TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Development of LZ4 Decompression Algorithm in Reconfigurable Computing

Author:
Georgios Galatianos

Thesis Committee:
Sotirios IOANNIDIS
Dionisios PNEVMATIKATOS
Vasilis SAMOLADAS

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

March 14, 2024

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Development of LZ4 Decompression Algorithm in Reconfigurable Computing

by Georgios GALATIANOS

As we move on to the age of information the need for data compression tends to be vital and pivotal. Not only from the perspective of storage but equally important from the perspective of data transfers and transactions. As we enter a period marked by steadily climbing energy prices, a serious issue emerges on how we can lower the cost of our power consumption but also how to lower our power consumption regarding the needs for green transition due to the climate change. In today's fast-paced world, where both accuracy and speed are essential, the importance of doing things in parallel has never been greater. All these matters can be emerged and resolved in this project. We tried to use a state-of-the-art lossless compression algorithm such as LZ4 and combine it with the speed and the power benefits that the FPGAs can give us. Our goal is to reach the point of a fast decompression with zero losses while in the meantime using the minimum physical resources we can.

Acknowledgements

I would like to thank my mentor Dimitris Theodoropoulos who guided me through the whole process and trusted me. Also a huge thank you to the CEO of Cast Inc. Nikos Zervas who offered his help and guidance through the whole process, who trusted me and counted on me in this difficult timing. Furthermore I am thankful for my friends Stefanos, Ilias, Spyros that we shared some amazing memories and pushed ourselves and helped each other, they played a significant role in today's result. Finally the most important thank you is to my parents that without them nothing from all that would be possible. They offered me everything and a lot of trust and freedom throughout this journey.

Contents

List of Figures

List of Tables

List of Abbreviations

ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BRAM	**B**lock **R**andom **A**ccess **M**emory
CPR	**C**ompatible **P**ublic **R**esults
CPU	**C**entral **P**rocessor **U**nit
CS	**C**omputer **S**cience
DDR4	**D**ouble **D**ata **R**ate type texbf4 memory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
FF	**F**lip **F**lops
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GDDR6	**G**raphics **D**ouble **D**ata **R**ate type **6** memory
GPU	**G**raphic **P**rocessor **U**nit
HBM	**H**igh **B**andwidth **M**emory
HDL	**H**ardware **D**escription **L**anguage
HPC	**H**ight **P**erformance **C**omputing
LUT	**L**ook **U**p **T**able
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
PL	**P**rogrammable **L**ogic
PR	**P**ercentile **R**ank
PS	**P**rocessing **S**ystem
RAM	**R**andom **A**ccess **M**emory
SDK	**S**oftware **D**evelopment **K**it
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SSD	**S**olid **S**tate **D**rive
TDP	**T**hermal **D**esign **P**ower
URAM	**U**ltra **R**andom **A**ccess **M**emory

Dedicated to my family and friends…

Chapter 1

Introduction

1.1 Motivation

Data compression, a pivotal technique in managing the exponential growth of digital information, has become increasingly critical in the digital era. It involves reducing the size of data files, enabling more efficient storage, faster transmission, and cost-effective processing. The evolution of hardware development plays a crucial role in advancing data compression techniques, offering the computational power and specialized capabilities required to implement sophisticated algorithms effectively.

As hardware technology progresses, new architectures and processing units are designed to handle the complexities of modern compression algorithms with greater efficiency. These advancements facilitate not only the compression of large datasets but also ensure minimal loss of information, maintaining data integrity. The development of specialized processors and accelerators, such as GPUs and FPGAs, has been instrumental in this context, offering parallel processing capabilities that significantly speed up compression tasks.

Moreover, hardware innovations have expanded the applicability of compression techniques across various fields, including telecommunications, where they enhance bandwidth utilization; in cloud computing, by optimizing resource allocation and reducing storage costs; and in multimedia streaming, improving the user experience through faster loading times and lower buffering incidences.

In summary, the symbiosis between data compression and hardware development is reshaping the landscape of digital information management. As hardware technology continues to evolve, it promises to unlock new potentials in data compression, driving efficiencies and innovation across multiple domains of the digital world.

This work focused on the design and implementation of a hardware IP for data decompression using the LZ4 algorithm.

1.2 Scientific Contributions and Goals

The objective of this project is to develop an LZ4 decompressor optimized for Field-Programmable Gate Arrays (FPGAs), leveraging the unique capabilities of these devices to enhance data decompression processes. The motivation behind this endeavor stems from the critical role of data compression in contemporary technology landscapes, where the efficient storage and transmission of data are paramount. Understanding the fundamentals of data compression—and specifically the mechanisms of the LZ4 algorithm—is essential for the successful implementation of a decompressor. This foundation not only informs the technical approach but also aligns with the broader necessity for advanced compression techniques in the digital era.

FPGAs offer a versatile platform for hardware acceleration, providing the flexibility to implement specialized algorithms with optimized performance characteristics. The decision to utilize FPGAs in this project is driven by their ability to execute parallel processing tasks, a feature that is particularly beneficial for the decompression of data. By designing a decompressor on an FPGA, the project aims to achieve high throughput and low latency in decompression operations, qualities that are increasingly demanded as data volumes continue to expand.

The project's methodology involved translating the principles of LZ4 decompression into a hardware description language (HDL), with VHDL selected for its robustness and widespread support in the field of digital circuit design. This translation required a deep dive into the algorithm's logic to ensure that the resulting hardware implementation would faithfully reproduce the decompression process while maximizing the performance benefits offered by FPGAs.

Some contributions that this work targets:

- Architecture design for data decompression based on LZ4 algorithm
- Implementation of the architecture in reconfigurable computing
- Full test and validation of the implementation

Through this work, the project seeks to bridge the gap between the theoretical aspects of data compression and the practical advantages of hardware accelerators like FPGAs. By doing so, it contributes to the evolving landscape of digital technology, where the ability to efficiently manage data is not just beneficial but necessary. This implementation of an LZ4 decompressor on an FPGA stands as a testament to the synergy between compression algorithms and hardware innovation, aiming to meet the demands of an increasingly data-driven world.

1.3 Thesis Outline

- **Chapter 2 - Theoretical Background:** In this chapter all the theoretical background needed to understand this project will be explained. From the base of the LZ4 algorithm, to the details of an LZ4 file and how it is created. Finally a general idea of how this work can be utilized will be described.
- **Chapter 3 - Related Work:** In this chapter various works from many areas will be collected and introduced.
- **Chapter 4 - System Architecture:** In this chapter the architecture of the project is being discussed in a more abstract way and in a higher level.
- **Chapter 5 - FPGA Implementation and architecture design:** In this chapter all the steps of the work will be analyzed. Both the hardware selection and the parts that will be used and the way that the data will be interpreted.
- **Chapter 6 - Results:** In this chapter the final results of the project will be presented. Also the procedure that was followed to validate the correctnes of the work.
- **Chapter 7 - Conclusions and Related Work:** In this chapter there will be a conclusion and the final view of the work also with our ideas of the next steps of this work and how it can be further developed.

Chapter 2

Theoretical Background

In this chapter the theoretical background and the LZ4 algorithm will be explained and analyzed. Additionally, the LZ4 file will be detailed with all the parts that will provide the necessary base for the project. Finally, the abstract design and the components that will be used are described and why this selection is made.

2.1 LZ4 algorithm

The LZ4 compression algorithm stands as a significant advancement in the field of data compression, noted for its exceptional speed while providing a respectable compression ratio. Developed by Yann Collet[1], LZ4 is designed for fast compression and decompression speeds, making it particularly suitable for real-time applications where processing time is critical. This analysis delves into the principles, architecture, and operational mechanisms of the LZ4 algorithm, providing a comprehensive understanding of its functionality within the realm of compression and decompression processes.

2.1.1 Principles and Architecture

LZ4 operates on the principle of finding short matches, called sequences, between the current data segment being compressed and the previously processed data. It employs a sliding window technique, which allows it to reference back to data that has already been compressed within a certain distance. This approach is instrumental in achieving compression by replacing repeated occurrences of data with references to their first occurrence. The core of LZ4's architecture is its efficient encoding of literals (unmatched data) and match sequences. The algorithm uses a token mechanism to represent the length of literals and sequences, optimizing the space required to store this information.

2.1.2 Compression Process

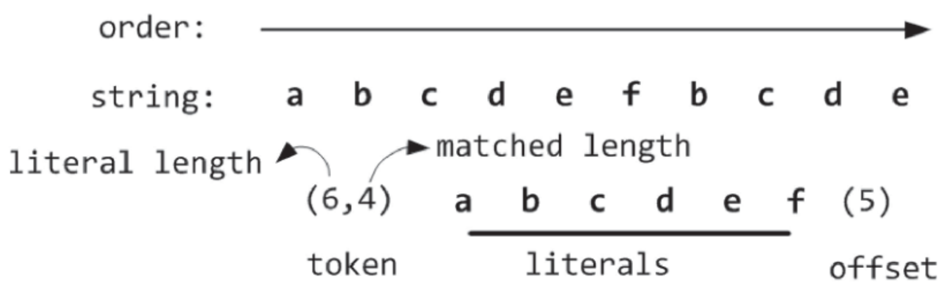

FIGURE 2.1: LZ4 compression algorithm

During compression, LZ4 scans the input data for sequences that have previously appeared within the sliding window. When a match is found, it is encoded as a reference to the position of its first occurrence rather than duplicating the actual data. This reference comprises two main components: the distance to the previous occurrence (offset) and the length of the match. Literals, or data segments without matches, are copied directly into the output stream. LZ4's efficiency is largely due to its ability to perform these operations quickly, with minimal overhead, enabling high-speed compression that is particularly beneficial in real-time systems.

2.1.3 Decompression Process

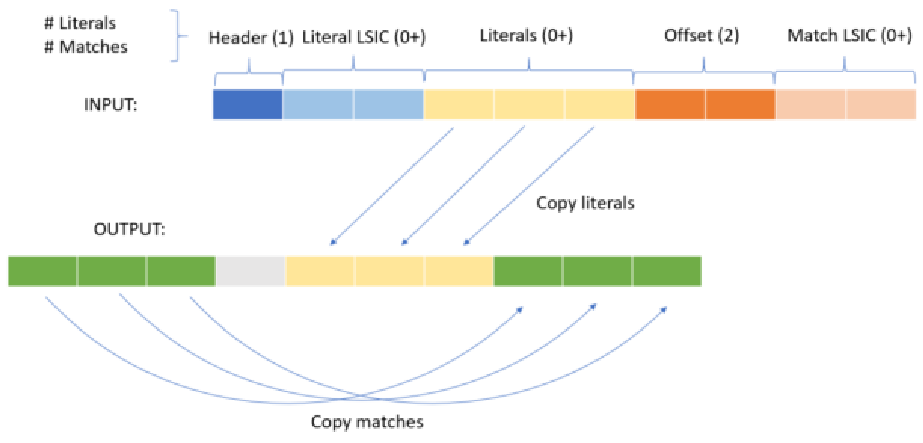

FIGURE 2.2: LZ4 decompression algorithm

The decompression process in LZ4 is straightforward and even faster than compression. Given that the format of the compressed data contains explicit lengths for literals and match sequences, the decompressor can rapidly reconstruct the original data by copying literals directly and replicating sequences based on their offsets and lengths. This simplicity and speed are critical advantages of LZ4, ensuring minimal delay in accessing decompressed data.

