is also possible to enable and use multiple modules at once, provided only one module is using any bus at a given time. This means that the packaging circuit can allow up to two modules to be active at the same time, one for each output bus. If two modules' input buses are also mutually exclusive (for example, the encode and decode sub-modules), both can be enabled for simultaneous, continuous operation. Not all modules can operated continuously in these paired-processing modes due to the fact that some of these modules may still read from the same input bus(es) and thus may obtain the same input value during the course of their operation, which is usually not the desired behavior. However, due to input latching, even modules that share an input bus can be allowed to process concurrently as long as each is enabled and initiated on a different clock cycle.

Besides decoding the chip select signal and MUX-ing the appropriate output pins to the currently enabled module(s), the last function that the packaging module itself provides is arbitration of potential bus conflicts: if both active modules in the current mode depend on the same input bus and both have raised signals signifying that they are out of data, the packaging module will temporarily disable one of these modules while the other receives data. The module to be disabled will always be the same for a given mode so that the control device knows which module to provide data to first (although, this order can also be reversed manually by changing the current mode to enable the other module). This again helps to ensure the safety of the data and correctness of the circuit. Table 3.3 details each of the 12 implemented modes, with an 'x' in a cell denoting that the corresponding module is active in that mode. Additionally, asterisk in the cell denotes that that module receives input bus priority during arbitration; the interfacing circuit will need to be aware of this priority in order to determine which module it should supply data to during these instances.

3.4.4 Hardware Synthesis Results and Performance

This section discusses the hardware logic requirements and theoretical performance of the synthesized SystemVerilog circuit design. 3.4 shows the synthesis results for the top-level module, including the logic contained in each sub-module. Overall, these logic requirements are very modest, requiring only a few thousand hardware elements all together to produce the whole circuit. This augurs well for potential parallelization of the module, as these synthesis results suggest a fairly compact and inexpensive circuit once implemented in hardware.

Every hardware module except for the extract module produces its result in a constant number of clock cycles, which makes reasoning about their performance fairly easy. Based on circuit analysis by the TimeQuest timing analyzer, the top-level module can support clock a clock speed of just

Table 3.3
Top-Level Circuit Modes

Mode	Encode	Decode	Pad	Overwrite	Extract
0					
1					x
2				x	
3			x		
4		x			
5	x				
6				x	x
7	x				x
8			x	x*	
9	x		x*		
10		x*		x	
11	x*	x			

over 123MHz, or approximately 8.13ns per clock cycle. For a 32-bit encoding unit size and an encoding degree of 2, the resulting 46-bit encodings would each take approximately $(2+46)*8.13 = 390.24ns$ to process at that speed, resulting in a data throughput of $\frac{32}{390*10^{-9}} = 78.25Mib/s = 9.78MiB/s$ in un-encoded data bytes, or approximately $14.06KiB/s$ of encoded data. This is quite slow compared to typical SSD write speeds, which can exceed $500MiB/s$, and would represent a significant performance bottleneck if the encodings were performed in sequence. Useful amounts of data throughput can still be achieved, however, by using these modules in parallel to process many encoded words at a time; even just 64 of these coding modules in parallel would be enough to process incoming data at roughly $626MiB/s$.

Table 3.4
HDL Synthesis Results

Component Type	Number Required
Total logic elements	807
Combinational functions	755
Total registers	292
Total pins	168

4. System Integration Considerations

Both implementations of the KS-code described are "complete" in that they are, when provided with the appropriate data and instructions, capable of correctly applying the coding processes that constitute this rewriting code. However, as outlined by the project scope in section 1.1, these data transformations are a necessary but *insufficient* condition to the application of write compression to solid state devices; this method of data storage raises a number of potential issues, both for the performance and capacity of the storage device as well as the accessibility of the data being stored, that must be addressed before such a code can be put into practice. Details such as data addressing, physical memory structure, and determining which operation to apply to the data all need to be considered in order to correctly store and retrieve these encodings, and the design choices made in doing so will, in part, determine how the system performs overall.

This chapter briefly describes a few of the integration problems that the KS-code presents and potential approaches to solving them; the analysis in this chapter is framed in terms of applying the KS-code to data being already being read and written in some pre-existing system of file storage and organization, making some basic assumptions about how general-purpose computers typically access storage devices and manage data in order to inform and, in some cases, simplify the conclusions being made. The set of information and actors necessary to store and retrieve data in non-volatile memory, especially when applying the coding processes (where applicable), is referred to herein generically as the "storage system," and may or may not include some or all of: the operating system(s) of the computer itself, the SSD memory controller, and the encoding manager. Essentially, depending on where the locus of control is for the processes of applying and storing/ retrieving endings (whether it be in hardware or software, an integrated design within the SSD itself or some kind of peripheral coding device, etc.) the storage system may or may not have access to certain information, such as the system of addressing used by the operating system or the status of each page on the SSD. All of this is dependent on how the KS-code is applied, and so this chapter tries to remain as general as possible when discussing such issues.

4.1 Data Addressing

A storage system requires a method of identifying where data is stored on a storage device in order to retrieve it later. Current storage devices are, from the system's perspective, just a context-free collection of storage locations which may or may not contain data, so it is up to the systems using them to impose a structure on the data stored (or not stored) therein. While this can be as simple as maintaining a large table that associates each address in storage with some unique tag for each data set therein, the system has to keep some record of what it is writing and where in order to effectively utilize the storage. This process is straightforward for un-encoded data, as the system determines exactly where and how much data it will write and can save this information accordingly. When using KS-coding, however, data being written to storage is expanded, creating a disparity between

the size of the data in system memory and its size in storage; for primary encodings, this expansion is at least predictable, but secondary encodings may represent any amount of data in a certain range, leading to unpredictable I/O results if left unaccounted for. To correctly store and address to encoded data, the system will either need to be aware of and account for this expansion by allocating additional storage addresses to the data being stored, or have its addresses translated via some mechanism such that the logical addresses used by the device still correspond to the expected amount of data on disk once decoded.

4.1.1 File Systems and Clustering

Most storage systems store and organize data in discrete data objects called **files** [17]. Although files, as a logical construct, can be simply modeled as a contiguous one-dimensional array of bytes, they are rarely represented as such in storage. Instead, a computer reads and writes to files as a series of uniformly sized data chunks called **clusters**, each of which contains a distinct segment of the overall file. Each cluster can be written or read individually to almost[1] any location in storage, and the system represents each stored file according to the ordering and storage address of its constituent clusters. It accomplishes this by cataloging this file cluster information in one or more of a set of "meta-files" that it uses to maintain the properties and disk locations of each of its files. This set of meta-files constitutes a **file system**, and allows a computing device to maintain and organize many files across one or multiple storage devices. File systems can typically be configured to use one of several different cluster sizes when being created, but most file systems will define cluster size as a simple multiple of the **address block size** (not to be confused with a flash block) in order to ensure that each address only ever contains data from one cluster at a time, with 4KiB per cluster being a common default [18].

4.1.2 Addressing Adaptations for Encoded Data

The methods of addressing used for clusters in a file system are all particularly well suited to keeping track of constant-size, logical power-of-two units of data, but the encoding process challenges both of these assumptions for data being written in storage. In order to correctly identify and retrieve such data, it may be necessary to create new addressing schemes and methods of record-keeping. All such methods of addressing should both provide an unambiguous way of translating an un-encoded address space to the encoded one and provide a means of accessing each stored page individually for retrieval.

4.2 Data Storage

The size of a cluster is a constant property of the file system and is set during its creation; for many systems, the default cluster size is 4KiB [18], and so this cluster size will be assumed during the bulk of the storage performance analysis in this and the following sections, but the conclusions remain largely the same; the primary consequence of file clustering on the data coding and storage process

[1] The system reserves certain addresses on each storage device, usually at the beginning of the disk's address space, in order to store boot-loaders and information about the disk's contents at a known location.

is just that data will be stored and retrieved according to some constant number of un-encoded bytes. Unfortunately, this method of encoding can produce outputs which are less than favorable for efficiently storing and retrieving them, requiring certain modifications or compromises be made in order to be able to correctly read and write encoded data. This section discusses the general parameters of this problem, while the following two sections deal with specifics related to storing primary and secondary encodings, respectively.

4.2.1 Encodings and Page Alignment

Having a constant unit of data to encode removes at least one source of variance in the encoding process, as it can be reasonably assumed that the input to and output from the encoding and decoding processes will be of a consistent size. This makes it easier to reason about how data will be affected by the encoding process, and how to store the subsequent encodings, but also imposes an important constraint on how the data is retrieved and decoded: however the cluster encodings are stored, it must be possible to retrieve and decode these clusters arbitrarily when the storage system requests them.

Recall from chapter 1 that a page in an SSD is the minimum unit of physical storage that can be accessed in one read or write operation by the SSD read circuit, and that, because of this, page-aligned access is preferred over unaligned access for optimal drive performance. Each of these pages on a given SSD typically represents a power-of-two number of cells (plus some additional cells to store page status information and error checking/ correcting codes), or anywhere from 512 bytes to 16KiB of data. On a typical system, maintaining page alignment is fairly easy, as clusters and pages tend to both be simple powers of two, resulting in a uniform relationship between pages per cluster or vice versa. Even in cases where the file cluster size is less than the page size, the SSD memory controller can still map to and fill portions of these pages relatively efficiently as long as the total number of pages therein aligns to the page size [11]. As the encoding process alters the size of these clusters during storage, however, this uniform size relationship may be lost, and so the relationship between the size of these encodings and the SSD page size plays a significant role in determining how to most efficiently store this data.

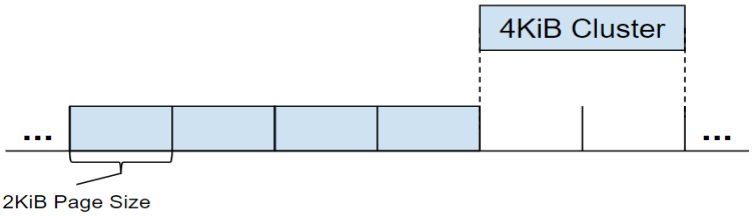


Figure 4.1: A page-aligned 4KiB cluster fits exactly within page boundaries for a 2KiB page size

4.2.2 Modifying Cluster Size

Although not proposed as a desirable solution due to its lack of portability, the simplest way to guarantee aligned disk access is to simply require that the system store data in units such that the size of the resulting encodings approximately match a whole multiple of the page size. This could be accomplished for primary encodings by redefining the file cluster size in terms of its *post*-encoding storage utilization but would necessitate significant changes to the way the system accesses and manages file data in both memory and storage [17] [19]. Taking this approach with secondary encodings would additionally require the system to support a variable unit of file transfer and addressing, and both of these adaptations would need to be highly specific to the particular encoding methodology being used. While this strategy does result in conveniently-sized encodings for storage, it would also entail producing a file system and corresponding API completely tailored to this type of storage, significantly complicating the file I/O process for the system in the process. Any method of applying the KS-code to file I/O should be transparent to the file access system, both to avoid over-reliance on the system's particular behavior in order for the coding method to work and to preserve the standard storage device interface that most systems expect. Figure 4.2 shows an illustration of an altered cluster being encoded and stored.

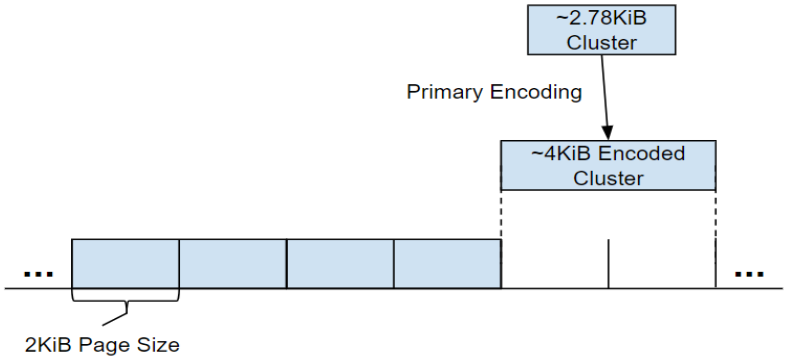


Figure 4.2: Altering cluster size to match its *encoded* size guarantees correct page alignment, but requires significant changes to device file access

4.3 Primary Encoding Storage

Primary cluster storage requirements are relatively easy to analyze due to the fact that these encodings, owing to the consistent size of the data to be encoded, are all uniform in size for a given cluster size and encoding degree. Unfortunately, due to the fact that these encodings do not necessarily align to a common flash page size, storing them to the pages of an SSD can result in misaligned access, storage overhead, or both. To illustrate this problem, consider a 4KiB cluster encoded with a degree of 3, resulting in a primary encoding of 7424 bytes, or exactly 7.25KiB. Unless this encoded cluster is being written to an SSD with a page size of 256 bytes or smaller (of which no examples currently exist), it will be unable to fully occupy all of the pages to which it is written; assuming a page size of 2KiB, storing the encoding will require at least 4 pages, or a total 8KiB of storage utilization, leaving 0.75KiB potentially unused in its final page. Writing additional encoding data to

this unused partial page is desirable in order to store encodings efficiently and improve compression ratios, but may also be costly from a read/ write performance perspective. This section examines the trade-off between maximizing storage utilization and maximizing performance when storing primary cluster encodings. Figure 4.3 demonstrates this effect.