2.2 LZ4 File Format

LZ4 employs a lossless compression technique, ensuring that the original data can be perfectly reconstructed from the compressed data. This aspect is crucial for applications where data integrity is paramount. The file structure of LZ4 is designed to optimize speed, leveraging a stream of blocks that contain the compressed data. Each block within the LZ4 file can be compressed independently, allowing for parallel decompression and enhancing the algorithm's performance in multi-core systems.

The LZ4 file structure comprises two primary components: the frame and the block. The frame acts as the container for the compressed data, encapsulating the blocks and including important metadata such as the content size and checksums for data integrity verification. This metadata is essential for validating the integrity of the decompressed data, ensuring it matches the original input precisely.

Within the frame, the blocks are the core units of compression. LZ4 allows for varying block sizes, which can be adjusted based on the specific requirements of the application. This flexibility enables a balance between compression ratio and speed, as larger blocks generally offer better compression at the cost of increased processing time. Each block contains a sequence of bytes where literal bytes and sequences of repeated bytes are encoded. The encoding leverages the principle of back-references for sequences that have occurred previously within the stream, significantly reducing the file size for data with repetitive patterns.

A distinctive feature of the LZ4 file structure is its support for two modes of operation: block mode and stream mode. Block mode is tailored for scenarios where the entire data set is available at the time of compression, facilitating the use of larger blocks for improved compression ratios. Conversely, stream mode is designed for real-time applications, where data is compressed in smaller chunks as it becomes available. This mode is particularly advantageous for streaming applications, where latency and throughput are critical factors.

The LZ4 file structure also incorporates mechanisms for error detection and correction, underscoring the algorithm's robustness in diverse application scenarios. Checksums are used to ensure the integrity of both the compressed blocks and the overall frame, providing a safeguard against data corruption during transmission or storage.

2.2.1 Frame Format

Magic Number	Frame Descriptor	Data Block	(...)	EndMark	Content Checksum
4 bytes	3-15 bytes			4 bytes	0-4 bytes

TABLE 2.1: General Structure of LZ4 Frame format

- Magic Number

  4 Bytes, Little endian format. Value: 0x184D2204

- Frame Descriptor

  3 to 15 Bytes, to be detailed in chapter 1.2.2, as it is the most important part of the spec. The combined Magic Number and Frame Descriptor fields are sometimes called LZ4 Frame Header. Its size varies between 7 and 19 bytes.

- Data Blocks

  To be detailed in its own chapter.

- EndMark

  The sequence of data blocks concludes with the occurrence of a 32-bit value, specifically 0x00000000, which signifies the termination of the data stream.

- Content Checksum

  The "Content Checksum" plays a crucial role in validating the integrity of the data after decoding. This checksum is computed using the xxHash-32 algorithm on the original (decoded) data, with the seed value set to zero. Its application is optional and is only employed when the corresponding flag within the frame descriptor is set. This mechanism verifies that the data blocks are received accurately, in the correct order, and free of errors. Furthermore, it guarantees that the processes of encoding and decoding have not resulted in any discrepancies. The reliability of this checksum makes it a recommended tool for ensuring data integrity.

  In addition, the discussion includes the "EndMark" and "Content Checksum" fields, which are components of what is occasionally referred to as the LZ4 Frame Footer. These fields have a variable size, ranging between 4 and 8 bytes. This detail underscores the flexibility and adaptability of the LZ4 framing format in accommodating different requirements for data integrity verification.

- Frame Concatenation

  In certain scenarios, it may be advantageous to concatenate multiple frames, for instance, when augmenting an existing compressed file with new data without restructuring the entire file. Under these circumstances, each frame is endowed with a unique set of descriptor flags, considering each frame as an independent. The sole connection among the frames is the order in which they appear sequentially.

Frame Descriptor

FLG	BD	(Content Size)	Dictionary ID	Header checksum
1 byte	1 byte	0-8 bytes	0-4 bytes	1 byte

TABLE 2.2: Frame Descriptor

FLG byte

Bit	7-6	5	4	3	2	1	0
Field Name	Version	Block Independ.	Block Checksum	C. Size	C. Checksum	Reserved	Dict. ID

TABLE 2.3: FLG byte

- Version Number

  In the outlined specification, a 2-bit field designated as the version number is crucial, necessitating its configuration to ′01′. This is imperative as alternative values are incompatible with the current specification iteration, necessitating distinct flag configurations for other version numbers.

- Block Independence flag

  Regarding block independence, a pivotal flag determines the relationship between data blocks. A setting of ′1′ denotes block independence, ensuring that each block can be decoded without reference to others. Conversely, a ′0′ setting indicates a dependency on preceding blocks within the LZ4 window size limit of 64 KB, necessitating sequential decoding. This dependency notably enhances the compression efficiency for smaller data blocks but precludes the feasibility of random access or the application of multi-threaded decoding techniques.

- Block checksum flag

  The specification also introduces a block checksum flag, enabling the association of a 4-byte checksum with each data block. This checksum, derived from the xxHash-32 algorithm applied to compressed data, aims to facilitate the immediate identification of data corruption issues, whether from storage or transmission errors. The implementation of block checksums remains optional.

- Content Size flag

  Further, the Content Size flag, when activated, signifies the inclusion of the uncompressed data size, represented as an 8-byte unsigned little-endian value after the flags, offering an optional utility for specifying data volume.

- Content checksum

  Additionally, the presence of a content checksum flag indicates the appending of a 32-bit checksum following the EndMark, serving as an integrity verification measure for the data content.

- Dictionary ID flag

  Lastly, the Dictionary ID flag, upon being set, mandates the inclusion of a 4-byte Dictionary ID field following the descriptor flags and the Content Size, further contributing to the specification's flexibility in data compression and integrity assurance mechanisms.

- Reserved Bits

  The specification mandates that the value of reserved bits must be set to zero. These bits are earmarked for potential future use, possibly for the introduction of new features. Should the specification evolve to include these new features, decoders that comply with the existing version will not support decoding frames that utilize these reserved bits.

- Content Size

The content size refers to the size of the data before it undergoes compression. Including this piece of information is optional and is contingent upon a specific flag being activated. When present, the content size is articulated using an unsigned 8-byte format, accommodating a maximum of 16 Exabytes and adopting a Little-endian byte order. This value serves an informational purpose, aiding in either the display of data or the allocation of memory. Decoders can choose to either overlook this information or use it to verify the accuracy of the content.

- Dictionary ID

  Utilizing a dictionary can significantly enhance the compression efficiency of short data sequences. In this context, the dictionary acts as a pre-established "known prefix," enabling the compressor to encode the data more compactly. Depending on the configuration specified by the frame descriptor, each block within a frame may be initialized with the dictionary independently, in the case of independent blocks, or collectively at the frame's commencement for linked blocks. It is imperative for both the compression and decompression processes to utilize the identical dictionary to ensure the data remains decodable.

  The introduction of the Dict-ID field serves as a mechanism to aid decoders in identifying the appropriate dictionary required for the accurate decompression of a frame. This field, an unsigned 32-bit value stored in little-endian format, is included only when its corresponding flag is activated. It is important to note that within a frame, only a single instance of the Dict-ID field is permissible.

  Moreover, the Dict-ID field is not mandatory. The selection of the dictionary can be communicated through alternative means, such as being inferred from the application's context.

- Header Checksum

  Regarding the header checksum, it encompasses a one-byte checksum that validates the integrity of the frame descriptor and any optional fields it may contain. This checksum is derived from the second byte of the xxh32() hash function, calculated as (xxh32()»8) & 0xFF, using a zero seed and the entire Frame Descriptor, including any present optional fields, as input. An incorrect checksum is indicative of a discrepancy within the descriptor.

BD byte

Bit	7	6-5-4	3-2-1-0
Field Name	Reserved	Block Max Size	Reserved

TABLE 2.4: BD byte

- Block Maximum Size

  In the realm of data decoding, understanding the maximum size of blocks is crucial for memory allocation purposes. The term "size" pertains to the size of the data prior to compression. The specification outlines several predetermined sizes for blocks, which are essential for decoders to know in order to allocate appropriate memory. It is important to note that decoders have the discretion to reject block sizes that exceed any system-specific thresholds. Furthermore, certain values are presently not in use but may find application in subsequent revisions of the specification. Decoders that adhere to the current version of the specification are equipped to handle only the block sizes explicitly mentioned within it. Block Maximum Size is one value among the following:

  1. 4 or 100
  2. 5 or 101
  3. 6 or 110
  4. 7 or 111

2.2.2 Block Format

Block Size	**data sequence**	**(Block Checksum)**
4 bytes		0 - 4 bytes

TABLE 2.5: Data Blocks

Token	**Literal length(optional)**	**Literals**	**Offset**	**Match length(optional)**
1-byte	0-n bytes	0-L bytes	2-bytes	0-n bytes

TABLE 2.6: Data sequence

An LZ4 compressed block is intricately structured to optimize data compression through a series of sequences, each sequence being a combination of literals and a match copy operation, thus facilitating effective data encoding.

This detailed architecture is pivotal in achieving LZ4's renowned compression efficiency and speed.

The foundation of each sequence is marked by the initiation of a token, a one-byte entity that bifurcates into two distinct 4-bit fields. This binary division allows each field to span a range from 0 to 15, encapsulating the sequence's initial parameters. The upper 4 bits of the token delineate the length of the subsequent literals, serving as a prelude to the data compression process. This length quantification is direct for values less than 15, indicating the exact number of literals incorporated within the sequence, with the possibility of zero indicating the absence of literals.

However, the architecture introduces a complexity when this value reaches the threshold of 15. In such instances, the format necessitates additional bytes to accurately convey the total length of literals, with each byte capable of representing a value from 0 to 255. This incremental mechanism ensures a flexible and dynamic representation of literal lengths, accommodating sequences of varied sizes. The continuation of reading and adding bytes persists as long as a byte value of 255 is encountered, symbolizing the sequence's capacity for indefinite expansion within the constraints of practical implementation limits. This system underscores the absence of a predefined "size limit" within the LZ4 block format, although practical constraints are acknowledged in application-specific contexts, as further elaborated in sections dedicated to implementation notes.