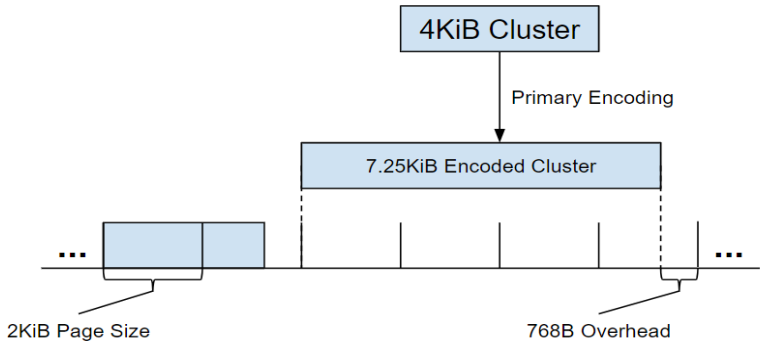


Figure 4.3: Primary encoding expansion can lead to non-standard storage unit, resulting in storage space overhead to maintain alignment

4.3.1 Primary Encoding Aligned Storage

Writing each encoding to the beginning of a page, thus allowing some storage overhead to persist, ensures that all primary encoding access remains aligned and that each encoding will occupy the fewest number of pages possible; encodings are essentially treated as though they are 0-padded to the size of the next page. This is beneficial for performance, as it ensures that no time is wasted reading or writing to "extra" pages as a result of data misalignment. This method of storage also makes addressing simpler for primary encodings due to the fact that page offsets do not need to be stored or calculated in order to identify which encoding data belongs to; mapping a logical page address to an encoding address just requires a simple multiplication by the data expansion factor, rounded up to account for this overhead space.

Maintaining cluster alignment can, however, waste a considerable amount of space if the partial pages they are stored to are left unused. Table 4.1 shows the amount of potentially unused storage for varying page sizes and encoding degrees, assuming a cluster size of 4KiB, while tables 4.2 and 4.3 show the minimum and maximum possible compression ratios when taking into account this storage waste.[2] Although storage overhead is usually small, typically no more than a couple of kibibytes, it is assessed for each cluster written to the storage device, resulting in a significant loss of space over many encodings; certain combinations of encoding and page size can result in a storage penalty of nearly 50% if the pages they occupy are not fully utilized. Some encoding degrees avoid this issue altogether by having an encoding size that does align to the SSD page boundary, which may be a reason to prefer these encodings over others.

[2]Unused space is treated as a set of ideally padded bits for secondary storage analysis, allowing some of this wasted space to be "reclaimed" in the second round of writing.

Table 4.1
Storage Overhead for Encoded 4KiB Cluster for Varying Degree and Page Size

Degree/ Page Size (B)	512	1024	2048	4096	8192	16384
2	0.25	0.25	0.25	2.25	2.25	10.25
3	0.25	0.75	0.75	0.75	0.75	8.75
4	0	0.5	1.5	3.5	7.5	7.5
5	0.25	0.25	0.25	2.25	6.25	6.25
6	0.13	0.13	1.13	1.13	5.13	5.13
7	0	0	0	0	4	4
8	0	0	1	3	3	3
9	0	0	0	2	2	2
10	0	0	1	1	1	1

Table 4.2
Minimum Compression Ratio for Encoded 4KiB Cluster w/Storage Overhead

Degree/ Page Size (B)	512	1024	2048	4096	8192	16384
2	1	1	1	0.88	0.88	0.69
3	0.94	0.92	0.92	0.92	0.92	0.79
4	0.9	0.89	0.88	0.85	0.83	0.83
5	0.85	0.85	0.85	0.84	0.83	0.83
6	0.82	0.82	0.82	0.82	0.82	0.82
7	0.79	0.79	0.79	0.79	0.81	0.81
8	0.77	0.77	0.78	0.79	0.79	0.79
9	0.75	0.75	0.75	0.77	0.77	0.77
10	0.73	0.73	0.74	0.74	0.74	0.74

Table 4.3
Maximum Compression Ratio for Encoded 4KiB Cluster w/Storage Overhead

Degree/ Page Size (B)	512	1024	2048	4096	8192	16384
2	1.17	1.17	1.17	1	1	0.75
3	1.19	1.16	1.16	1.16	1.16	0.91
4	1.22	1.19	1.15	1.08	1	1
5	1.2	1.2	1.2	1.13	1.05	1.05
6	1.19	1.19	1.16	1.16	1.08	1.08
7	1.19	1.19	1.19	1.19	1.11	1.11
8	1.18	1.18	1.16	1.13	1.13	1.13
9	1.17	1.17	1.17	1.14	1.14	1.14
10	1.17	1.17	1.15	1.15	1.15	1.15

4.3.2 Unaligned Storage and Super-clustering

While this is made more complicated by the nonstandard (i.e. non-power-of-two) sizes produced by the primary encoder, storage overhead can be reduced or eliminated by writing additional encoding data to the extra page space after another encoding; while the hardware access unit remains constant at one page, the SSD FTL typically allows addressing granularity down to units of as little as 512

bytes, which means writing to any overhead space greater than or equal to 512 bytes is possible [11]. While this will improve the storage utilization of the device, it also means that many clusters will be written across page boundaries, resulting in additional clusters needing to be accessed in order to read and write each encoding. Returning to the previous example, wherein a 7.25KiB encoding had occupied 8KiB of storage space, if a second 7.25KiB cluster encoding is stored by writing a portion of its data to the overhead space within the previous encoding's page, the result will be 512 bytes stored to this overhead space, plus an additional 6.75KiB stored across another 4 pages, with a further 1.25KiB of overhead for this encoding to potentially be filled by the next. This has the effect of reducing the storage overhead for each cluster to 0.25KiB on average, but also requires that an additional page be accessed for 3 out of every 4 clusters stored. Figure 4.4 illustrates this unaligned storage method.

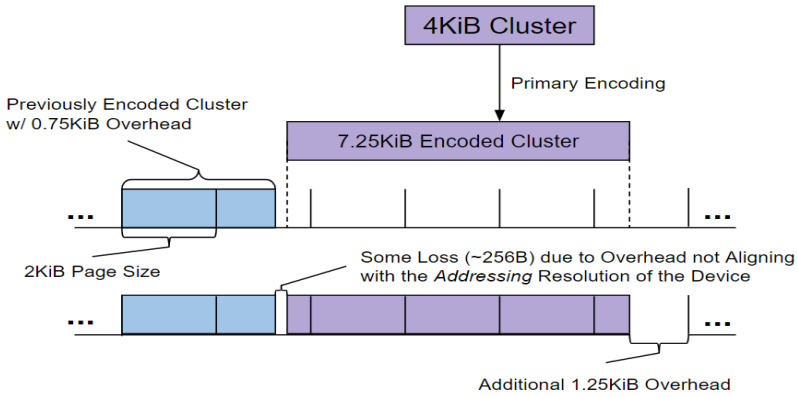


Figure 4.4: Storing to overhead space can reduce memory waste, but creates unaligned pattern of access; the file shown in purple now occupies 5 pages instead of the minimal 4, requiring more access time to retrieve

A potentially better solution to this problem is to group multiple cluster encodings together such that their combined size aligns to a multiple of the page size, a technique we refer to as **super-clustering**; although primary encoding sizes typically do not match an integer multiple of the page size, combining several of these encodings will eventually produce a data unit that does, as illustrated in figure 4.5. Similar to how clusters that are smaller than a single page can be grouped together to form a read/ write unit that can fully utilize the storage hardware, super clusters are able to exactly match the size of a certain number of flash pages, and are read and written as a single unit to avoid the inefficiency associated with access misalignment. Table 4.4 lists the minimum number of encoded 4KiB clusters necessary to align to given hardware page size for varying encoding degree. This approach has the potential to provide near-ideal performance and storage efficiency, provided the super clusters are managed properly. Once written, super clusters will be read out as a single unit, and so capitalizing on that entire read is crucial to maximizing the performance of this storage technique; if only one cluster within the super cluster is actually needed, the rest of the reads performed are wasted. Fortunately, data written to mass storage devices is often done so sequentially, which means that there is a much better chance for the clusters in a super cluster, having been written at the same time, to be read at the same time as well.

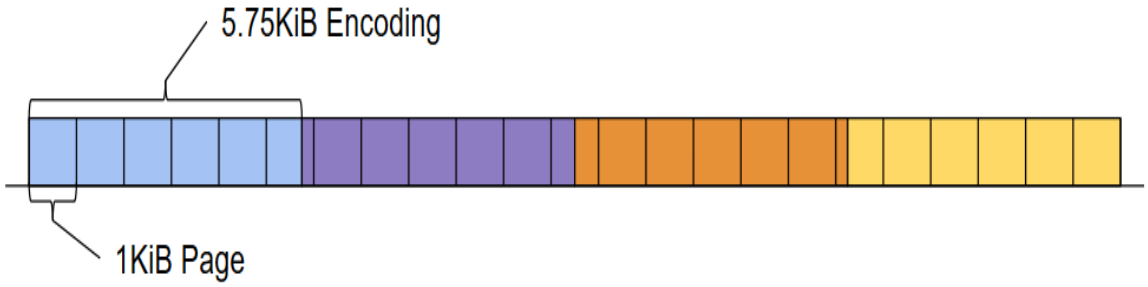


Figure 4.5: 4 clusters encoded with degree 2 results in 23KiB of encoded data, which aligns perfectly to a 1KiB page size

Table 4.4
Minimum Number of 4KiB Clusters Required for Perfect Page Alignment

Degree/ Page Size (B)	512	1024	2048	4096	8192	16384	Pages Occupied
2	2	4	8	16	32	64	23
3	2	4	8	16	32	64	29
4	1	2	4	8	16	32	17
5	2	4	8	16	32	64	39
6	4	8	16	32	64	128	87
7	1	1	1	1	2	4	24
8	1	1	2	4	8	16	26
9	1	1	1	2	4	8	28
10	1	1	2	4	8	16	30

4.4 Storage Alignment for Secondary Encodings

As is the case for primary encodings, reading and writing clusters in the second round of coding is complicated by the fact that the size of the data is altered. Unlike primary cluster encodings, however, secondary encodings provide no fixed relationship between the size of the input data and the size of the encoding in storage; the secondary encoding of a given cluster can be of any size within a given range, depending on the encoding degree being used, making it much more difficult to guarantee any access alignment properties while maintaining space efficiency. One additional change to the typical write process for secondary encoding operations is that every write now requires a read as well in order to supply the existing page data to the encoder for overwriting. Although reading is by far the fastest of the basic operations of an SSD [16], this will, in addition to the encoding process itself, add some small amount of overhead to the writing process. The primary challenge here will be coming up with an efficient method of storing and retrieving cluster data such that each cluster can be readily identified and that enough secondary data bits are used to justify this method of encoding in the first place.

Depending on how the storage of primary encodings is handled, it may be useful to characterize secondary cluster access in terms of accessing other *clusters*, rather than pages of flash; while each encoded cluster is technically made up of a series of small encodings, each of which could be

overwritten individually, the boundaries between these encodings do not align to page boundaries, meaning that there is no way of knowing where each encoding starts for an arbitrary page taken from a stored encoded cluster. There are ways of solving this, either by again forcing this alignment through padding or by keeping track of an "encode word offset" for each page, but the benefit to such designs is marginal given the fact that this type of granularity is not typically needed for secondary encodings; the encoder will likely need many pages in order to store each secondary cluster, so requiring access to the entire primary cluster is not an excessive constraint, provided that cluster is stored and read in an efficient manner.

4.4.1 Minimal Write Secondary Storage

While the exact number of round two storage bits available in each encoded cluster can vary, there is at least a lower bound to this variance that will allow the system to make a few assumptions about storing secondary encodings. In this regard, the simplest approach to secondary storage would be to just treat primary encodings as fixed-size storage units based on the assertion that they all contain at least the minimum number of potential secondary storage bits and never attempt to write more data to them than that. Storing a cluster as a secondary encodings is then just a matter of writing the cluster data to as many primary encodings as is necessary given the minimum secondary capacity of the primary clusters for the chosen degree. This greatly simplifies the writing and addressing processes, as it does not require any additional information to be kept about the encodings, but may again result in significant storage overhead and compression performance loss due to the underutilization of clusters with a higher number of secondary data bits (as this approach essentially guarantees the minimum possible compression ratio for each encoding). Using these additional bits, however, means allowing a certain degree of variability into the addressing and storage access processes; supporting this variable access pattern is possible, but will require several additional pieces of information to be kept about the encodings.

4.4.2 Secondary Storage Space Management

In order to support writing variable amounts of data to varying numbers of primary encodings, it would be useful to at least know how much data can be stored to a given primary encoding beforehand. This can be done in a number of ways, but the best time to do so would be during the initial creation of the encoding itself. Once encoded, a "soft padding" can be applied to the result, applying the padding algorithm to determine where separator bits *would* be added if padding were to be applied but without changing the actual values of the encoding. Once this is done, the total number of current and potential separator bits are multiplied by the encoding degree, minus any data bits "lost" due to separators appearing in the final $m - 1$ bits of the encoding. The result of this calculation can then be associated with the corresponding encoding in order to inform the secondary encoding and storage process once that encoding is ready to be overwritten.

Once a primary encoded cluster has been padded and is ready to be written to, the storage system can essentially treat this encoding as a fresh quasi-page to be written to (with the caveat that its capacity may vary). Having this capacity information beforehand affords predictability, as the system will know ahead of time how much data will be stored when writing to a particular cluster and whether or not additional storage will be necessary beyond that. This also makes it possible for secondary storage locations to be chosen more efficiently by attempting to match the free space in

one or more encoded clusters with the incoming data cluster size (or a multiple thereof) as closely as possible, thus mitigating the effects of secondary storage overhead on compression performance.

Figure 4.6 demonstrates an example of this approach: the top image depicts two clusters overwritten opportunistically onto (padded) primary encodings of arbitrary secondary capacity, while the bottom image depicts the same two clusters stored as pairs of secondary encodings which were selected in an attempt to more closely match the size of the data being encoded. In both cases, the effective compression ratio is $\frac{24}{23} = 1.043$ (four 4KiB clusters stored as primary encodings, occupying 23KiB of memory, plus an additional two 4KiB clusters stored in a second round of writing), but the encodings in the second case can be accessed more efficiently due to the fact that each secondary encoded cluster only requires the reading and subsequent decoding of two secondary encodings, whereas the first case requires access to three secondary encodings for the second encoded cluster. Allowing multiple secondary clusters to "share" the same encoding also requires maintaining some offset information for each encoding therein to delimit each encoding during retrieval, which adds to the complexity and storage overhead of the addressing system.

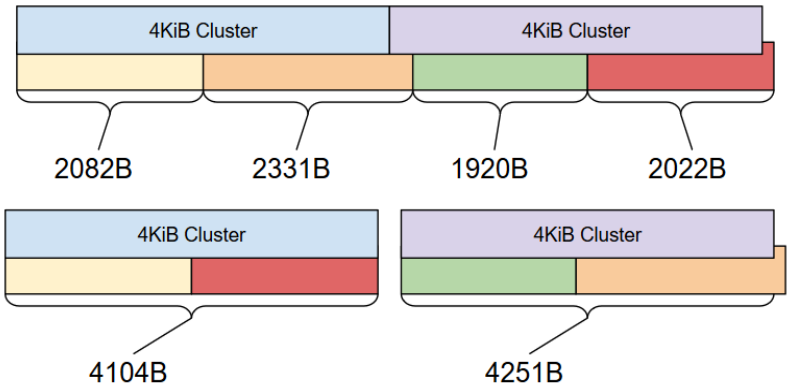


Figure 4.6: Two 4KiB file clusters are stored as secondary encodings over four primary encodings of degree 2 (each primary encoding represents 4KiB of data and occupies 5888B = 5.75KiB of flash memory). Top: secondary cluster data is overwritten onto padded primary encodings in no particular order (secondary storage capacity shown in bytes), with two clusters partially occupying a single encoding. Bottom: by selecting which encoding(s) to overwrite based on their capacity, it is possible to more efficiently store secondary data by minimizing overhead while keeping secondary data from overlapping on the same primary encoding.

4.5 Coding Policy

Apart from the concerns of storing and retrieving encoded data from the storage device, it is also necessary for the storage system to determine what encoding process should be applied to a given data object and at what time. In other words, the system must be able to identify whether an encoded cluster is a primary or secondary encoding when decoding it, as well as identifying which clusters on disk are eligible for padding and overwriting, and should have a method of determining when to apply these operations in order to maximize performance. This section discusses the information and criteria used to decide which round of encoding to employ, referred to generally as the **coding**

policy of the system; when discussing this policy, it will be important to distinguish between file clusters as a logical construct - representing the current state of each file in the system - and the representation of those clusters in storage, which may differ from the logical cluster when a file is modified.

To manage encodings on the storage device, the storage system might assign one of the following four states to each stored encoded cluster: (1) 'primary,' a cluster that has been encoded and stored according to the primary encoding procedure; (2) 'stale,' a stored primary cluster that has been superseded by an updated representation of the corresponding *logical* cluster; (3) 'padded,' a stale cluster that has been padded by the storage system and is now ready to be overwritten; and (4) 'secondary,' a cluster that has been overwritten and now contains secondary data bytes. Because clusters in storage must be created as primary encodings before a secondary encoding can be produced using their encoded data, each newly written cluster starts its life in the 'primary' state before progressing through each of the subsequent states in order. The system's coding policy then controls when each of these transitions takes place and applies the appropriate operation for the given cluster state at the chosen time. Coding policy activities include determining when a primary encoding has gone stale; deciding whether stale clusters should be padded immediately, during overwrite, or at another time; and choosing whether to encode new data as as one or more secondary encodings as soon as sufficient storage is available or whether additional primary clusters should be produced first. Once the data in a secondary cluster also becomes stale, then and only then are the *pages* containing that cluster marked stale, allowing their values to be erased via garbage collection and reclaimed as fresh pages.

4.5.1 Stale Cluster Detection

The first coding policy task for each cluster is to determine exactly when that cluster's data becomes stale. This task is fairly straightforward, as the storage system can simply infer a stale cluster whenever a new write to the logical address that that cluster was previously stored at takes place. It should be noted, however, that in order to prevent the premature deletion of the stale cluster, the storage system must also be capable of preempting the SSD from marking the *pages* in which the encoding is stored as stale; this is one of the primary challenges to employing a functioning rewriting code on SSDs, as they are constantly doing their own pruning, sorting, and garbage collecting of data in the background, and the system may not have any way of knowing about or preventing this, depending on where and how it is integrated with the device (e.g. as an external module, an integral component of the memory controller, or something in between). One way to do this might be to utilize an additional layer of write indirection, whereby the storage device is able to redirect writes from the host system to different logical addresses on the storage device. In any case, once this data has been "overwritten," its state can be updated to 'stale' and the policy can proceed to the next step.

4.5.2 Padding Policy

Once a page has been identified as stale, padding must be applied to that page before it is overwritten in order to maximize the stale page's secondary storage capacity.[3] While there is not anything

[3]Stale clusters are not technically required to be padded before overwriting, but this is almost never recommended in this case.

particularly interesting about this in and of itself - the padding process is straightforward and can be applied by the storage device at almost any time without any additional input - it is still worth considering the timing of this process, as this could, in theory, affect the performance of the device. Padding can be applied to a stale cluster at almost any time, but there are three instances where this is most opportune: as soon as the cluster is detected as stale, as soon as it is chosen for secondary storage, and during some time in between these when there is relatively little storage activity. These three strategies will be referred to respectively as 'immediate padding', 'on-demand padding', and 'background padding'.

Immediate padding, as the name implies, means that pages will be padded immediately once they are marked as stale, thus ensuring that these clusters will be ready for immediate overwriting when performing a secondary cluster encoding. This strategy can impact disk performance when updating clusters (as it would then be necessary to read, pad, and write the stale cluster back to disk in addition to writing the updated cluster elsewhere on disk), but ensures optimal round II write performance by preparing stale clusters for overwriting before attempting to write to them. On-demand padding, by contrast, only pads clusters as needed, performing the pad operation just before writing to the cluster. This strategy is more efficient, as the storage device only has to read and write the stale cluster once for both padding and overwriting the cluster, but slightly increases secondary write latency due to the fact that the encoding device now has to make two passes over the data: first to pad the stale data, then to overwrite the resulting cluster. The final padding strategy, background padding, relies on the storage system to determine when the device is not currently in use and to then automatically pad stale clusters while no other disk activity is occurring. This strategy attempts to pad the cluster for free, so to speak, by performing the operation while the storage device would otherwise be idle anyway. This could potentially eliminate the added latency resulting from either of the other two strategies, but may fail to keep up with demand for secondary storage space during periods of high disk activity, necessitating the use of on-demand padding as a fallback strategy.

4.5.3 Secondary Encoding Policy

After a cluster is padded using some combination of the above strategies, all that is left for the compression device to do is to overwrite it with new data. Once padded and staged for overwriting, the storage device can essentially use this cluster as it would any other available storage space on disk when storing new cluster data. Still, this presents yet one more decision for the storage device: whether to prefer, if possible, writing to the available padded cluster space, or whether to continue producing new primary clusters to expand the amount of potential secondary storage in the future.

Again, performance considerations will largely drive this decision making process, as performance for both read and write operations can vary based on the encoding round and degree of encoding used. For lower encoding degrees, primary encodings should generally preferred for both read and write performance, as they tend to be more information dense than their secondary encoding counterparts, occupying fewer physical pages in storage and thus requiring fewer physical accesses to read or write. As encoding degree increases, the inverse becomes true, with primary encodings occupying increasingly more space due to higher primary encoding expansion factors, while secondary encodings benefit from larger groups of free data bits bits interspersed within the padded encodings on disk. Secondary write operations are, again, slightly slower in that they require a full read-modify-write cycle, but this relatively small increase in latency is overshadowed by the superior bandwidth of the overwrite process for higher encoding degrees.

Another advantage of producing additional primary encodings is to eventually provide the secondary storage memory manager with additional padded encodings that can be used to produce potentially more favorable secondary storage units with lower overhead; a useful decision criterion would perhaps be to continue producing primary clusters until enough padded clusters have been created to provide a secondary storage unit that is within some threshold of maximum secondary storage overhead. Ultimately, most of the primary clusters produced will eventually become secondary clusters over time thus necessitating the creation of more primary clusters, and so a strategy of preferring one type of encoding or the other would likely converge to some average performance level over time, but maintaining a healthy stockpile of available secondary storage should allow the system to keep secondary cluster overhead low while providing some leeway for short-term bursts of secondary writes should that ever be advantageous.

5. Conclusion

This report outlines the structure and function of flash memory, its application to bulk data storage in the form of so-called solid state drives, and discusses the limitation of "block erasure" inherent to such memories. Additionally, it describes a method of data encoding as a means to reduce the impact of this limitation on drive performance, and provides two implementations of this encoding method: a software implementation written in C, and a hardware implementation designed using the SystemVerilog HDL. Due to the limited nature of the project scope within the context of a full implementation of this file compression method, many details related to the process of storing and addressing to the encodings produced by such encodings are left unresolved in the implementations themselves, but the report provides a summary of their effects on current methods of secondary data storage and discusses several approaches to altering these storage methods to better suit this encoded data.