Example 1: A literal length of 48 will be represented as:

- 15: value for the 4-bits High field
- 33: (=48-15) remaining length to reach 48

Example 2: A literal length of 280 will be represented as:

- 15: value for the 4-bits High field
- 255: following byte is maxed, since 280-15 >= 255
- 10: (=280 - 15 - 255) remaining length to reach 280

Example 3: A literal length of 15 will be represented as:

- 15: value for the 4-bits High field
- 0: (=15-15) yes, the zero must be output

Note: These examples are taken from [2]

In the LZ4 compression algorithm, the process of data compression within a block follows a structured format, beginning with the interpretation of a token followed by optional length bytes, which precede the actual data literals. These literals, the uncompressed data segments, are quantified precisely by the length decoded from the preceding token and length bytes. It is crucial to understand that the number of literals can vary, extending to the possibility of there being none. This variability underscores the flexibility and efficiency of the LZ4 algorithm in adapting to the data it compresses.

After the literals, the block format transitions into a match copy operation, a pivotal component in the compression mechanism. This operation is initiated by deciphering an offset value, encoded as a 2-byte entity utilizing little-endian formatting. This means the first byte represents the lower significant value, and the second byte, the higher. The offset is essential in identifying the location within the previously processed data from which a match can be copied. This operation underlines the dictionary-based approach of LZ4, where earlier data segments are reused to compress repeating patterns efficiently. The offset's value is directly proportional to the distance backward from the current position, with 1 indicating one byte behind the current position. The design of LZ4 caps the maximum offset at 65535, acknowledging the limitations of a 2-byte representation. Values beyond 65535 are beyond the encoding capability of the format, emphasizing the balance between compression efficiency and complexity. An offset value of 0 is deemed invalid, signifying a corrupted block. This safeguards against data integrity issues, ensuring that each block adheres to the protocol for valid compression.

The LZ4 file's block format employs a nuanced method for encoding match lengths, which is critical for understanding its compression mechanism. This process hinges on the utilization of the second token field, specifically the lower 4 bits. This segment, with its range from 0 to 15, plays a pivotal role in determining the length of a match. Notably, a 0 value in this context does not signify an absence but rather the minimum operational length for a copy action, which is established at 4 bytes. This baseline is referred to as the minmatch. Thus, any value within this field directly translates to a match length that must incorporate an additional 4 bytes, spanning a range from 4 to 18 bytes.

The encoding takes a special turn when the value reaches 15, indicating that the match length extends beyond 18 bytes. In such instances, the format necessitates the sequential reading of extra bytes, each with a potential value between 0 and 255, to cumulatively ascertain the final match length. The presence of a 255 value amongst these bytes signals the requirement to read and integrate yet another byte into the total length. This mechanism allows for an indefinite extension of the match length, constrained only by practical implementation limits rather than theoretical ones. This flexible yet systematic approach underscores the LZ4 algorithm's capacity for adaptability and efficiency in data compression.

Upon concluding the decoding of a match length, the algorithm proceeds to the next sequence within the LZ4 block. This transition is marked by the initiation of a new token, signifying the start of a distinct sequence. It is essential to recognize that this sequential flow facilitates the LZ4 algorithm's dynamic handling of data, enabling the efficient compression of diverse data types while maintaining high throughput rates.

In summary, the LZ4 block format's method for encoding match lengths exemplifies the algorithm's sophisticated approach to data compression. By employing a flexible yet precise mechanism for determining match lengths, coupled with a structured sequence processing system, LZ4 stands out as a robust solution for fast and effective data compression. This detailed exploration not only underscores the technical intricacies of the LZ4 algorithm but also highlights its practical applicability across various computing environments.

2.2.3 End of block conditions

In the LZ4 compression, the termination of a block is governed by a set of specific criteria designed to ensure data integrity and compatibility with existing decompression algorithms. These end-of-block conditions are crucial for maintaining the efficiency and reliability of the LZ4 compression algorithm, particularly in scenarios involving data streams of varying lengths.

The termination of an LZ4 block is uniquely defined to ensure that decompression processes can accurately and efficiently reconstruct the original data. One fundamental rule dictates that the final sequence within a block consists solely of literal bytes, with the block concluding immediately subsequent to these literals, omitting the need for an offset field. This ensures that the decoder precisely identifies the end of the compressed data stream. Additionally, the protocol mandates that the last five bytes of the input must be literals. This requirement guarantees that the final sequence contains a minimum of five literal bytes, thereby facilitating a consistent decompression process across diverse data sizes.

A noteworthy specification addresses inputs smaller than five bytes. In such instances, the LZ4 algorithm treats the entire input as a single sequence of literals. This approach allows for the compression of even the smallest data sizes, including an empty input, which is represented by a zero byte. This zero byte is interpreted as a final token that contains neither literals nor a match, showcasing the algorithm's flexibility in handling various input lengths.

The rules concerning the placement of the last match within a block further underscore the algorithm's precision. The last match is required to commence at least 12 bytes prior to the block's conclusion and is considered a component of the penultimate sequence. This sequence is succeeded by the final sequence, which, as previously mentioned, is comprised exclusively of literals. This stipulation prevents the possibility of compressing blocks shorter than 12 bytes, as such blocks do not provide sufficient space for the required sequence of literals and match. This limitation extends to independent blocks shorter than 13 bytes, which necessitate the initiation of the block with at least one literal byte, thereby enabling subsequent matches to replicate the literal.

These meticulously designed end conditions are not arbitrary but are instead implemented to assure backward compatibility and optimal performance with a broad spectrum of decoders, many of which are engineered for high-speed operations. The adherence to these rules enables the LZ4 algorithm to deliver fast, reliable compression across a wide array of data types and sizes, ensuring that even when a block does not conform to these conditions, it can be identified and potentially rejected by a compliant decoder as incorrect. This rigorous framework underscores the algorithm's commitment to data integrity and efficiency, essential traits for modern data compression techniques.

Chapter 3

Related Work

In this chapter there will be analysed various works regarding LZ4 decompression in many platforms. Platforms such as CPUs, GPUs and FPGAs. This will give us a good idea of what is happening in the area of data decompressing in the field and also have a benchmark on what our work can accomplish.

3.1 CPU

3.1.1 Hardware

The benchmarking process will commence with CPUs as the reference point, acknowledging their position as the least efficient hardware category in the context of our study. This is not to undermine the CPU's versatility and critical role in general-purpose computing but to contextualize its performance in specialized tasks such as compression and decompression. For the purpose of this analysis, we will employ Level 1 compression, recognized as the default setting, alongside corresponding decompression performance metrics. This level has been selected for its widespread adoption and relevance to a broad spectrum of applications.

The hardware under scrutiny will encompass commercial CPUs that represent the pinnacle of current technology. This selection criteria ensure that the data derived from the benchmarks reflects the capabilities of state-of-the-art hardware, thereby providing insights into the upper limits of performance achievable with contemporary CPU architectures.

The compilation of performance data will draw from OpenBenchmarking.org[3], a reputable source that aggregates user-contributed benchmark results. This study will consider a dataset consisting of 267 public submissions, recorded between 19 January 2024 and the latest entry on 28 February 2024. This dataset offers a robust foundation for analysis, ensuring that the conclusions drawn are based on a significant volume of data points.

It is crucial to recognize the inherent variability in performance outcomes, especially within the Linux and open-source ecosystem. The diversity in operating system configurations can introduce a wide range of performance variances. Therefore, the forthcoming analysis is designed to provide generalized insights into hardware performance, rather than definitive benchmarks applicable to all conceivable system setups. This approach acknowledges the complex interplay between hardware capabilities and software configurations, emphasizing the importance of considering a broad spectrum of operating environments.

The levels in the dropdown menu represent different compression settings for LZ4, affecting the balance between compression speed and ratio. Higher levels typically provide better compression at the cost of speed, while lower levels are faster but less efficient in reducing file size. Each level is tailored for specific use cases, ranging from real-time applications needing quick data transfer to scenarios where maximum compression is desired despite longer processing times.

3.1.2 Timing

COMPONENT	PR	# CPR	MB/s (AVG)
AMD Ryzen 9 7950X 16-Core	97th	9	6473
AMD Ryzen 7 7700 8-Core	96th	4	6435
AMD Ryzen 7 7700X 8-Core	94th	7	6411
AMD Ryzen Threadripper 7980X 64-Core	93rd	8	6395
AMD Ryzen 5 7600X 6-Core	93rd	6	6395
AMD Ryzen 9 7900X 12-Core	88th	4	6320
AMD Ryzen 9 7900X3D 12-Core	82nd	4	6187
AMD Ryzen 9 7950X3D 16-Core	82nd	4	6171
AMD Ryzen 7 8700G	81st	15	6113
Intel Core i9-14900K	80th	9	6092
AMD Ryzen 5 8600G	76th	11	6078

TABLE 3.1: Silesia archive Performance[3]

3.2 GPU

3.2.1 Hardware

Analysis of NVIDIA GPUs: A100, A30, A10, and H100 PCIe

This analysis presents a detailed comparison of NVIDIA's GPUs: A100, A30, A10, and H100 PCIe, focusing on their specifications, performance, intended applications, and cost-effectiveness. Each of these GPUs targets specific market segments and applications, making them suitable for various computing tasks.

Specifications

GPU Model	CUDA Cores	Performance(TFLOPs)	Memory Type	Memory Size
A100	6912	312	HBM2	40GB / 80GB
A30	4096	165	HBM2	24GB
A10	6912	130	GDDR6	24GB
H100 PCIe	16896	1000	HBM3	80GB

TABLE 3.2: Comparison of NVIDIA GPU Specifications

Here's a breakdown of the table:

- GPU Model: This is the model name of the GPU.
- CUDA Cores: The number of CUDA cores in the GPU, which are parallel processors that can handle computing tasks.
- Performance: The peak performance capability of the GPU, measured in teraflops (TFLOPs).
- Memory Type: The type of memory used in the GPU, with HBM (High Bandwidth Memory) being faster than GDDR (Graphics Double Data Rate).
- Memory Size: The total amount of memory available on the GPU.

Performance

In terms of performance, the A100 and H100 stand out as the powerhouses among the four, designed to tackle the most demanding AI and HPC tasks. The A100's architecture is optimized for deep learning and scientific computing, offering unmatched computational capabilities. The H100 takes this a step further, incorporating NVIDIA's latest advancements to deliver groundbreaking performance in AI model training and simulation.

The A30 and A10, while not as powerful as their counterparts, offer significant capabilities for their intended use cases. The A30 excels in AI inference and light training tasks, providing a balance of performance and efficiency. The A10, with its focus on mixed workloads, delivers competent AI performance and excellent capabilities for graphics-intensive applications.

Intended Applications

Each GPU is designed with specific applications in mind:

The A100 is best suited for deep learning training and inference, scientific computing, and data analytics, where its computational power can significantly reduce processing times. The A30 targets enterprises and cloud service providers needing a versatile GPU for a mix of AI, HPC, and inference tasks. The A10 is ideal for creative and design professionals requiring a blend of AI computational capabilities and graphics performance. The H100 is aimed at cutting-edge AI research and large-scale HPC environments, where its advanced AI and computing capabilities can accelerate the most complex tasks.