Ultimately, our analysis indicates that this compression scheme, while reducing total storage device capacity and negatively impacting read times, could be useful for write heavy applications, and that the hardware implementation herein provided should be capable of useful data throughput via hardware repetition, allowing many encodings to be performed in parallel. The reduction in the frequency of block erase operations when rewriting to flash has the potential to be highly beneficial to the write performance of flash devices due their outsize effect on the speed and latency of writes which necessitate them, especially for flash devices with high storage space utilization (and thus fewer fresh pages to allow the device to "hide the latency" of block erasure). The potential performance benefit of this code is hampered by the fact that its effectiveness is highly dependent on the properties of the data that it is applied to, as useful compression performance can vary by up to 50% for this class of encodings based on the value of the input data, but a preliminary experimental evaluation of the average-case encoding characteristics for various file types shows promising results, with many of these data sets producing compression ratios closer to the theoretical maximum than the minimum.

Although the coding techniques described in the literature are implemented here as functional hardware/ software modules, further work will be necessary to produce a functional realization of this KS-code that can be applied effectively to current storage drives. This report describes a general set of approaches to this problem but, as discussed earlier, many details will have to be considered for each particular application of the code, and the design decisions made in doing so could potentially have a significant impact on the overall complexity and performance of the product. While it is possible to estimate the performance of such applications of the code based on some assumptions made about the nature of the hardware and the data being stored, a full practical evaluation of the performance benefits of this method of storage (or, possibly, the lack thereof), necessitating a complete implementation of these coding and storage processes, is necessary for a definitive conclusion as to its usefulness.

References

[1] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to flash memory," *Proceedings of the IEEE*, vol. 91, pp. 489–502, April 2003.

[2] S. H. Kim and J. W. Kwak, "Garbage collection technique using erasure interval for nand flash memory-based storage systems," *International Journal of Applied Engineering Research*, vol. 11, no. 7, pp. 5188–5194, 2016.

[3] S. T. Klein and D. Shapira, "Boosting the compression of rewriting on flash memory," in *2014 Data Compression Conference*, pp. 193–202, March 2014.

[4] A. Inoue and D. Wong, *NAND Flash Applications Design Guide*. Toshiba America Electronic Components, Inc., 5231 California Ave., Irvine, CA 92617, USA, 2 ed., March 2004.

[5] L. C. et al., "Nonvolatile memories: Nor vs. nand architectures," in *Memories in Wireless Systems*, Signals and Communication Technology, pp. 29–53, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[6] A. Tal, "Two flash technologies compared: Nor vs nand," tech. rep., M-Systems, 2002.

[7] K. Eshghi and R. Micheloni, "Ssd architecture and pci express interface," in *Inside Solid State Drives (SSDs)*, vol. 37 of *Springer Series in Advanced Microelectronics*, pp. 19–45, Dordrecht, Netherlands: Springer Netherlands, 2013 ed., 2013.

[8] R. N. J. Kadlec and R. Kuchta, "Nand flash memory organization and operations," *Journal of Information Technology & Software Engineering*, vol. 5, no. 1, pp. 1–8, 2015.

[9] R. Stoica and A. Ailamaki, "Improving flash write performance by using update frequency," *Proc. VLDB Endow.*, vol. 6, pp. 733–744, July 2013.

[10] T. C. et. al., "A survey of flash translation layer," *Journal of Systems Architecture*, vol. 55, no. 5, pp. 332 – 343, 2009.

[11] "hymap," tech. rep., Hyperstone Inc., 465 Corporate Square Drive, Winston-Salem, NC 27105, USA, March 2015.

[12] R. L. Rivest and A. Shamir, "How to reuse a "write-once memory"," *Information and Control*, vol. 55, no. 1, pp. 1 – 19, 1982.

[13] Y. Wu and A. Jiang, "Position modulation code for rewriting write-once memories," *IEEE Transactions on Information Theory*, vol. 57, pp. 3692–3697, June 2011.

[14] E. Zeckendorf, "Reprsentation des nombres naturels par une somme de nombres de fibonacci ou de nombres de lucas," *Bulletin de la Societe Royale de Liege*, vol. 41, pp. 179–182, 1972.

[15] S. Klein, "Combinatorial representation of generalized fibonacci numbers," *Fibonacci Quarterly*, 1998.

[16] "Nand flash memory," tech. rep., Micron Technology, 8000 S. Federal Way, P.O. Box 6, Boise, ID 83707, 2006.

[17] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005.

[18] R. Gerend and L. Poggemeyer, "Ntfs overview." Web, September 2018.

[19] R. Kieth, "Managing memory-mapped files." Web, February 1993.

[20] S. K. et. al., "Fast, energy efficient scan inside flash memory ssds," in *The Second International Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures*, 2011.

[21] "Partition alignment of intel ssds for achieving maximum performance and endurance," tech. rep., Intel Corporation, February 2014.

A. Implementation Source Code

A.1 Software Implementation

A.1.1 FZcompress.h

```
/*****************************************************************
 * FZcompress.h
 *****************************************************************/
#ifndef FZ_COMPRESS_H_
#define FZ_COMPRESS_H_

#include <stdint.h>

#define FUNC_ENC        "encode"
#define FUNC_DEC        "decode"
#define FUNC_PAD        "pad"
#define FUNC_OVR        "overwrite"
#define FUNC_EXT        "extract"

#define MSG_MEM_ERR     "Error allocating memory for series, ↩
    aborting operation\n"
#define MSG_READ_ERR    "Error reading from input file, aborting ↩
    operation\n"
#define MSG_WRITE_ERR   "Error writing to output file, aborting ↩
    operation\n"
#define MSG_FORMAT_ERR  "Error formatting output file, aborting ↩
    operation\n"
#define MSG_WRONG_FMT   "Invalid file format, input must be ↩
    previously encoded file\n"
#define MSG_WRONG_ENCD  "Wrong encoding type, input file must be ↩
    unpadded primary encoding\n"
#define MSG_BAD_OFFST   "Invalid offset for coding word size\n"
#define MSG_EMPTY_WARN  "Warning: input file(s) empty, no operation↩
     performed\n"
```

```
#define LONG_WORDS          0

#if LONG_WORDS == 1
typedef uint64_t data_word;
#else
typedef uint32_t data_word;
#endif

/* This 2 byte signature will identify each encoded file created by this
 * library, and resembles the characters "FIBO"
 */
#define MAGIC_WORD          0xF1B0
#define HEADER_BYTES        8
#define BYTE_BITS           8
#define WORD_SIZE           sizeof(data_word)
#define WORD_BITS           (WORD_SIZE * BYTE_BITS)

/*
 * Header structure used to identify encoded files.
 *
 * MEMBERS:
 *   uint16_t magic - 2 byte "magic number" to ID files as created by this
 *       library.
 *   uint8_t type   - 0 for primary, 1 for padded, 2 for secondary
 *   uint8_t offset - Number of bytes represented by last encoded word in file;
 *       this is needed because the encoded file always aligns to a number of
 *       encoded words (plus a few bits of padding to round out the last byte,
 *       which are ignored) but the file being encoded may provide 1-4 bytes
 *       to produce the final word. Since there isn't a way to determine how
 *       many trailing bytes from the last encoding belong to the file and how
 *       many are just padding, this number is saved during primary encoding.
```

```
 *      A 1 in the first (most significant) bit signifies a "long word", i.e.
 *      64-bit, encoding unit size.
 *   uint32_t extra_bytes - Used to store the number of secondary data bytes
 *      available for secondary encodings. Supports secondary encoding of
 *      files up to ~4.2GB
 */
typedef struct fz_header_s {
    uint16_t magic;
    uint8_t type;
    uint8_t offset;
    uint32_t extra_bytes;
} fz_header_t;

/*
 * Encodes a file according to its Zeckendorf sum representation.
 *
 *   Inputs:
 *     FILE* in - open file descriptor pointing to the file to be encoded
 *     FILE* out - open file descriptor pointing to the file in which to write
 *         the encoded file data
 *     int deg - the encoding degree to be used
 *
 *   Outputs:
 *     FILE* out - a file containing the output encoded data
 *
 *   Return:
 *     int - 0 if successful, -1 on any error condition
 */
int encode_file     (FILE* in, FILE* out, int deg);

/*
 * Decodes a file encoded as a set of Zeckendorf sums.
 *
 *   Inputs:
 *     FILE* in - open file descriptor pointing to the file to be decoded
```

```
 *     FILE* out - open file descriptor pointing to the file in ↩
    which to write
 *         the decoded file data
 *     int deg - the encoding degree used to originally encode the ↩
    file
 *
 *   Outputs:
 *     FILE* out - a file containing the output decoded data
 *
 *   Return:
 *     int - 0 if successful, -1 on any error condition
 */
int decode_file     (FILE* in, FILE* out, int deg);

/*
 * Pads a file such that no strings of more than 2*'deg' - 2 0's ↩
    appear therein.
 * Intended for use with file data encoded according to the ↩
    algorithm
 * implemented by encode_file().
 *
 *   Inputs:
 *     FILE* in - open file descriptor pointing to the file to be ↩
    padded
 *     FILE* out - open file descriptor pointing to the file in ↩
    which to write
 *         the padded file data
 *     int deg - the encoding degree used to encode the input file
 *
 *   Outputs:
 *     FILE* out - a file containing the output padded data
 *
 *   Return:
 *     int - number of secondary bytes in padded encoding if ↩
    successful,
 *         -1 on any error condition
 */
int pad_file        (FILE* in, FILE* out, int deg);

/*
```

```
 * "Overwrites" a file by interspersing data from 'over' into the ↩
    data values
 * read from 'in'. Input file must obtain certain data properties ↩
    in order for
 * this process to work correctly and to be reversable; the ↩
    encode_file()
 * function can be used to obtain such a file.
 *
 *   Inputs:
 *     FILE* in - open file descriptor pointing to the file to be ↩
    overwritten
 *     FILE* out - open file descriptor pointing to the file in ↩
    which to write
 *         the overwritten file
 *     FILE* over - open file descriptor pointing to the file from ↩
    which to
 *         obtain data to write "onto" the input file
 *     int deg - the encoding degree used to encode the input file
 *
 *   Outputs:
 *     FILE* out - a file containing the output encodings
 *
 *   Return:
 *     int - 0 if successful, -1 on any error conditions
 */
int overwrite_file  (FILE* in, FILE* out, FILE* over, int deg);

/*
 * Extracts data that has been "overwritten" into another file by ↩
    the algorithm
 * implemented in overwrite_file().
 *
 *   Inputs:
 *     FILE* in - open file descriptor pointing to the file to be ↩
    encoded
 *     FILE* out - open file descriptor pointing to the file in ↩
    which to write
 *         the encoding
 *     int deg - the encoding degree to be used
 *
 *   Outputs:
```

```
 *     FILE* out - a file containing the output encodings
 *
 *   Return:
 *     int - 0 if successful, -1 on any error condition
 */
int extract_file    (FILE* in, FILE* out, int deg);

#endif
```

A.1.2 FZcompress.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "FZcompress.h"

#define DEBUG 1

static int encode_bits;

/*
 * Returns the value of the bit which is 'offset' positions from ↩
    the left-most
 * (most significant) bit in the given 'byte'.
 *
 *   Inputs:
 *     uint8_t byte - a byte of data
 *     int offset - the bit offset into the given byte to examine, ↩
    starting
 *          from the most significant bit
 *
 *   Outputs:
 *     None
 *
 *   Return:
 *     Non-zero if the bit value at the specified offset is 1, 0 ↩
    otherwise
 */
static uint8_t get_char_bit(uint8_t byte, int offset) {
```

```
    return byte & (0x1 << (BYTE_BITS - offset - 1));
}

/*
 * Sets the value of the bit which is 'offset' positions from the left-most
 * (most significant) bit in the given 'byte' to 'value'
 *
 *   Inputs:
 *     uint8_t* byte - a pointer to a byte of data
 *     int offset - the bit offset into the given byte to set, starting
 *          from the most significant bit
 *     int value - the value to assign to the chosen bit; an input of 0 sets
 *          the selected bit to 0, a non-zero input sets it to 1
 *
 *   Outputs:
 *     uint8_t* byte - a pointer to the updated byte value
 *
 *   Return:
 *     None
 */
static void set_char_bit(uint8_t* byte, int offset, int value) {
    uint8_t mask = 0x1 << (BYTE_BITS - offset - 1);
    *byte = (value != 0) ? (*byte | mask) : (*byte & ~mask);
}

/*
 * Reads in the first 8 bytes of a given file as header data matching the
 * structure of an fz_header_t. Bytes are read in big-endian byte order, so
 * the first byte of the file is the most significant byte of the first value
 * in the header.
 *
 *   Inputs:
 *     FILE* file - an open file pointer to read header info from
 *     fz_header_t* header - a pointer to a header of type fz_header_t (see
```

```
 *          FZcompress.h for header info)
 *
 *   Outputs:
 *     fz_header_t* header - the header structure filled with the ↩
    header info
 *          read from the file
 *
 *   Return:
 *     int - 0 if the reading process was successful, -1 otherwise
 */
static int read_header(FILE* file, fz_header_t* header) {
    int bytes_read = 0;
    uint8_t in_byte;

    bytes_read += fread(&in_byte, sizeof(in_byte), 1, file);
    header->magic = in_byte << 8;
    bytes_read += fread(&in_byte, sizeof(in_byte), 1, file);
    header->magic |= in_byte;
    bytes_read += fread(&in_byte, sizeof(in_byte), 1, file);
    header->type = in_byte;
    bytes_read += fread(&in_byte, sizeof(in_byte), 1, file);
    header->offset = in_byte;
    bytes_read += fread(&in_byte, sizeof(in_byte), 1, file);
    header->extra_bytes = in_byte << 24;
    bytes_read += fread(&in_byte, sizeof(in_byte), 1, file);
    header->extra_bytes |= in_byte << 16;
    bytes_read += fread(&in_byte, sizeof(in_byte), 1, file);
    header->extra_bytes |= in_byte << 8;
    bytes_read += fread(&in_byte, sizeof(in_byte), 1, file);
    header->extra_bytes |= in_byte;

    if(bytes_read != HEADER_BYTES) {
        return -1;
    } else {
        return 0;
    }
}


/*
```

```
 * Writes 8 bytes of header information from the provided fz_header_t header
 * to a given file. It is generally assumed that the provided file pointer will
 * point to the beginning of the file (although there's technically nothing to
 * stop you from not doing so).
 *
 *   Inputs:
 *     FILE* file - an open file pointer to write header info into
 *     fz_header_t header - a header structure containing values to be written
 *          (see FZcompress.h for header info)
 *
 *   Outputs:
 *     none
 *
 *   Return:
 *     int - 0 if the write process was successful, -1 otherwise
 */
static int write_header(FILE* file, fz_header_t header) {
    int bytes_written = 0;

    uint8_t out_byte = (uint8_t)(header.magic >> 8);
    bytes_written += fwrite(&out_byte, sizeof(out_byte), 1, file);
    out_byte = (uint8_t)header.magic;
    bytes_written += fwrite(&out_byte, sizeof(out_byte), 1, file);
    out_byte = header.type;
    bytes_written += fwrite(&out_byte, sizeof(out_byte), 1, file);
    out_byte = header.offset;
    bytes_written += fwrite(&out_byte, sizeof(out_byte), 1, file);
    out_byte = (uint8_t)(header.extra_bytes >> 24);
    bytes_written += fwrite(&out_byte, sizeof(out_byte), 1, file);
    out_byte = (uint8_t)(header.extra_bytes >> 16);
    bytes_written += fwrite(&out_byte, sizeof(out_byte), 1, file);
    out_byte = (uint8_t)(header.extra_bytes >> 8);
    bytes_written += fwrite(&out_byte, sizeof(out_byte), 1, file);
    out_byte = (uint8_t)header.extra_bytes;
    bytes_written += fwrite(&out_byte, sizeof(out_byte), 1, file);

    if(bytes_written != HEADER_BYTES) {
```

```
        return -1;
    } else {
        return 0;
    }
}

static int read_word_bigend(FILE* file, data_word* in_word) {
    int bytes_read = 0;
    *in_word = 0;

    for(int i = sizeof(*in_word) - 1; i >= 0; i -= 1) {
        uint8_t in_byte;
        if(fread(&in_byte, sizeof(in_byte), 1, file) == 1) {
            *in_word |= (in_byte << BYTE_BITS*i);
            bytes_read += 1;
        } else {
            // Signal error to program if non-EOF read failure ↩
               occurred
            if(feof(file) == 0) { bytes_read = -1; }
            break;
        }
    }

    return bytes_read;
}

static int write_word_bigend(FILE* file, data_word out_word, int ↩
   num_bytes) {
    int bytes_written = 0;

    for(int i = sizeof(out_word) - 1; i >= (signed)(sizeof(out_word↩
       ) - num_bytes); i -= 1) {
        uint8_t out_byte = (uint8_t)(out_word >> (BYTE_BITS*i));
        if(fwrite(&out_byte, sizeof(out_byte), 1, file) == 1) {
            bytes_written += 1;
        } else {
            break;
        }
    }

    return bytes_written;
```

```
}

/*
 * Generates an extended Fibonacci series with a given degree 'm'. ↩
    Note that
 * the series is constructed in descending order, with the largest ↩
    series value
 * in the lowest index of the array and vice versa.
 *
 *   Inputs:
 *     data_word* buffer - a pointer to an unsigned integer array ↩
    in which to
 *         store the generated series
 *     int m - the degree of extended Fibonacci series to be ↩
    generated
 *     int size - the number of series elements to generate
 *
 *   Outputs:
 *     data_word* buffer - a pointer to the calculated series ↩
    elements
 *
 *   Return:
 *     None
 */
static void generate_series(data_word* buffer, int m,  int size) {
    for(int i = size - 1; i >= 0; i -= 1) {
        if((size - i - 1) < m) {
            buffer[i] = size - i;
        }
        else {
            buffer[i] = buffer[i + 1] + buffer[i + m];
        }
    }
}

/*
 * Calculates the true number of series elements required to ↩
    represent the
 * maximum value of a given number of bits by performing a "short ↩
    calculation"
 * of the desired series.
```

```
 *
 *   Inputs:
 *     int m - the degree of extended Fibonacci series to use
 *     int bits - the number of bits to be encoded
 *
 *   Outputs:
 *     none
 *
 *   Return:
 *     int - Number of series elements needed to store all values ↩
    of size
 *         'bits' for degree 'm'
 */
static int get_series_len(int m, int bits) {
    data_word max_val = (unsigned)(-1) >> (sizeof(max_val)*↩
       BYTE_BITS - bits);
    data_word* values = (data_word*)calloc(m, sizeof(*values));
    if(values == NULL) { return -1; }

    for(int i = 0; i < m; i += 1) { values[i] = i+1; };
    if(max_val <= m) return max_val;

    int len = m;

    while(values[m-1] < max_val) {
        // Check for overflow
        if(((values[m-1] >> 1) + (values[0] >> 1)) <= (max_val >> ↩
           1)) {
            uint64_t next_val = values[m-1] + values[0];
            for(int i = 0; i < m - 1; i += 1) {
                values[i] = values[i+1];
            }
            values[m-1] = next_val;
            len += 1;
        } else {
            break;
        }
    }

    free(values);
    return len;
```

```
}

/*****************************************************************
 *  See FZcompress.h for function description
 *****************************************************************/
int encode_file(FILE* in, FILE* out, int deg) {
    data_word* series;
    data_word in_word = 0;
    uint8_t out_byte = 0;
    int bit_idx = 0;

    int bytes_read = read_word_bigend(in, &in_word);
    if(bytes_read == 0) {
        printf(MSG_EMPTY_WARN);
        return -1;
    } else if(bytes_read < 0) {
        fprintf(stderr, MSG_READ_ERR);
        return -1;
    }

    // Set up and reserve space for 8 byte header
    fz_header_t header = { 0, 0, 0, 0 };
    if(write_header(out, header) != 0) {
        fprintf(stderr, MSG_FORMAT_ERR);
        return -1;
    }

    //Generate the Fibonacci series of the given degree 'm'
    series = (data_word*)calloc(encode_bits, sizeof(*series));
    if(series == NULL) {
        fprintf(stderr, MSG_MEM_ERR);
        return -1;
    } else {
        generate_series(series, deg, encode_bits);
    }

    // Read in each data byte from the file
    while(bytes_read > 0) {
        // If out of bytes before reading a full word, encode the ↩
           remainder
        if(DEBUG != 0) {
```

```
            printf("Encoding %08x %d with degree %d\n", in_word, ↩
                bytes_read, deg);
        }

        // Encode each word according to its Zeckendorf sum ↩
            notation
        for(int i = 0; i < encode_bits; i += 1) {
            if(in_word >= series[i]) {
                in_word -= series[i];
                set_char_bit(&out_byte, bit_idx, 1);
            }

            // Write encoded bytes to the output file
            bit_idx = (bit_idx + 1) % BYTE_BITS;
            if(bit_idx == 0) {
                if(fwrite(&out_byte, sizeof(out_byte), 1, out) != ↩
                    1) {
                    fprintf(stderr, MSG_MEM_ERR);
                    free(series);
                    return -1;
                }
                out_byte = 0;
            }
        }

        if(bytes_read == sizeof(in_word)) {
            bytes_read = read_word_bigend(in, &in_word);
        } else {
            break;
        }
    }

    free(series);

    // Error reporting and cleanup
    if(bytes_read < 0) {
        fprintf(stderr, MSG_READ_ERR);
        return -1;
    } else {
        // Last byte was partially encoded to, write it out
        if(bit_idx != 0) {
```

```
            if(fwrite(&out_byte, sizeof(out_byte), 1, out) != 1) {
                fprintf(stderr, MSG_WRITE_ERR);
                return -1;
            }
        }

        // Write the header info once the encode is complete
        rewind(out);
        header.magic = MAGIC_WORD;
        header.offset = (sizeof(data_word) == sizeof(uint64_t)) ?
            (0x8000 | bytes_read) : bytes_read;
        if(write_header(out, header) != 0) {
            fprintf(stderr, MSG_FORMAT_ERR);
            return -1;
        }
    }

    return 0;
}

/*****************************************************************
 *  See FZcompress.h for function description
 *****************************************************************/
int decode_file(FILE* in, FILE* out, int deg) {
    data_word* series;
    uint8_t in_byte;
    uint8_t next_byte;
    data_word out_word = 0;
    int series_idx = 0;

    fz_header_t header = { 0, 0, 0, 0 };
    uint8_t offset;

    //Check file header validity
    if(read_header(in, &header) == 0) {
        if(header.magic != MAGIC_WORD) {
            fprintf(stderr, MSG_WRONG_FMT);
            return -1;
        } else if(header.type != 0) {
            fprintf(stderr, MSG_WRONG_ENCD);
            return -1;
```

```
    } else if(((header.offset & 0x80) == 1) && ((header.offset ↩
       & 0x7F) > 3)) {
        fprintf(stderr, MSG_BAD_OFFST);
        return -1;
    } else {
        offset = header.offset & 0x7F;
    }
} else {
    fprintf(stderr, MSG_READ_ERR);
    return -1;
}

//Generate the Fibonacci series of the given degree 'm'
series = (data_word*)calloc(encode_bits, sizeof(*series));
if(series == NULL) {
    fprintf(stderr, MSG_MEM_ERR);
    return -1;
} else {
    generate_series(series, deg, encode_bits);
}

// Check to make sure the file contains data
int have_next = fread(&in_byte, sizeof(in_byte), 1, in);
if((have_next != 1) && feof(in) == 0) {
    fprintf(stderr, MSG_READ_ERR);
    free(series);
    return -1;
}

// Read in each data byte from the file
while(have_next == 1) {
    if(DEBUG != 0) {
        printf("Decoding %02x with degree %d\n", in_byte, deg);
    }

    have_next = fread(&next_byte, sizeof(next_byte), 1, in);
    if((have_next != 1) && feof(in) == 0) {
        fprintf(stderr, MSG_READ_ERR);
        free(series);
        return -1;
    }
```

```
        // Check each bit in the file, add correspondig series ↩
           value if set
        for(int i = 0; i < BYTE_BITS; i += 1) {
            if(get_char_bit(in_byte, i) != 0) {
                out_word += series[series_idx];
            }