Cost-Effectiveness

When evaluating cost-effectiveness, it's essential to consider the total cost of ownership, which includes not just the initial purchase price but also power consumption and cooling requirements. The A100 and H100, while offering superior performance, come with a higher price tag, making them best suited for environments where computational speed and efficiency are paramount. The A30 and A10 offer a more balanced approach, providing good performance at a lower cost, making them suitable for a wider range of applications and budgets.

The A100 and H100's advanced features and performance justify their higher cost for organizations that require the utmost in computational capabilities. For those with more moderate needs or budget constraints, the A30 and A10 provide capable alternatives that can deliver excellent performance in a variety of tasks without the same level of investment.

Conclusion

In conclusion, NVIDIA's A100, A30, A10, and H100 PCIe GPUs cater to a broad spectrum of computing needs, from AI and deep learning to graphics and visualization. The A100 and H100 are unparalleled in their performance and are suited for tasks that demand the highest computational capabilities. In contrast, the A30 and A10 offer versatility and cost-effectiveness for a wide range of applications. The choice among these GPUs should be guided by the specific requirements of the tasks at hand, including the necessary computational power, budget constraints, and energy efficiency considerations.

3.2.2 Timing

Benchmark	**HW technology**	**Results**
Data analytics: INT columns	H100 PCIe	472.28 GB/s
Data analytics: INT columns	A100	369.79 GB/s
Data analytics: INT columns	A30	224.72 GB/s
Data analytics: INT columns	A10	228.71 GB/s
Graphics: Textures Data	H100 PCIe	114.57 GB/s
Graphics: Textures Data	A100	81.61 GB/s
Graphics: Textures Data	A30	39.24 GB/s
Graphics: Textures Data	A10	50.27 GB/s
Graphics: Geometry Data	H100 PCIe	51.77 GB/s
Graphics: Geometry Data	A100	39.08 GB/s
Graphics: Geometry Data	A30	21.86 GB/s
Graphics: Geometry Data	A10	27.09 GB/s
Silesia	H100 PCIe	43.79 GB/s
Silesia	A100	34.13 GB/s
Silesia	A30	22.07 GB/s
Silesia	A10	26.59 GB/s

TABLE 3.3: Timing Performance[4]

3.3 FPGA

3.3.1 Hardware

Xilinx Alveo U200

Card Specifications	
Compute	
INT8 TOPs (peak)	18.6
Dimensions	
Height	Full
Length	¾
Width	Dual Slot
Memory	
Off-chip Memory Capacity	64 GB
Off-chip Total Bandwidth	77 GB/s
Internal SRAM Capacity	35 MB
Internal SRAM Total Bandwidth	31 TB/s
Interfaces	
PCI Express	Gen3x16
Network Interfaces	2x QSFP28 (100GbE)
Logic Resources	
Look-up Tables (LUTs)	892,000
Power and Thermal	
Maximum Total Power	225W
Thermal Cooling	Passive

TABLE 3.4: U200 Card Specifications

The Xilinx Alveo U200 is a highly capable accelerator card designed to significantly boost the performance of a wide range of applications, including data analytics, machine learning, and video processing. As part of Xilinx's Alveo series, the U200 card is engineered for both cloud and on-premises applications, offering a flexible solution for accelerating computing workloads.

- Key Features
  - High Performance

    Equipped with Xilinx UltraScale+ FPGA, it delivers exceptional speed and efficiency for various compute-intensive applications.
  - Adaptable Acceleration

Offers customizable acceleration capabilities, allowing users to optimize their applications for performance or power efficiency.

- Concurrent Computing

  Enables the execution of multiple tasks simultaneously, increasing throughput and efficiency.

- Accelerated Computing

  It is designed to accelerate a wide range of applications, from machine learning models and data analytics to video processing, providing significant speedups compared to traditional computing solutions.

• Benefits

- Efficiency

  By offloading specific computational tasks to the Alveo U200, systems can achieve higher throughput and lower latency, leading to more efficient data processing and analysis.

- Cost-Effective

  Utilizing the U200 can lead to lower total cost of ownership by reducing the need for additional computing resources and lowering power consumption.

- Scalability

  The ability to customize and reconfigure the FPGA for different tasks makes the U200 a scalable solution that can evolve with changing computing needs.

3.3.2 Timing

Source	**HW technology**	**Results**
Xilinx[5]	Xilinx Alveo U200	1.1 GB/s
Xilinx Stream	Xilinx Alveo U200	293 MB/s
AMD[6]	Xilinx Alveo U200	443 MB/s
AMD Stream	Xilinx Alveo U200	368 MB/s

TABLE 3.5: Hardware Performance Metrics Silesia Benchmark

3.3.3 Resources

Source	LUT	LUTMem	REG	BRAM	URAM	Fmax (MHz)
Xilinx	8.8K	692	6.7 K	2	4	262
Xilinx Stream	802	32	1K	16	0	300
AMD	7.3K	1.1K	7 K	2	4	262
AMD Strean	6K	370	5K	0	4	300

TABLE 3.6: Resource Utilization

Chapter 4

System Architecture

The development of an LZ4 decompressor necessitates a comprehensive understanding of the algorithm's theoretical underpinnings, especially focusing on the essential components required for its efficient operation. At the core of LZ4 compression lies the strategic encoding of sequences, which serves as the primary mechanism for reducing data size. This encoding process underscores the need for a robust memory system capable of reconstructing the original file from the compressed data. Such a system must facilitate easy access to previously decompressed sequences, thereby necessitating the integration of Random Access Memory (RAM). RAM plays a pivotal role in not only storing the decompressed sequences for subsequent retrieval but also in ensuring the seamless reconstruction of the original file.

To manage the flow of compressed and decompressed data efficiently, the decompressor architecture incorporates temporary storage mechanisms. These mechanisms are essential for holding incoming compressed bytes and the corresponding decoded bytes destined for the output file. Given the linear nature of the compression and decompression processes, wherein data is processed in a sequential manner without random access, a First In First Out (FIFO) queue emerges as an optimal solution. The FIFO queue facilitates a streamlined process by ensuring that bytes are decoded and written out in the exact order they are received, thereby eliminating any potential bottlenecks or inefficiencies in the data flow.

The quintessence of the LZ4 decompressor's design lies in its interpreter or parser, colloquially referred to as the "mind" of the system. This critical component is embodied by a Finite State Machine (FSM), which orchestrates and fine-tunes the entire decompression process. The FSM meticulously controls the input data flow, making real-time decisions on the placement of literals, and managing the sequence of operations required to reconstruct the original data accurately. Furthermore, the FSM is adept at navigating complex scenarios, including stall conditions and error states, ensuring the decompressor's resilience and robustness.

Error detection and management form an integral part of the decompressor's control unit, bolstered by the FSM's capabilities. The FSM's role extends to addressing corner cases and stall conditions, which may arise due to anomalies in the compressed data or unexpected operational scenarios. To effectively resolve these challenges, the system's design incorporates multiplexers (muxes). These muxes play a pivotal role in selecting between different data paths or operational modes based on the FSM's directives, thereby enabling the system to adapt to various conditions seamlessly.

In essence, the LZ4 decompressor's architecture is a harmonious blend of memory systems, data management mechanisms, and intelligent control logic. The use of RAM ensures that decompressed sequences are readily accessible, facilitating efficient data reconstruction. The FIFO queue, in tandem with RAM, streamlines the flow of data through the decompressor, maintaining the integrity and order of the decompressed output. At the helm, the FSM orchestrates the decompression process with precision, managing literals, sequences, and addressing any operational anomalies that may arise. This comprehensive approach ensures that the LZ4 decompressor not only achieves high efficiency and speed but also maintains the highest standards of data fidelity and reliability.

In conclusion, the theoretical design of an LZ4 decompressor is a testament to the intricate balance between efficient data management, sophisticated control logic, and the need for flexible, error-resilient operation. By delving into the principles that underpin each component of the decompressor, we gain a deeper understanding of the algorithm's capabilities and the technological ingenuity required to implement it effectively. This expanded exploration provides a solid foundation for further research and development in the field of data compression, highlighting the critical role of hardware engineering in advancing digital storage and transmission technologies.

Chapter 5

FPGA Implementation and Architecture Design

In this chapter the implementation of the whole project will be described. Every detail that creates the decompressor will be analyzed.

5.1 Datapath

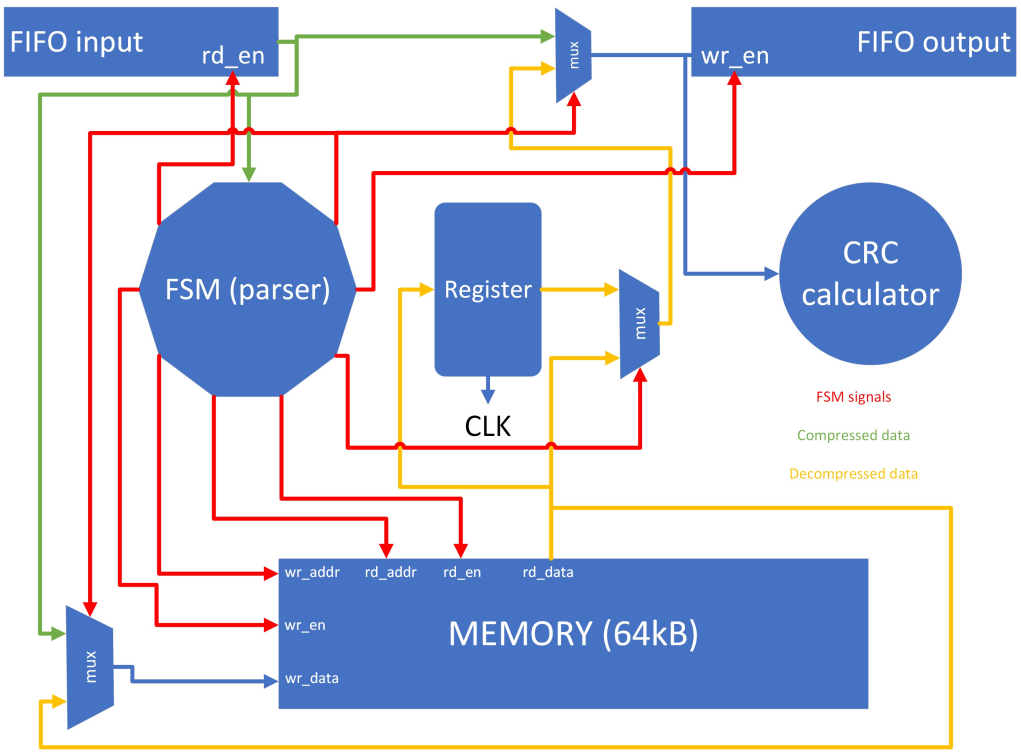

FIGURE 5.1: The Datapath[7]

5.1.1 Memory

Overview of Memory Functionality

The memory unit in the LZ4 decompression datapath plays a pivotal role. It serves the dual purpose of storing the decompressed data and facilitating the retrieval of previously decompressed bytes, which are crucial for the algorithm's operation. The LZ4 algorithm, as we have analyzed, heavily relies on referencing past decompressed data to effectively decompress new data.

Memory vs. FIFO (First In, First Out queues)

A critical design choice in this system is the use of a memory unit instead of a FIFO (First-In, First-Out queue). This decision stems from the algorithm's need to access decompressed data non-sequentially. In a FIFO, the reading address is automatically set and follows a strict order, which is incompatible with the LZ4's requirement for random access. The decompressor needs the flexibility to set the reading address based on the specific data block being processed. Each data block in LZ4 contains a unique sequence of literals and a distinct offset for data retrieval. This offset, combined with the current write address, determines the required reading address in the memory.

Memory Structure and Data Storage

The memory is structured to store literals from the second part of the data section. These literals are not only fed to the output for decompression but are also stored within the memory for potential future reference. This dual role emphasizes the importance of the memory unit in ensuring the continuity and integrity of the decompression process.

Memory Size Configuration

The size of the memory is a user-configurable parameter, defined in the source code. It is recommended to size the memory based on the maximum window size of the LZ4 algorithm, which is 64 KB. This recommendation is guided by the need to balance memory resource utilization with the requirement to accommodate the largest possible data window that the LZ4 algorithm might reference. By aligning the memory size with the LZ4 window size, the system ensures that it has adequate capacity to store and retrieve any data sequence that the decompression process might require.

5.1.2 FIFO

Role and Function of Input and Output FIFOs

The datapath of the LZ4 decompression algorithm incorporates two essential components: the input FIFO and the output FIFO. These First-In, First-Out queues serve as critical interfaces for data flow within the system. The input FIFO is responsible for receiving and temporarily storing bytes from the compressed file, acting as the primary feed for the decompression system. Conversely, the output FIFO holds the decompressed data, ready for external retrieval and use.

Interaction with Finite State Machine and Memory

Each clock cycle, a byte from the compressed file is fed from the input FIFO into the system. This byte plays a pivotal role in the decompression process. It is first interpreted by the finite state machine, a component responsible for determining the nature of the data (whether it is literal data or a reference to previous data). The finite state machine's interpretation of this byte guides the subsequent decompression steps.

Simultaneously, the input byte is directed to two destinations: the output FIFO and the memory unit. The byte's transmission to the output FIFO is part of the process of reconstructing the original, uncompressed data. On the other hand, its delivery to the memory unit is crucial for future decompression tasks, particularly when the algorithm needs to reference previously decompressed data.

Importance in the Decompression Process

The input and output FIFOs are fundamental in maintaining a smooth and efficient flow of data through the LZ4 decompression system. They ensure that the finite state machine and memory unit are continuously supplied with the necessary data at the required pace. This coordination is vital for the real-time processing capabilities of the LZ4 decompression algorithm.

5.1.3 Multiplexers

Function and Roles of Multiplexers

The LZ4 decompression datapath utilizes three multiplexers, each playing a distinct role in managing data flow and controlling signals. These multiplexers are strategically placed: two are situated behind the output FIFO input, and one is behind the memory unit.

Multiplexers Associated with the Output FIFO

Output FIFO Input Selection Multiplexer: The first multiplexer near the output FIFO is tasked with selecting the source of data for the output FIFO. It chooses between the input FIFO and the memory unit. This selection is critical because the output FIFO can receive data directly from the input FIFO (representing uncompressed literals) or from the memory (representing previously decompressed data). The finite state machine (FSM) drives the control signal for this multiplexer, ensuring the correct source is selected based on the current state and requirements of the decompression process.

Stall State Handling Multiplexer: The second multiplexer near the output FIFO addresses stall conditions in the system. A stall occurs when either of the FIFOs becomes empty or full, disrupting the normal flow of data. This multiplexer, with a register, manages these stall conditions by ensuring the correct literal is written to the output FIFO. When the system enters a stall state, read, and write signals are set to zero, necessitating the need to maintain data integrity. The multiplexer achieves this by drawing data from the register's debounced exit during stall states, thus avoiding any loss of data.

Multiplexer Associated with the Memory

Memory Data Selection Multiplexer: The multiplexer behind the memory is responsible for determining what data is written into the memory unit. The data written can either be a literal from the input FIFO or a repeated literal from previously decompressed data. This selection is crucial because it ensures that the memory contains a comprehensive record of decompressed data, facilitating accurate and efficient decompression. The finite state machine also controls the select signal for this multiplexer, aligning the data written to memory with the current operational phase of the LZ4 algorithm.

5.2 Stall Conditions

5.2.1 Input FIFO stall conditions

Stall conditions in the LZ4 decompression datapath represent scenarios where the normal flow of data through the FIFOs (First-In, First-Out queues) is interrupted, necessitating a temporary pause in operations. These conditions are specific to each FIFO and are triggered by distinct situations.

Stall Conditions in the Input FIFO

Trigger for Stall in Input FIFO: The stall condition in the input FIFO is initiated when this FIFO is empty. An empty input FIFO indicates that the external data source has not supplied sufficient data for processing. This lack of data halts the decompression process since there is nothing available to interpret or decompress.

Impact and Management of Stall in Input FIFO

During a stall caused by an empty input FIFO, all activities related to data interpretation must be paused. This pause remains in effect until at least one byte of data is stored in the input FIFO. The impact of an empty input FIFO is primarily observed in specific stages of the decompression process. These stages include token processing, handling of literals, offset calculations, and the initial frame processing at the start of the file.

Non-affected Processes

It is important to note that not all stages of the decompression process are impacted by the input FIFO's stall condition. For instance, when the decompression procedure involves copying already decrypted bytes from the memory, the state of the input FIFO is irrelevant. In such cases, the decompression process can continue unaffected, as it relies on previously decompressed data stored in memory rather than new data from the input FIFO.

5.2.2 Output FIFO stall conditions

Stall Conditions Arising from Output FIFO

In the LZ4 decompression datapath, stall conditions can also occur in relation to the output FIFO, particularly when it becomes full. This situation presents a critical challenge as it risks data loss if not effectively managed.

Impact of a Full Output FIFO

Risks of a Full Output FIFO: When the output FIFO is full, there is a risk of overwriting data that has not yet been transferred to the final decompressed file. This situation necessitates a system stall to prevent data loss. The primary concern here is ensuring that data, especially that extracted from the memory, is not lost, or corrupted during this stall period.

Memory Reliability Concerns: While memory units are generally reliable, there is a non-zero risk that data held for multiple cycles could degrade or turn into 'garbage'. This risk is particularly pertinent in scenarios where the system experiences prolonged stalls or operates under sub-optimal conditions.

Solution: Register and Multiplexer Integration

Implementation of a Register: To mitigate the risk associated with memory reliability during stall conditions, a register is placed at the output of the memory. This register serves as a temporary holding area for the data intended for the output FIFO. Its purpose is to maintain the integrity of this data, especially during stall conditions when writing to the output FIFO is not immediately possible.

Role of the Multiplexer: A multiplexer is employed to facilitate the correct routing of data under these circumstances. It plays a crucial role in selecting the appropriate data source for the output FIFO. During the initial cycle following a stall condition, the multiplexer selects data from the register. This ensures that the accurate and uncorrupted data is written to the output FIFO. Subsequently, the multiplexer switches back to selecting data directly from the memory output.

5.3 Finite State Machine

The finite state machine (FSM) serves as the core control unit within the LZ4 decompression system, directing the flow and manipulation of data through precise signal management. This FSM is meticulously designed to handle various operational signals critical for the datapath's functionality, including the write and read addresses of the memory, write and read enables for both the memory and FIFOs, and the selection and stall control for multiplexers.

With thirteen distinct states, the FSM orchestrates the decompression process starting from the initial system reset, through the parsing of frame headers, managing block sizes and tokens, to the handling of literals and match lengths, and concluding with the final state or transitioning to an error state if anomalies are detected. Each state is uniquely responsible for managing data flow and ensuring the integrity and efficiency of the decompression process, reflecting the algorithm's complexity and the necessity for precise control mechanisms.

This FSM is not only a testament to the sophistication of the LZ4 decompression algorithm but also highlights the importance of a well-structured control system in managing complex data operations. It emphasizes the critical role of FSMs in hardware design, particularly in applications requiring high-speed data processing and accuracy. The parser of the input that controls all the signals. Specifically, the FSM controls these signals:

1. Write address of the memory
2. Read address of the memory
3. Write enable of the memory
4. Read enable of the memory
5. Write enable of the input FIFO
6. Read enable of the output FIFO
7. Stall choice in mux no 1
8. Write select in mux no 2 and 3

The FSM has thirteen states that describe the part of the data block we are or the process that is now occurring. The thirteen states are the following:

1. Initial
2. Magic Number
3. Frame bytes
4. Checksum Check
5. Block Size

6. Read token
7. Keep reading literals
8. Copy literals
9. Read offset
10. Keep reading match-length
11. Copy past
12. Final State
13. Error State