            // Write decoded words to the output file
            series_idx = (series_idx + 1) % encode_bits;
            if(series_idx == 0) {
                int to_write;

                if(DEBUG != 0) {
                    printf("Got %08x from decoder\n", out_word);
                }

                if(have_next) { to_write = sizeof(out_word); }
                else          { to_write = offset; }

                if(write_word_bigend(out, out_word, to_write) != ↩
                   to_write) {
                    fprintf(stderr, MSG_WRITE_ERR);
                    free(series);
                    return -1;
                }

                out_word = 0;
            }
        }

        in_byte = next_byte;
    }

    free(series);
    return 0;
}

/*****************************************************************
 *  See FZcompress.h for function description
 *****************************************************************/
```

```
int pad_file(FILE* in, FILE* out, int deg) {
    const int history = (deg/BYTE_BITS) + 2;
    // This byte buffer stores a certain number of bytes such that the bit
    // that is 2*deg-1 positions prior to the current byte can be accessed
    uint8_t* in_bytes = (uint8_t*)calloc(history, sizeof(*in_bytes));
    int null_ct = deg - 1; // Number of consecutive 0's counted
    uint8_t extra_bits = 0;
    uint32_t extra_bytes = 0;

    int series_idx = 0; // Used to keep track of encoding boundaries
    int write_bytes = 0;

    // Read header and check file validity
    fz_header_t header = { 0, 0, 0, 0 };
    if(read_header(in, &header) == 0) {
        if(header.magic != MAGIC_WORD) {
            fprintf(stderr, MSG_WRONG_FMT);
            free(in_bytes);
            return -1;
        } else if(header.type != 0) {
            fprintf(stderr, MSG_WRONG_ENCD);
            free(in_bytes);
            return -1;
        } else if(((header.offset & 0x80) == 1) && ((header.offset & 0x7F) > 3)) {
            fprintf(stderr, MSG_BAD_OFFST);
            free(in_bytes);
            return -1;
        }
    } else {
        fprintf(stderr, MSG_READ_ERR);
        free(in_bytes);
        return -1;
    }

    // "Reserve" header space in output file
    if(write_header(out, header) != 0) {
```

```
    fprintf(stderr, MSG_FORMAT_ERR);
    free(in_bytes);
    return -1;
}

// Read in each data byte from the file
while(fread(&in_bytes[0], sizeof(in_bytes[0]), 1, in) == 1) {
    if(DEBUG == 1) {
        printf("Padding %02x\n", in_bytes[0]);
    }
    // Check each bit in the file character input
    for(int i = 0; i < BYTE_BITS; i += 1) {
        int set_bit = ((i-(deg-1)) % BYTE_BITS);
        int set_byte = ((i-(deg-1)) < 0) ? 1 - (i-(deg-1))/↩
           BYTE_BITS: 0;
        set_bit = (set_bit < 0) ? set_bit + BYTE_BITS : set_bit↩
           ;

        // Count each run of 0-bits, resetting when a 1 is ↩
           found
        if(get_char_bit(in_bytes[0], i) != 0) {
            if((extra_bits += (deg-1)) >= BYTE_BITS) {
                extra_bytes += extra_bits/BYTE_BITS;
                extra_bits %= BYTE_BITS;
            }
            null_ct = 0;
        } else {
            null_ct += 1;
        }

        if(DEBUG != 0) {
            printf("Null ct = %d, last bit was %d at %d of %02x↩
               \n", null_ct,
                get_char_bit(in_bytes[0], i), i, in_bytes[0]);
        }

        // Break any runs of 2*deg - 1 0's by inserting a 1
        if(null_ct >= (2*deg - 1)) {
            set_char_bit(&(in_bytes[set_byte]), set_bit, 1);
            null_ct -= deg;
```

```
            if((extra_bits += (deg-1)) >= BYTE_BITS) {
                extra_bytes += extra_bits/BYTE_BITS;
                extra_bits %= BYTE_BITS;
            }
            if(DEBUG != 0) {
                printf("Set bit at (%d, %d)\n", set_byte, ↩
                    set_bit);
            }
        }


        // Keep track of encoding boundaries; important because↩
            the data
        // sparseness property is not guaranteed BETWEEN ↩
            encoded words, only
        // WITHIN them
        series_idx = (series_idx + 1) % encode_bits;
        if(series_idx == 0) {
            if(null_ct >= deg) {
                set_byte = ((i-(null_ct-deg))/BYTE_BITS);
                set_bit = ((i-(null_ct-deg)) % BYTE_BITS);
                set_byte = (set_byte < 0) ? 1 - set_byte : ↩
                    set_byte;
                set_bit = (set_bit < 0) ? set_bit + BYTE_BITS :↩
                    set_byte;

                set_char_bit(&(in_bytes[set_byte]), set_bit, 1)↩
                    ;
                if((extra_bits += (deg-1)) >= 8) {
                    extra_bytes += extra_bits/BYTE_BITS;
                    extra_bits %= BYTE_BITS;
                }
                if(DEBUG != 0) {
                    printf("Set bit at (%d, %d)\n", set_byte, ↩
                        set_bit);
                }
            }

            null_ct = deg - 1;
        }
    }
```

```
    // Start writing out bytes once the fifo buffer is filled
    if(write_bytes == history-1) {
        if(fwrite(&(in_bytes[write_bytes]), sizeof(in_bytes[0])↩
            , 1, out) != 1) {
            fprintf(stderr, MSG_WRITE_ERR);
            free(in_bytes);
            return -1;
        } else if(DEBUG == 1) {
            printf("Result %02x\n", in_bytes[write_bytes]);
        }
    }
    else {
        write_bytes += 1;
    }

    // Cycle each byte in the byte buffer one index forward
    for(int i = history - 2; i >= 0; i -= 1) {
        in_bytes[i+1] = in_bytes[i];
    }
}

// Check that we reached the end of the file and didn't error ↩
   out
if(feof(in) == 0) {
    fprintf(stderr, MSG_READ_ERR);
    free(in_bytes);
    return -1;
} else {
    // Write out the rest of the buffer
    for(int i = write_bytes; i >= 0; i -= 1) {
        if(fwrite(&(in_bytes[write_bytes]), sizeof(in_bytes[0])↩
            , 1, out) != 1) {
            fprintf(stderr, MSG_WRITE_ERR);
            free(in_bytes);
            return -1;
        } else if(DEBUG == 1) {
            printf("Result %02x\n", in_bytes[write_bytes-1]);
        }
    }
```

```
        free(in_bytes);
    }

    // Update the padded file header with state and number of ↩
       secondary bytes
    rewind(out);
    header.type = 1;
    header.extra_bytes = (uint32_t)extra_bytes;
    if(write_header(out, header) != 0) {
        fprintf(stderr, MSG_FORMAT_ERR);
        return -1;
    }

    return extra_bytes;
}

/*****************************************************************
 *  See FZcompress.h for function description
 *****************************************************************/
int overwrite_file(FILE* in, FILE* out, FILE* over, int deg) {
    uint8_t in_byte;
    uint8_t over_byte;

    uint8_t out_byte = 0;
    int write_ct = 0; // Number of bits left to write
    int over_idx = 0;
    int series_idx = 0; // Used to keep track of encoding ↩
       boundaries
    int early_term = 0; // Terminate if end of 'over' is reached

    // Read in header and check file validity
    fz_header_t header = { 0, 0, 0, 0 };
    if(read_header(in, &header) == 0) {
        if(header.magic != MAGIC_WORD) {
            fprintf(stderr, MSG_WRONG_FMT);
            return -1;
        } else if(header.type != 1) {
            fprintf(stderr, MSG_WRONG_ENCD);
            return -1;
        } else if(((header.offset & 0x80) == 1) && ((header.offset ↩
           & 0x7F) > 3)) {
```

```
            fprintf(stderr, MSG_BAD_OFFST);
            return -1;
        } else if(header.extra_bytes == 0) {
            // Very simple check, but it's the best we can do ↩
               without scanning
            // to the end of the 'over' file to check that the ↩
               number of bytes
            // therein is less than or equal to the number of ↩
               secondary bytes
            // in the input file
            fprintf(stderr, "No secondary storage bytes available ↩
               in input, "
                "exiting\n");
            return -1;
        }
    } else {
        fprintf(stderr, MSG_READ_ERR);
        return -1;
    }

    // "Reserve" header space in output file
    if(write_header(out, header) != 0) {
        fprintf(stderr, MSG_FORMAT_ERR);
        return -1;
    }

    // Read in each data byte from the file
    while((fread(&in_byte, sizeof(in_byte), 1, in) == 1) && (↩
       early_term == 0)) {
        out_byte = in_byte;
        if(DEBUG != 0) {
            printf("Overwriting %02x\n", in_byte);
        }
        // Write 'deg' - 1 bits from 'over' after each 1-bit from '↩
           in'
        for(int i = 0; i < BYTE_BITS; i += 1) {
            // Write 'over' data to 'out_byte' if write is enabled;↩
                otherwise,
            // keep scanning input data for 1-bits
            if(write_ct > 0) {
                if(over_idx == 0) {
```

```
            // Attempt to read in a new data byte from 'over'; depending
            // on the relative size of 'in' and 'over' and the number
            // of bits available for writing, we may reach the end of
            // one file before the other, so just return if we cannot
            // read more data
            if(fread(&over_byte, sizeof(over_byte), 1, over) != 1) {
                printf("Finished encoding secondary data, exiting\n");
                if(feof(over) == 0) {
                    fprintf(stderr, MSG_READ_ERR);
                    return -1;
                }

                early_term = 1;
                break;
            }
        }

        set_char_bit(&out_byte, i, get_char_bit(over_byte, over_idx));
        write_ct -= 1;
        over_idx = (over_idx + 1) % BYTE_BITS;
        if(DEBUG != 0) {
            printf("Processed data byte at idx %d\n", over_idx);
        }
    } else if(get_char_bit(in_byte, i) != 0) {
        write_ct = deg - 1;
    }

    // Stop writing once an encoding boundary is reached
    series_idx = (series_idx + 1) % encode_bits;
    if(series_idx == 0) {
        write_ct = 0;
    }
}
```

```
            // Write the overwritten data to output file
            if(fwrite(&out_byte, sizeof(out_byte), 1, out) != 1) {
                fprintf(stderr, MSG_WRITE_ERR);
                return -1;
            }
        }

        if((feof(in) != 0) && (early_term == 0)) {
            fprintf(stderr, MSG_READ_ERR);
            return -1;
        }

        rewind(out);
        header.type = 2;
        if(write_header(out, header) != 0) {
            fprintf(stderr, MSG_FORMAT_ERR);
            return -1;
        }

        // Check to see if we ran out of 'in' words before we encoded ↩
           everything
        if(feof(over) == 0) {
            printf("Warning, unable to complete secondary encoding, ↩
               input encoded "
                "file has insufficient secondary storage capacity\n");
            return -1;
        }

        return 0;
    }

    /*****************************************************************
     *  See FZcompress.h for function description
     *****************************************************************/
    int extract_file(FILE* in, FILE* out, int deg) {
        uint8_t in_byte;

        uint8_t out_byte = 0;
        int read_ct = 0; // Number of bits left to read
        int out_idx = 0;
```

```
int series_idx = 0; // Used to keep track of encoding ↩
    boundaries

// Read in header and check file validity
fz_header_t header = { 0, 0, 0, 0 };
if(read_header(in, &header) == 0) {
    if(header.magic != MAGIC_WORD) {
        fprintf(stderr, MSG_WRONG_FMT);
        return -1;
    } else if(header.type != 2) {
        fprintf(stderr, MSG_WRONG_ENCD);
        return -1;
    } else if(((header.offset & 0x80) == 1) && ((header.offset ↩
       & 0x7F) > 3)) {
        fprintf(stderr, MSG_BAD_OFFST);
        return -1;
    } else if(header.extra_bytes == 0) {
        // Sanity check to make sure input file actually has ↩
           bytes to read
        fprintf(stderr, "No secondary storage bytes available ↩
           in input, "
            "exiting\n");
        return -1;
    }
} else {
    fprintf(stderr, MSG_READ_ERR);
    return -1;
}