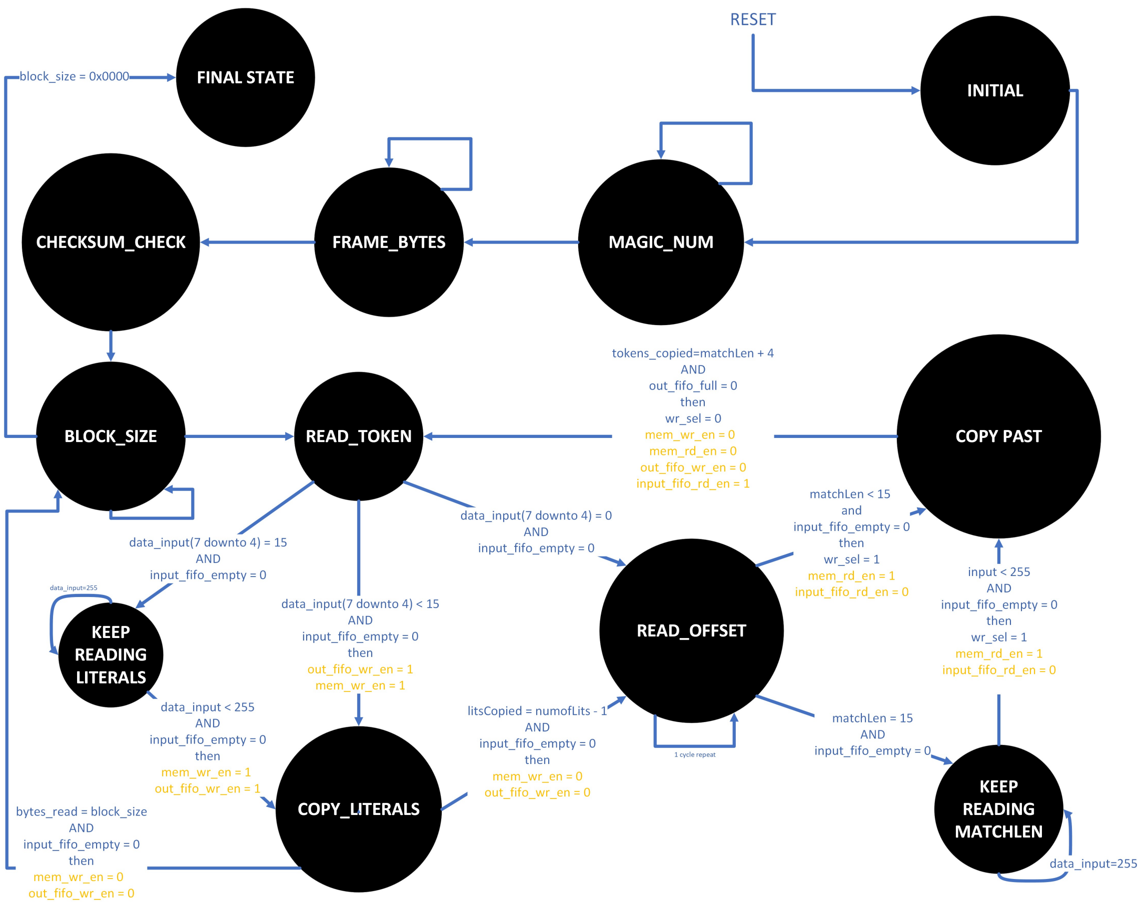

FIGURE 5.2: Finite State Machine[8]

5.3.1 Initial

In this initial state, the primary activity involves monitoring the input FIFO to ensure it contains data before commencing operations. This state is crucial as it establishes the system's readiness to begin processing compressed data by enabling the read signal. The transition from this state is contingent upon the presence of data in the input FIFO, signifying the FSM's move to actively handle compressed data. This phase underscores the FSM's role in managing data flow effectively, marking the transition from a state of readiness to active data processing.

5.3.2 Magic Number

In the "Magic Number" state of the finite state machine for LZ4 decompression, the system validates the presence of a specific predefined sequence of bytes at the beginning of the compressed file, known as the magic number. This step is crucial as it confirms the file format and ensures that the data being processed is indeed LZ4 compressed. The magic number is a constant value predefined in the LZ4 specification, serving as an identifier that unequivocally marks the file as compatible with LZ4 decompression algorithms. Successfully identifying the magic number allows the finite state machine to proceed confidently to the next stage, knowing that the file adheres to the expected LZ4 format. This validation step is foundational, setting the stage for the decompression process by verifying file integrity and format compliance.

5.3.3 Frame bytes

In the "Frame Bytes" state of the LZ4 decompression finite state machine, the system processes metadata that provides context about the compression procedure used on the data block. This state is executed twice to ensure thorough analysis of critical information, such as whether the data blocks are independent of one another or if a dictionary was used during compression. These details are instrumental in determining the decompression strategy and are essential for correctly interpreting the subsequent data. This stage lays the groundwork for the decompression process by establishing parameters that affect how data blocks are handled.

5.3.4 Checksum Check

In the "Checksum Check" state of the LZ4 decompression finite state machine, the focus is on the integrity verification of the decompressed data. The checksum, generated during the initial compression process and embedded within the compressed file, is stored and later compared against a computed checksum of the decompressed data. This step is crucial for ensuring data integrity, as it verifies that the decompressed data matches the original pre-compression data without any corruption or loss.

5.3.5 Block Size

In the "Block Size" state of the LZ4 decompression finite state machine, the system identifies the size of the forthcoming compressed data block. This information is crucial for the system to allocate resources and prepare for the decompression process. Additionally, this state incorporates a mechanism to detect the end of the compressed file, characterized by four consecutive zero bytes. Upon detecting this pattern, the FSM transitions to the final state, indicating the completion of the decompression process. If the end-of-file condition is not met, the FSM proceeds to the "Read Token" state. This block size detection is repeated across four cycles to accurately capture the four-byte length of the block size, ensuring precise preparation for handling the incoming data block.

5.3.6 Read token

In the "Read Token" state of the LZ4 decompression FSM, critical information is decoded to determine the next steps in the decompression process. This state interprets the token byte to extract the length of literals and the sequence of literals to be copied. Based on the decoded lengths, the FSM transitions to one of three possible states:

1. "Copy Literals" if the number of literals is less than 15.
2. "Keep Reading Literals" if exactly 15 (indicating the potential for additional literals)
3. "Read Offset" if no literals are present.

5.3.7 Keep reading literals

In the "Keep Reading Literals" state, the finite state machine focuses on gathering all necessary bytes that represent the literals to be decompressed. This accumulation process continues until a byte value less than 255 is encountered, signaling the completion of the literal gathering phase. Upon reaching this condition, the state transitions to "Copy Literals," where the actual decompression of these accumulated literals takes place. This state is essential for ensuring that all literals are correctly prepared for decompression, facilitating accurate reconstruction of the original data.

5.3.8 Copy literals

In the state where literals are copied, the finite state machine ensures that these literals are accurately transferred to the output FIFO and simultaneously written into the system memory. This stage requires meticulous control over the write address to ensure data integrity, especially as addresses approach their limits and must cycle back to the beginning. This cyclical address management is vital to prevent overwrites and maintain the continuous flow of decompressed data, highlighting the precision needed in handling memory operations during decompression. In this phase of the LZ4 decompression process, the finite state machine evaluates the progress within the current block, determining the appropriate next step based on two potential scenarios:

1. If the cumulative number of bytes read and decompressed matches the block size, indicating the end of the current block, the FSM transitions back to the "Block Size" state to prepare for processing a new block.
2. Alternatively, if the process has extracted all literals but has not reached the end of the block, indicating more data is to be decompressed, the FSM moves to the "Read Offset" state. This step involves interpreting the offset to continue decompressing data from memory, signifying the ongoing processing within the current block.

5.3.9 Read offset

In the "Read Offset" state, the finite state machine dedicates two clock cycles to interpret the offset value, which is presented in little endian format as per the LZ4 specification. This necessitates a specific handling approach to accurately interpret the byte order, ensuring the decompression algorithm correctly calculates the distance to reference previously decompressed data. This step is fundamental for enabling the algorithm to access and replicate sequences from the history buffer based on the offset information. After reading the offset, the finite state machine assesses the next steps based on the token’s low 4 bits. If these bits equal 15, indicating incomplete match length, the state transitions to "Keep Reading Matchlength" for further bytes. Otherwise, with less than 15, it suggests the next action involves copying from memory, pausing input FIFO data feed to accommodate the memory copy process.

5.3.10 Keep reading match-length

In the "Keep Reading Matchlength" state, the finite state machine accumulates bytes to determine the total length of data to be repeated from the decompressed content. This process continues until a byte value below 255 is encountered, signaling that the full match length has been gathered. Following this, the FSM transitions to the "Copy Past" state to replicate the specified sequence from the previously decompressed data, ensuring accurate reconstruction of the original dataset.

5.3.11 Copy past

During the "Copy Past" state in the LZ4 decompression finite state machine, previously decrypted bytes are replicated to both the output file and memory at a new address for later access. This state intricately manages both the reading and writing addresses. This dual handling of reading and writing addresses ensures efficient data management. Additionally, this state addresses stall conditions related to the output FIFO, maintaining uninterrupted data flow. Following the completion of this state, the FSM transitions to "Read Token," preparing to interpret the token for the next sequence, thus seamlessly continuing the decompression process.

5.3.12 Final State

In the "Final State" of the LZ4 decompression finite state machine, the completion of the decompression process is signified. This state is reached once the entire compressed file has been successfully decompressed, indicating that all data has been processed and the output is now fully available.

5.3.13 Error State

In the "Error State" of the finite state machine (FSM) for LZ4 decompression, the FSM transitions to this state to manage any errors that occur during the decompression process. Activities such as reading and writing are typically halted to address the error. However, detailed error handling mechanisms are not developed within the scope of this project, indicating a potential area for future enhancement to improve system robustness and reliability.

5.4 Address Handling

In the context of LZ4 compression, managing memory addresses is critical. The LZ4 algorithm, known for its high-performance lossless compression, operates by encoding sequences that have previously appeared in the data stream. It leverages a dictionary-like approach where sequences are stored and referenced by their position in the output stream, allowing for efficient encoding of repeating patterns. A key component in this approach is the concept of 'offsets', which denote the distance between the current sequence and the location of its match within the previously encoded data.

The offset is a numerical value that points to a location in the past of the data stream where a matching sequence can be found. In typical scenarios, calculating the read address from the write address using the offset is straightforward. However, a unique situation arises when the memory is treated in a cyclical manner, also known as 'flipped' memory. This occurs when the end of the available memory space is reached, and the writing continues from the beginning, effectively wrapping around.

In the case where the offset is greater than the write address as indicated in 5.3, a conventional subtraction operation is not sufficient to determine the correct read address. The diagram accompanying this explanation visualizes this peculiar condition. To address this, the LZ4 algorithm implements a specialized calculation that correctly identifies the location from which to copy the data.

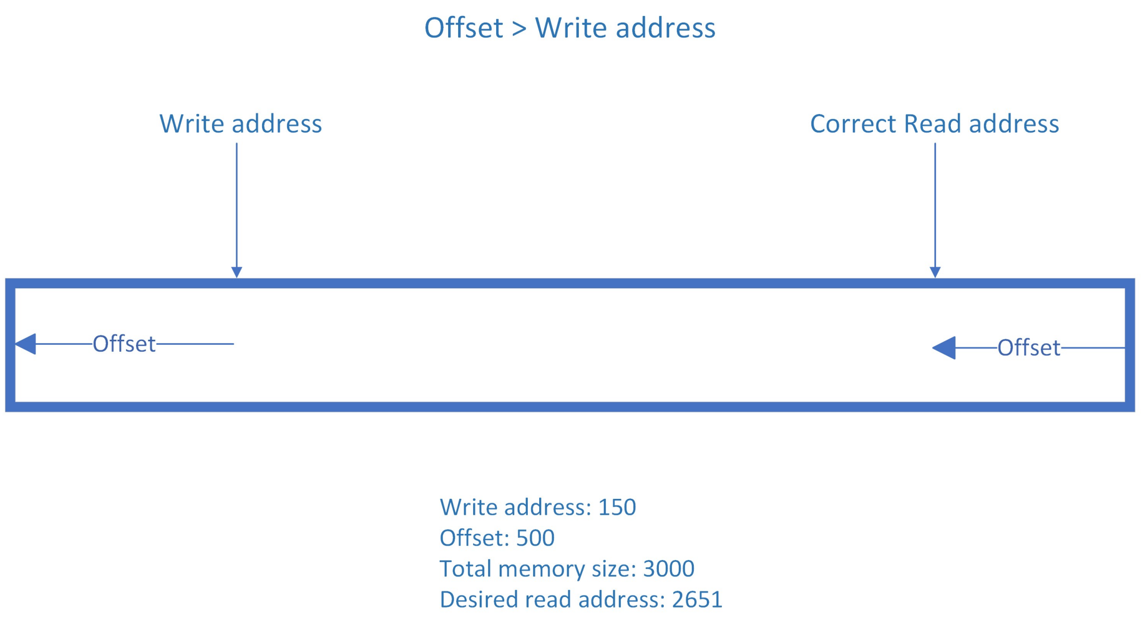

FIGURE 5.3: Offset handling

The formula for determining the read address in this scenario is as follows:

$$read\_address = sizeof(RAM) - offset + write\_address + 1$$

This equation takes into account the cyclical nature of the memory usage. By subtracting the offset from the total size of the RAM, we effectively move backwards through the memory space to locate the start of the desired sequence. Adding the write address then relocates this point relative to the current write position. Finally, the addition of 1 resolves the zero-based indexing used in computer memory addressing, which means that the first position is indexed as 0 rather than 1.

It is crucial to comprehend that the wrap-around memory model introduces complexity in address calculation, as it defies the linear progression of addresses. The LZ4 algorithm's ability to handle such complexities without error is a testament to its robustness and efficiency. The described method ensures that even when sequences are found at the very end of the memory, they can be accurately referenced and used for compression, thereby maintaining the LZ4 algorithm's high compression ratios and speed.

Chapter 6

Results

In this chapter the procedure will be explained and detailed, also the results regarding the resource utilization and the speed of the decompression. In this implementation, the results are defined from the decompression's ability to reconstruct the original file the same. In the figure below, the flowchart of the whole procedure is described.

6.1 Tools Used

6.1.1 Vivado IDE

Xilinx Vivado Integrated Design Environment (IDE), released in 2012, is the basis for all Xilinx tools. It serves as a GUI front-end for the Vivado Design Suite. All Vivado Design Suite tools integrate a native TCL interface, which can be accessed from IDE's GUI and the TCL console. Vivado IDE can compile, synthesize, implement, place and route FPGA hardware designs written in high-level languages such as C/C++, and HDLs such as VHDL and Verilog.

6.1.2 JetBrains CLion

CLion is an IDE for C and C++ available on Linux, macOS, and Windows, incorporating the CMake build system. It initially supports compilers like GCC and Clang, along with the GDB debugger, LLDB, and Google Test.

The CLion was used for the original compression algorithm already developed in C language[2] to understand both compression and decompression procedures. Also in was used to compress the test files in order to have a file to compare that is for sure correctly compressed.

6.2 Validation

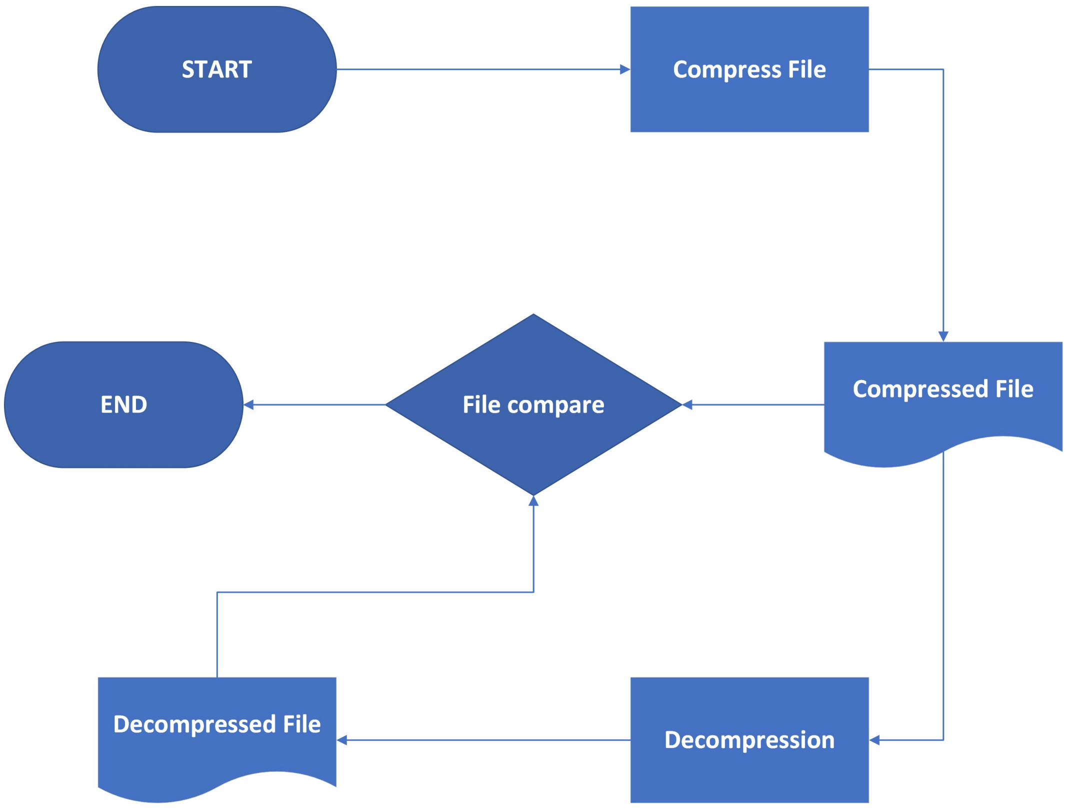

FIGURE 6.1: Validation Flowchart

6.2.1 Compression

The compression process utilized in the validation process uses the official LZ4 software, which can be found in its GitHub repository[2]. This repository is known for hosting the LZ4 compression algorithm.

Initially, a thorough examination of the file structure was undertaken to gain insights into the components that make up a compressed file. This foundational understanding is essential for effectively applying the compression technique.

Following this preliminary step, the project leveraged existing code samples available within the same LZ4 repository, specifically the frameCompress.c [9] file located in the examples section. This piece of code demonstrates the complete process of both compressing and decompressing an input file. Notably, it not only carries out the compression to produce a smaller file size but also decompresses the file to verify the integrity of the data post-compression. It is through this method that the compressed file, which is integral to this project, was generated. The compression setup for the final form of the compressed file is this:

- Max block size: 64KB
- Block dependency: Independent
- Content Checksum: OFF
- Frame type: LZ4 Frame
- Content Size: OFF
- Dictionary: OFF
- Block Checksum: OFF
- Compression level: Default

6.2.2 Decompression

The decompression process, integral to this study, was executed within the Vivado Xilinx Integrated Development Environment (IDE). This environment was chosen for its robust capabilities in handling complex digital logic designs, making it an ideal choice for implementing and testing compression and decompression algorithms.

In the specific context of this project, a compressed file undergoes decompression through a meticulously analyzed architecture. This architecture was previously studied and designed to closely replicate the original LZ4 algorithm's functionality, albeit tailored for hardware implementation. The objective of this process is to regenerate a file identical to the original source, thereby validating the integrity and efficacy of the decompression mechanism.

To assess the success of the decompression process, the project utilized two primary methods of verification. Firstly, the output file was scrutinized to confirm its byte-for-byte equivalence with the original file. This direct comparison serves as the most tangible measure of success, demonstrating the algorithm's capability to accurately restore the original data from its compressed form.

Secondly, and equally important, was the utilization of simulation tools available within the Vivado Xilinx IDE. These tools allow for a detailed examination of the decompression process on a cycle-by-cycle basis. By monitoring the signals and data flow within the architecture during simulation, the project could identify and rectify any discrepancies or errors. This granular level of analysis is crucial for understanding the dynamic behavior of the decompression algorithm in a hardware environment. It enables the identification of specific operational issues that might not be apparent from the output file alone.

Testbench

In the development of a comprehensive testbench, the primary objective was to facilitate the analysis and evaluation of the decompression system by inputting a compressed file and observing the reconstruction of the original file. This testbench is meticulously designed to simulate the real-world operation of the decompression algorithm, allowing for a detailed examination of its performance and reliability.

The process within the testbench begins with the sequential feeding of the compressed file into the system, byte by byte. This methodical input is crucial for mimicking the actual conditions under which the decompression algorithm operates, ensuring that the evaluation is both realistic and thorough. At this stage, the decompression mechanism itself is treated as a black box, focusing solely on its input and output behavior without delving into the internal workings of the algorithm.

To manage the input of the compressed file effectively, the testbench employs a controlled approach. It initiates the process by activating the write enable signal, signaling the system to start accepting the compressed data. Once the entirety of the compressed file has been fed into the system, the write enable signal is deactivated, marking the end of the input phase.

Parallel to the input process, the system commences the output phase as soon as the first bytes of decompressed data are ready. This phase is characterized by a continuous output of decompressed data, culminating in the reconstruction of the original file. The system's ability to start producing output while still receiving input exemplifies the efficiency and effectiveness of the decompression process, highlighting its potential for real-time applications.

The culmination of the testbench's operation is signified by the detection of the EndMark, a predefined marker indicating the end of the compressed data stream. This marker is essential for determining the completion of the decompression process, ensuring that the system has successfully processed the entire input and produced a faithful reconstruction of the original file.

6.2.3 Benchmark

Benchmarking plays a pivotal role in evaluating the efficacy of compression algorithms. The Canterbury Corpus[10], utilized in this project, stands as a cornerstone for such benchmark assessments. Established in 1997 as an evolution of the Calgary Corpus, the Canterbury Corpus offers a meticulously curated collection of files specifically selected for their representativeness of typical data encountered in compression scenarios. This assortment of files is instrumental in gauging the performance of both existing and novel compression methods.

The Canterbury Corpus encompasses a diverse range of data sets, including the historically significant Calgary collection and the Large Corpus. The latter is particularly advantageous for testing compression algorithms that require substantial data volumes to reach optimal performance levels. Additionally, the corpus includes specialized collections tailored to various file types, thereby offering a comprehensive framework for comparative analysis across a broad spectrum of data compression techniques.

The selection process of the files within the Canterbury Corpus was underpinned by the objective to mirror the average results yielded by prevalent compression algorithms. This approach was predicated on the premise that files producing "typical" outcomes with established algorithms would likely elicit similar results with emerging methods. The rationale and methodology behind the compilation of the Canterbury Corpus are detailed in a paper presented at the Data Compression Conference (DCC) in 1997. This publication elucidates the challenges inherent in identifying "typical" files for such a corpus, underscoring the meticulous consideration and effort invested in assembling this benchmarking tool.

A distinguishing feature of the Canterbury Corpus is its permanence; the collection is designed to remain unchanged to ensure its utility as a reliable benchmark for future compression algorithm evaluations. This constancy allows for the consistent comparison of compression techniques over time, facilitating an objective assessment of progress and innovation in the field of data compression.