// Read in each data byte from the file
while(fread(&in_byte, sizeof(in_byte), 1, in) == 1) {
    if(DEBUG != 0) {
        printf("Extracting %02x\n", in_byte);
    }
    // Read 'deg' - 1 bits after each 1-bit from 'in'
    for(int i = 0; i < BYTE_BITS; i += 1) {
        // Read 'in' data to 'out_byte' if write is enabled; ↩
           otherwise,
        // keep scanning input data for 1-bits
        if(read_ct > 0) {
```

```
                set_char_bit(&out_byte, out_idx, get_char_bit(↩
                    in_byte, i));
                read_ct -= 1;
                out_idx = (out_idx + 1) % BYTE_BITS;

                // Write to output once a byte has been completed
                if(out_idx == 0) {
                    if(fwrite(&out_byte, sizeof(out_byte), 1, out) ↩
                        != 1) {
                        fprintf(stderr, MSG_WRITE_ERR);
                        return -1;
                    }
                }
            } else if(get_char_bit(in_byte, i) != 0) {
                read_ct = deg - 1;
            }

            // Stop reading once an encoding boundary is reached
            series_idx = (series_idx + 1) % encode_bits;
            if(series_idx == 0) {
                read_ct = 0;
            }
        }
    }

    return 0;
}

/*****************************************************************
 * Main function for testing purposes and application use
 *****************************************************************/
int main(int argc, char** argv) {
    char* op;
    int   deg;
    char* fname_in;
    char* fname_out;
    char* fname_over;
    FILE* in_file;
    FILE* out_file;
    FILE* over_file;
    int   err = 0;
```

```
if(argc < 5 || argc > 6) {
    printf("Usage: compress {encode, decode, pad, overwrite, ↩
       extract} DEG IN OUT [OVER]\n");
    printf("Applies the requested rewriting code operation on ↩
       data from IN using an encoding\n");
    printf("degree of DEG and writes the result to the file ↩
       name provided by OUT.\n");
    printf("\n");
    printf("    DEG    the primary encoding degree to be ↩
       applied for 'encode' operations,\n");
    printf("           or the degree of the previously encoded ↩
       data for all other oeprations\n");
    printf("    IN     the name of the input file to encode/ ↩
       decode\n");
    printf("    OUT    the desired name of the output file to ↩
       be used\n");
    printf("    OVER   the name of the secondary input file, ↩
       used to obtain data from to\n");
    printf("           \"overwrite\" into the main input file ↩
       for 'overwrite' operations\n");
}
else {
    // Read in args
    op = (char*)argv[1];
    deg = atoi(argv[2]);
    fname_in = (char*)argv[3];
    fname_out = (char*)argv[4];
    encode_bits = get_series_len(deg, WORD_BITS);

    if(strcmp(op, FUNC_OVR) == 0) {
        fname_over = argv[5];
    } else {
        fname_over = NULL;
    }

    if(deg < 2 || deg > 10) {
        printf("Unsupported encoding degree, degree "
            "should be in range [2, 10]\n");
        exit(0);
    }
```

```
// Open input/ output files
in_file = fopen(fname_in, "r");
if(in_file == NULL) {
    printf("Unable to open input file \"%s\", err %m\n", ↩
       fname_in);
    exit(-1);
}

out_file = fopen(fname_out, "w");
if(out_file == NULL) {
    printf("Unable to open output file \"%s\", err %m\n", ↩
       fname_out);
    exit(-1);
}

if(fname_over != NULL) {
    over_file = fopen(fname_over, "r");
    if(over_file == NULL) {
        printf("Unable to open over file \"%s\", err %m\n",↩
            fname_over);
        exit(-1);
    }
} else {
    over_file = NULL;
}

printf("Applying \'%s\' to file %s\n", op, fname_in);
printf("Encoding unit size is: %luB\n", WORD_SIZE);
printf("Encoded word size for degree %d: %db\n", deg, ↩
   encode_bits);

// Perform specified operation
if(strcmp(op, FUNC_ENC) == 0) {
    err = encode_file(in_file, out_file, deg);
} else if(strcmp(op, FUNC_DEC) == 0) {
    err = decode_file(in_file, out_file, deg);
} else if(strcmp(op, FUNC_PAD) == 0) {
    err = pad_file(in_file, out_file, deg);
    if(err > 0) {
```

```
            long file_bytes = ftell(in_file) - sizeof(↩
                fz_header_t);
            long data_bytes = (file_bytes)*((double)WORD_BITS/↩
                encode_bits);
            printf("Total number of bytes available for ↩
                secondary storage "
                 "in padded file %s: %d\n", fname_out, err);
            printf("Contains %ld primary + %d secondary data ↩
                bytes in %ld storage "
                 "bytes, overall compression is %f\n", ↩
                    data_bytes, err,
                    file_bytes, (((double)data_bytes/file_bytes)↩
                        + ((double)err/file_bytes)));
        }
    } else if(strcmp(op, FUNC_OVR) == 0) {
        err = overwrite_file(in_file, out_file, over_file, deg)↩
            ;
    } else if(strcmp(op, FUNC_EXT) == 0) {
        err = extract_file(in_file, out_file, deg);
    } else {
        fprintf(stderr, "Error: invalid operation \'%s\'\n", op↩
            );
        exit(-1);
    }

    printf("\nFinished encoding, closing files...\n");
    if(in_file != NULL) {
        err |= fclose(in_file);
    }
    if(out_file != NULL) {
        err |= fclose(out_file);
    }
    if(over_file != NULL) {
        err |= fclose(over_file);
    }

    if(err < 0) {
        printf("Error occurred during execution, output may be ↩
            empty or "
             "invalid\n");
    }
```

```
        printf("Done!\n");
        exit(err);
    }

    return 1;
}
```

A.2 Hardware Implementation

A.2.1 FZCompressSeq.sv

```
module FZCompressSeq #(
  parameter M_VAL = 2,
  parameter ENCODE_UNIT = 32,
  parameter SERIES_LEN = 46
)
(
  input logic clock,
  input logic reset,
  input logic [3:0]mode,
  input logic [ENCODE_UNIT-1:0]data_in,     //Unencoded data words ↩
     from PC memory
  input logic data_rdy,             //Signal to active modules that↩
      input data is ready on PC data bus
  input logic [SERIES_LEN-1:0]encode_in,  //Encoded data words from↩
      flash memory
  input logic encode_rdy,             //Signal to active modules ↩
     that input data is ready on flash data bus
  output logic [SERIES_LEN-1:0]encode_out,  //Encode/ pad/ ↩
     overwrite output
  output logic encode_complete,         //Indicates that an encoded↩
      word has been fully calculated
  output logic [ENCODE_UNIT-1:0]data_out,   //Decode/ extract ↩
     output
  output logic data_complete, //Indicates that a data word has been↩
      fully decoded
  output logic data_in_req,   //Indicates one or more modules ↩
     require additional PC data
```

```
  output logic encode_in_req    //Indicates one or more modules ↩
     require additional flash data
);

  logic [ENCODE_UNIT-1:0]data_bus_out[1:0];
  logic [SERIES_LEN-1:0]encode_bus_out[2:0];
  logic [4:0]chip_sel;
  logic [4:0]sig_out; //Output readyness signals

  always_comb begin
    case(mode)
      4'd0:  chip_sel = 5'b00000;
      4'd1:  chip_sel = 5'b00001;
      4'd2:  chip_sel = 5'b00010;
      4'd3:  chip_sel = 5'b00100;
      4'd4:  chip_sel = 5'b01000;
      4'd5:  chip_sel = 5'b10000;
      4'd6:  chip_sel = 5'b00011;
      4'd7:  chip_sel = 5'b10001;
      //Resolve potential collisions on input busses by disabling ↩
         one module until
      //the other has received its data
      4'd8:  chip_sel = (sig_out[1]&sig_out[2]) ? 5'b00010 : 5'↩
         b00110;
      4'd9:  chip_sel = (sig_out[2]&data_in_req)  ? 5'b00100 : 5'↩
         b10100;
      4'd10: chip_sel = (sig_out[1]&sig_out[3]) ? 5'b01000 : 5'↩
         b01010;
      4'd11: chip_sel = (sig_out[3]&data_in_req)  ? 5'b10000 : 5'↩
         b11000;
      default: chip_sel = 5'b00000;
    endcase
  end

  //Encode Round 1 - PC => Mem
  FZEncodeWordSeq #(M_VAL, ENCODE_UNIT, SERIES_LEN) encoder(
              .clock(clock),
              .enable(chip_sel[0]),
              .reset(chip_sel[0] & reset),//Only reset when enabled
              .in(data_in),         //PC data to encode
```

```
            .in_ready(data_rdy),      //Signal to module that ↩
               data_in is now valid
            .out(encode_bus_out[0]),  //Bus to flash memory ↩
               output buffer
            .out_ready(sig_out[0]));  //Circuit output ready ↩
               signal

//Decode Round 1 - Mem => PC
FZDecodeWordSeq #(M_VAL, ENCODE_UNIT, SERIES_LEN) decoder(
            .clock(clock),
            .enable(chip_sel[1]),
            .reset(chip_sel[1] & reset),//Only reset when enabled
            .in(encode_in),       //Flash data to decode
            .in_ready(encode_rdy),    //Signal to module that ↩
               encode_in is now valid
            .out(data_bus_out[0]),    //Bus to PC output buffer
            .out_ready(sig_out[1]));  //Circuit output ready ↩
               signal

//Pad bits - Mem => Mem
FZPadBitsSeq #(M_VAL, ENCODE_UNIT, SERIES_LEN) padder(
            .clock(clock),
            .enable(chip_sel[2]),
            .reset(chip_sel[2] & reset),//Only reset when enabled
            .in(encode_in),       //Flash data to pad
            .in_ready(encode_rdy),    //Signal to module that ↩
               encode_in is now valid
            .out(encode_bus_out[1]),  //Bus to flash memory ↩
               output buffer
            .out_ready(sig_out[2]));  //Circuit output ready ↩
               signal

//Overwrite/ Encode Round 2 - PC + Mem => Mem
FZOverwriteSeq #(M_VAL, ENCODE_UNIT, SERIES_LEN) overwriter(
            .clock(clock),
            .enable(chip_sel[3]),
            .reset(chip_sel[3] & reset),//Only reset when enabled
            .in(encode_in),       //Flash data to encode/ ↩
               overwrite
            .in_ready(encode_rdy),    //Signal to module that ↩
               encode_in is now valid
```

```
            .over(data_in),       //PC data to encode into ↩
               fireset input
            .over_ready(data_rdy),    //Signal to module that ↩
               data_in is now valid
            .out(encode_bus_out[2]),  //Bus to flash memory ↩
               output buffer
            .in_request(data_in_req), //Request for additional PC↩
                data if input is exhausted
            .out_ready(sig_out[3]));  //Circuit output ready ↩
               signal

//Extract/ Decode Round 2 - Mem => PC
FZExtractSeq #(M_VAL, ENCODE_UNIT, SERIES_LEN) extractor(
            .clock(clock),
            .enable(chip_sel[4]),
            .reset(chip_sel[4] & reset),//Only reset when enabled
            .in(encode_in),       //Flash data to decode/ extract
            .in_ready(encode_rdy),    //Signal to module that ↩
               encode_in is now valid
            .out(data_bus_out[1]),    //Bus to PC output buffer
            .in_request(encode_in_req),//Request for additional ↩
               flash memory data if input is exhausted
            .out_ready(sig_out[4]));  //Circuit output ready ↩
               signal

//chip_sel[0, 2, 3] are mutually exclusive, so currently selected↩
    encode circuit is effectively muxed to output port
assign encode_out = (encode_bus_out[0] & {(SERIES_LEN){chip_sel↩
   [0]}})
            | (encode_bus_out[1] & {(SERIES_LEN){chip_sel[2]}})
            | (encode_bus_out[2] & {(SERIES_LEN){chip_sel[3]}});
assign encode_complete = (sig_out[0] & chip_sel[0]) | (sig_out[2]↩
    & chip_sel[2]) | (sig_out[3] & chip_sel[3]);

//Again, chip_sel[1, 4] are mutually exclusive
assign data_out = (data_bus_out[0] & {(ENCODE_UNIT){chip_sel↩
   [1]}})
          | (data_bus_out[1] & {(ENCODE_UNIT){chip_sel[4]}});
assign data_complete = (sig_out[1] & chip_sel[1]) | (sig_out[4] &↩
    chip_sel[4]);
```

```
endmodule
```

A.2.2 FZEncodeWordSeq.sv

```
module FZEncodeWordSeq #(
  parameter M_VAL = 2,
  parameter ENCODE_UNIT = 32,
  parameter SERIES_LEN = 46
)
(
  input clock,
  input enable,
  input reset,
  input [ENCODE_UNIT-1:0]in,
  input in_ready,
  output logic [SERIES_LEN-1:0]out,
  output logic out_ready
);

  logic [ENCODE_UNIT-1:0]series[SERIES_LEN];
  int i;

  logic [ENCODE_UNIT-1:0]remainder;
  logic [$clog2(SERIES_LEN)-1:0]series_idx;
  logic run_lock;

  //Initialize logic and calculate series for simulation
  initial begin
    out = '0;
    out_ready = 1'b0;

    remainder = '0;
    series_idx = '0;
    run_lock = 1'b0;

    for(i = 0; i < SERIES_LEN; i += 1) begin
      if(i < M_VAL) begin
        series[i] = (i+1);
      end else begin
        series[i] = series[i-1] + series[i-M_VAL];
```

```
      end
    end
  end

  always@(posedge clock, negedge reset) begin
    //Asynchronous reset
    if(!reset) begin
      out <= '0;
      out_ready <= 1'b0;

      remainder <= '0;
      series_idx <= '{(SERIES_LEN - 1)};
      run_lock <= 1'b0;
    end else if(enable) begin
      //Begin new encoding session if no session in progress
      if(!run_lock && in_ready) begin
        out <= '0;
        out_ready <= 1'b0;

        remainder <= in;
        series_idx <= '{(SERIES_LEN - 1)};
        run_lock <= 1'b1;
      end else if(run_lock) begin
        //Continue encoding if already in progress
        if(remainder >= series[series_idx]) begin
          remainder <= remainder - series[series_idx];
          out[series_idx] <= 1'b1;
        end

        if(series_idx > 0) begin
          series_idx <= series_idx - 1'b1;
        end else begin
          out_ready <= 1'b1;
          run_lock <= 1'b0;
        end
      end
    end
  end

endmodule
```

A.2.3 FZDecodeWordSeq.sv

```
module FZDecodeWordSeq #(
  parameter M_VAL = 2,
  parameter ENCODE_UNIT = 32,
  parameter SERIES_LEN = 46
)
(
  input clock,
  input enable,
  input reset,
  input [SERIES_LEN-1:0]in,
  input in_ready,
  output reg [ENCODE_UNIT-1:0]out,
  output reg out_ready
);

  logic [ENCODE_UNIT-1:0]series[SERIES_LEN];
  int i;

  logic [SERIES_LEN-1:0]encoding;
  logic [$clog2(SERIES_LEN)-1:0]series_idx;
  reg run_lock;

  //Initialize logic and calculate series for simulation
  initial begin
    out = '0;
    out_ready = 1'b0;

    encoding = '0;
    series_idx = '0;
    run_lock = 1'b0;

    for(i = 0; i < SERIES_LEN; i += 1) begin
      if(i < M_VAL) begin
        series[i] = (i+1);
      end else begin
        series[i] = series[i-1] + series[i-M_VAL];
      end
    end
  end
```

```
  always@(posedge clock, negedge reset) begin
    //Asynchronous reset
    if(!reset) begin
      out <= {(ENCODE_UNIT){1'b0}};
      out_ready <= 1'b0;

      encoding <= {(SERIES_LEN){1'b0}};
      series_idx <= '{(SERIES_LEN - 1)};
      run_lock <= 1'b0;
    end else if(enable) begin
      if(!run_lock && in_ready) begin
        //Begin new decoding session if no session in progress
        out <= {(ENCODE_UNIT){1'b0}};
        out_ready <= 1'b0;

        encoding <= in;
        series_idx <= '{(SERIES_LEN - 1)};
        run_lock <= 1'b1;
      end else if(run_lock) begin
        //Continue decoding if already in progress
        if(encoding[series_idx] == 1'b1) begin
          out <= out + series[series_idx];
        end

        if(series_idx > 0) begin
          series_idx <= series_idx - 1'b1;
        end else begin
          out_ready <= 1'b1;
          run_lock <= 1'b0;
        end
      end
    end
  end

endmodule
```

A.2.4 FZPadWordSeq.sv

```
module FZPadBitsSeq #(
```

```
  parameter M_VAL = 2,
  parameter ENCODE_UNIT = 32,
  parameter SERIES_LEN = 46
)
(
  input clock,
  input enable,
  input reset,
  input [SERIES_LEN-1:0]in,
  input in_ready,
  output logic [SERIES_LEN-1:0]out,
  output logic out_ready
);

  logic [$clog2(SERIES_LEN)-1:0]out_idx;
  logic [$clog2(2*M_VAL)-1:0]null_ct; //Number of consecutive 0s ↩
     found
  logic run_lock;

  //Initialize logic for simulation
  initial begin
    out = '0;
    out_ready = 1'b0;

    out_idx = '0;
    null_ct = '0;
    run_lock = 1'b0;
  end

  always@(posedge clock, negedge reset) begin
    //Asynchronous reset
    if(!reset) begin
      out <= '0;
      out_ready <= 1'b0;

      out_idx <= '0;
      null_ct <= '0;
      run_lock <= 1'b0;
    end else if(enable) begin
      if(!run_lock && in_ready) begin
        //Begin new padding session if no session in progress
```

```
        out <= in;
        out_ready <= 1'b0;

        out_idx <= '{(SERIES_LEN - 1)};
        null_ct <= '{(M_VAL - 1)};
        run_lock <= 1'b1;
      end else if(run_lock) begin
        //Continue padding if already in progress
        if(out[out_idx] == 0) begin
          if(null_ct >= M_VAL*2 - 1) begin
            out[out_idx + M_VAL] <= 1'b1;
            null_ct <= null_ct - {'{(M_VAL - 1)}};
          end else if((out_idx == 0) && (null_ct >= (M_VAL - 1))) ↩
             begin
            out[null_ct - (M_VAL - 1)] <= 1'b1;
          end else begin
            null_ct <= null_ct + 1'b1;
          end
        end else begin
          if(null_ct >= M_VAL*2 - 1) begin
            out[out_idx + M_VAL] <= 1'b1;
          end

          null_ct <= '0;
        end

        if(out_idx > 0) begin
          out_idx <= out_idx - 1'b1;
        end else begin
          out_ready <= 1'b1;
          run_lock <= 1'b0;
        end
      end
    end
  end

endmodule
```

A.2.5 FZOverwriteSeq.sv

```
module FZOverwriteSeq #(
  parameter M_VAL = 2,
  parameter ENCODE_UNIT = 32,
  parameter SERIES_LEN = 46
)
(
  input clock,
  input enable,
  input reset,
  input [SERIES_LEN-1:0]in,      //Input encoding from Flash Mem
  input in_ready,
  input [ENCODE_UNIT-1:0]over,  //Input data to store from PC
  input over_ready,
  output logic [SERIES_LEN-1:0]out,
  output logic in_request,    //Allows circuit to request ↩
     additional PC data
  output logic out_ready
);

  logic [ENCODE_UNIT-1:0]data;
  logic [$clog2(SERIES_LEN)-1:0]out_idx;    //Output index, scans ↩
     and writes to data on output reg from 'in'
  logic [$clog2(ENCODE_UNIT)-1:0]data_idx;  //Data index, reads ↩
     data from input 'over' on data reg
  logic [$clog2(M_VAL)-1:0]write_enable;    //Enables (over)writing↩
      of data to 'out' when greater than 0
  logic run_lock;

  //Initialize logic for simulation
  initial begin
    out = '0;
    in_request = 1'b0;
    out_ready = 1'b0;

    data = '0;
    out_idx = '0;
    data_idx = '0;
    write_enable = '0;
    run_lock = 1'b0;
  end
```

```
always@(posedge clock, negedge reset) begin
  //Asynchronous reset
  if(!reset) begin
    out <= '0;
    in_request <= 1'b1;
    out_ready <= 1'b0;

    data <= '0;
    out_idx <= '0;
    data_idx <= '0;
    write_enable <= '0;
    run_lock <= 1'b0;
  end else if(enable) begin
    if(!run_lock && in_ready) begin
      //Begin new overwriting session if no session in progress
      out <= in;
      out_ready <= 1'b0;

      //Latch in data to encode if requested and ready
      if(in_request && over_ready) begin
        in_request <= 1'b0;

        data <= over;
        data_idx <= '{(ENCODE_UNIT-1)};
      end

      out_idx <= '{(SERIES_LEN - 1)};
      write_enable <= '0;
      run_lock <= 1'b1;
    end else if(in_request) begin
      //Block circuit operation until new data is received
      if(over_ready) begin
        in_request <= 1'b0;

        data <= over;
        data_idx <= '{(ENCODE_UNIT-1)};
      end
    end else if(run_lock) begin
      //Continue overwriting if already in progress
      if(write_enable > 0) begin
        out[out_idx] <= data[data_idx];
```

```
            write_enable <= write_enable - 1'b1;

            //Check if out of data to write
            if(data_idx > 0) begin
              data_idx <= data_idx - 1'b1;
            end else begin
              in_request <= 1'b1;
            end
          end else if(out[out_idx] == 1) begin
            //Write M-1 bits after each separator '1' bit
            write_enable <= '{(M_VAL - 1)};
          end

          //Check if out of primary data to write to
          if(out_idx > 0) begin
            out_idx <= out_idx - 1'b1;
          end else begin
            out_ready <= 1'b1;
            run_lock <= 1'b0;
          end
        end
      end
    end

endmodule
```

A.2.6 FZExtractSeq.sv

```
module FZExtractSeq #(
  parameter M_VAL = 2,
  parameter ENCODE_UNIT = 32,
  parameter SERIES_LEN = 46
)
(
  input clock,
  input enable,
  input reset,
  input [SERIES_LEN-1:0]in, //Input to decode from flash storage
  input in_ready,
  output logic [ENCODE_UNIT-1:0]out,
```

```
  output logic in_request, //Request next encoding once 'in' is ↩
     exhausted
  output logic out_ready
);

  logic [SERIES_LEN-1:0]encoding; //Used to latch input word from '↩
     in'
  logic [$clog2(SERIES_LEN):0]enc_idx;
  logic [$clog2(ENCODE_UNIT):0]out_idx;
  logic [$clog2(M_VAL):0]read_enable;
  logic run_lock;

  //Initialize logic for simulation
  initial begin
    out = '0;
    in_request = 1'b0;
    out_ready = 1'b0;

    encoding = '0;
    enc_idx = '0;
    out_idx = '0;
    read_enable = '0;
    run_lock = 1'b0;
  end

  always@(posedge clock, negedge reset) begin
    //Asynchronous reset
    if(!reset) begin
      out <= '0;
      in_request <= 1'b1;
      out_ready <= 1'b0;

      encoding <= '0;
      enc_idx <= '0;
      out_idx <= '{(ENCODE_UNIT-1)};
      read_enable <= '0;
      run_lock <= 1'b0;
    end else if(enable) begin
      if(!run_lock && in_ready) begin
        //Begin new extracting session if no session in progress
        if(in_request) begin
```

```
            in_request <= 1'b0;

            encoding <= in;
            enc_idx <= '{(SERIES_LEN - 1)};
            read_enable <= '0;
          end

          out_ready <= 1'b0;
          run_lock <= 1'b1;
        end else if(run_lock) begin
          //Continue extracting if already in progress
          if(read_enable > 0) begin
            out[out_idx] <= encoding[enc_idx];
            read_enable <= read_enable - 1'b1;

            //Check if out register full
            if(out_idx > 0) begin
              out_idx <= out_idx - 1'b1;
            end else begin
              out_ready <= 1'b1;
              out_idx <= '{(ENCODE_UNIT-1)};
              run_lock <= 1'b0;
            end
          end else if(encoding[enc_idx] == 1'b1) begin
            //Read M-1 bits after each separator '1' bit
            read_enable <= '{(M_VAL - 1)};
          end

          //Check if out of encoded data to extract from
          if(enc_idx > 0) begin
            enc_idx <= enc_idx - 1'b1;
          end else begin
            in_request <= 1'b1;
            run_lock <= 1'b0;
          end
        end
      end
    end

endmodule
```