There are 11 files in this corpus:

File	**Abbrev**	**Category**	**Size** (bytes)
alice29.txt	text	English text	152089
asyoulik.txt	play	Shakespeare	125179
cp.html	html	HTML source	24603
fields.c	Csrc	C source	11150
grammar.lsp	list	LISP source	3721
kennedy.xls	Excl	Excel Spreadsheet	1029744
lcet10.txt	tech	Technical writing	426754
plrabn12.txt	poem	Poetry	481861
ptt5	fax	CCITT test set	513216
sum	SPRC	SPARC Executable	38240
xargs.1	man	GNU manual page	4227

6.2.4 Validation

The validation process involves a meticulous comparison between the original file prior to compression and the resultant file after decompression. To accomplish this, the WinMerge software [11] is used, a tool renowned for its capability to conduct line-by-line comparisons between two or more files. This software facilitates the identification of any discrepancies that may exist between the original and decompressed files, thereby ensuring data integrity.

The comparison process extends beyond a simple textual examination; it also includes a verification of the file sizes. This step is imperative as it serves as a preliminary check of data consistency. Files that have undergone successful compression and subsequent decompression should exhibit identical file sizes, indicating that no data loss occurred during the process. This criterion is fundamental to the validation phase, as any variation in size could signify potential issues with the compression or decompression algorithms.

Upon confirming that both files are indeed identical in content and size, the project can be considered complete and successful. This outcome not only validates the effectiveness of the compression technique but also underscores the reliability of the implementation. Such a rigorous approach to testing and validation is crucial in the field of data compression, where the fidelity of the compressed data is paramount.

6.3 Resource and timing

6.3.1 Resource Utilization

The device that was used for the simulation was from the Zynq7000 product family. The product part is xc7z020iclg400 1L[12].

Clock	Frequency
8 ns	125 MHz

TABLE 6.2: Timing Setup

Resource	Utilization	Available	Utilization %
LUT	609	53200	1.14
FF	369	106400	0.35
BRAM	64	140	45.71
IO	22	125	17.60
BUFG	1	32	3.13

TABLE 6.3: Resource Utilization

Total On-chip power	Junction Temperature	Thermal Margin
0.205 W	27.4 °C	72.6 °C

TABLE 6.4: Power

6.3.2 Throughput and Latency

Latency

In the evaluation of system performance, particularly in compression and decompression algorithms, latency represents a critical metric. This measurement was conducted using a relatively small original file, specifically 39 bytes in size, to assess the system's responsiveness. The observed latency for this file size was recorded at 14 cycles. This figure can be attributed to the sequence of operations required to process the file through the decompression mechanism.

The decomposition of this latency reveals the system's operational intricacies. Initially, the process involves the handling of 4 bytes designated as the magic number, which serves as a file format identifier. This is followed by the processing of 4 frame bytes and an additional 3 frame bytes, culminating in the handling of 1 token byte. Beyond this sequence, there are 2 additional cycles needed before the first byte emerges from the input FIFO (First In, First Out) queue. This detailed breakdown elucidates the steps involved from the moment the first byte is introduced to the system to the point where the decompressed data is available for output.

It is noteworthy that the latency could vary with larger files. This variance is primarily due to the increased number of bytes required to describe the number of literals in the first sequence of the file. While a precise upper limit for latency in larger files cannot be definitively established due to the variability in file contents and structure, the latency remains exceptionally low across different file sizes. This characteristic underscores the efficiency of the implementation, which maintains minimal delay even as file complexity increases.

Throughput

File	Compress Rate	Throughput (MB/s)	Throughput (byte/cycle)
ptt5	83%	18.87	0.151
kennedy	63.5%	30.7	0.245
alice	42.7%	48.1	0.385
plrabn12	32.4%	55.67	0.445

TABLE 6.5: Throughput Results

The throughput was calculated based on four files that have different compress ratio. The throughput was calculated using this formula:

$$\textbf{Throughput} = \frac{\text{Compressed file size}}{\text{Total decompress time}}$$

The analysis derived from Table 6.5 of the LZ4 decompression testing indicates a nuanced relationship between the speed of decompression and the compression rate. Initially, one might infer that the throughput of the decompressor directly correlates with the rate of compression. However, further examination reveals that this interpretation does not fully capture the dynamics at play.

When exploring the varying speeds at which data decompresses, several key factors come into play, each influencing the process in unique ways. One crucial element is the data structure and its repetitiveness. For instance, text data that exhibits high levels of repetitiveness, such as in the case of the ptt5 dataset, generally compresses efficiently. However, the process of decompressing such data might not be as swift as it is for data structured in more natural language forms. This slowdown is attributed to the additional effort required to unravel the complex patterns of highly repetitive data.

The size of the files in question also plays a significant role in decompression speeds. Smaller files are typically expected to decompress quicker due to the lesser volume of data needing processing. Nonetheless, this isn't a hard and fast rule, as demonstrated by the 'plrabn12' poem. Despite its larger size, it showcases superior decompression speed. This anomaly suggests that the content and structural characteristics of data can have a more pronounced impact on decompression speed than the file size alone.

Moreover, the compression rate is another vital aspect to consider. A higher compression rate usually indicates that the decompression algorithm must execute a greater number of operations to fully restore the original data, potentially leading to slower decompression speeds. Yet, the ptt5 dataset, which boasts the highest compression rate among the examples, does not exhibit the slowest decompression speed. This observation hints at the presence of other influential factors beyond just the compression rate.

The inherent complexity of the data being decompressed further affects the speed of the process. For example, the binary data found in the 'kennedy' Excel file might pose more challenges during decompression compared to the straightforward plain text of the 'alice' dataset. Interestingly, 'alice' decompresses more rapidly, suggesting that the LZ4 algorithm, known for its speed optimization, processes plain text more efficiently than binary data.

The efficiency of the decompression algorithm, particularly LZ4's adeptness with plain text, sheds light on why datasets like 'alice' and 'plrabn12' experience faster decompression times. These datasets, primarily consisting of text, are decompressed more swiftly than 'ptt5', which contains potentially artificial patterns, and 'kennedy', which includes a mix of binary data and formatting. This efficiency underscores LZ4's capability to adeptly handle varied data types, allowing for quick decompression of even complex files like the 'kennedy' Excel file. The file's size and compression rate notwithstanding, LZ4's proficiency in managing a blend of binary and text data ensures its decompression speed surpasses expectations.

6.4 Comparison

Hardware	Speed	Benchmark
H100 PCIe	43.79 GB/s	Silesia Archive
A100	34.13 GB/s	Silesia Archive
A30	22.07 GB/s	Silesia Archive
A10	26.59 GB/s	Silesia Archive
Xilinx	1.1 GB/s	Silesia Archive
AMD	443 MB/s	Silesia Archive
CPU (average)	62.79 GB/s	Silesia Archive
This Work	41.67 MB/s	Canterbury Corpus

TABLE 6.6: Work Comparison

In the above figure 6.6, we observe a comparison of data compression speeds across a variety of hardware platforms, juxtaposed with the results from this work. An initial examination indicates that the compression speed of this project trails behind the industry benchmarks. However, this outcome is not unforeseen given several key factors that influenced the project's performance.

This project was conducted within the academic scope of a diploma thesis, which inherently entails certain limitations. Notably, the project was not furnished with the sophisticated optimizations typically employed in industry-level hardware designed for data compression tasks. Optimizations, such as algorithmic refinement and system tuning, are crucial for enhancing performance and were not within the scope of this thesis due to its educational and demonstrative nature.

Furthermore, the absence of parallel computing techniques in this project is a significant factor contributing to the observed speed discrepancy. Parallel computing is a potent strategy to amplify throughput, allowing simultaneous data processing that drastically accelerates performance. This technique is a staple in industrial contexts where speed is paramount, but its implementation can be complex and resource-intensive, often beyond the practical constraints of an academic project.

The benchmarks utilized for comparison in the industry derive from the use of the Silesia Archive, a common data set known for its challenging compression characteristics due to the diversity of its content. This corpus is widely adopted in the industry to stress-test and evaluate the performance of compression algorithms. Conversely, the benchmarks for this project employed the Canterbury Corpus, a different set of data with its own unique characteristics and challenges. The dissimilarity in data sets must be acknowledged as it can lead to variance in compression speed due to factors such as data redundancy and complexity.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In the development of this project, the primary objective was to construct a decompression system from the ground up, capable of accurately reconstructing the original file while optimizing both speed and energy efficiency. The endeavor was rooted in the challenge of balancing high-speed decompression with minimal energy consumption, a pivotal concern in the realm of data processing and hardware design.

The architecture of the decompression system was meticulously designed to ensure that it operates with exceptionally low latency, thereby facilitating the rapid reconstruction of the original data from its compressed state. This attribute is critical in applications where time efficiency is paramount, such as real-time data analysis and high-speed communication systems. The system's ability to deliver decompressed data promptly without compromising on the accuracy of the output underscores the effectiveness of the underlying algorithm and the precision of its implementation.

Furthermore, the project placed a significant emphasis on energy efficiency, a crucial factor in the sustainability and cost-effectiveness of hardware systems. By achieving low energy consumption, the decompression system not only addresses the environmental impact of technology but also reduces the operational costs associated with energy usage. This aspect of the design is especially relevant in the context of large-scale data centers and embedded systems, where energy efficiency directly correlates to operational sustainability and financial viability.

The outcome of this project was remarkable, yielding a decompression system that not only reproduces various files with fidelity to their original state but does so with minimal latency. The system demonstrated a commendable capacity to maintain low overheads, thereby ensuring that the additional computational resources required for decompression do not detract from the overall performance and efficiency of the system. This balance between speed, energy consumption, and accuracy in the decompression process represents a significant achievement in the field of data compression and decompression technologies.

By adhering to rigorous design principles and leveraging advanced algorithmic strategies, this project contributes valuable insights into the optimization of decompression systems. It exemplifies how targeted engineering efforts can overcome the inherent challenges associated with high-speed data reconstruction while adhering to stringent energy efficiency standards, thus paving the way for future innovations in hardware design and data processing methodologies.

7.2 Future Work

- Checksum development

  A crucial part of the LZ4 decompression is the checksum which in many stages ensures the integrity of the file that is being recreated. The checksum is generated in the compress process, is encapsulated in the compressed file and is used in the decompression process for data integrity. This security level is pivotal in ensuring that nothing was missed out, or a portion of the information was lost during the procedure.

- Error handling

  Many situations can be met where the procedure will have to be stopped and handled. This situations are errors that might appear during the interpretation of the compressed file. Some examples of these scenarios are the offset to be equal to zero or the total number of literals to be less than the number that was just received. Another problem could occur if for any reason bytes where missed during any moment of the procedure. This would drive the system in potential never ending situations or a mistaken creation of the file. As long as these problems can be perceivable during the decompression there is a need of real time handling of these errors. This error handling could highly increase the system integrity.

- Download and run in FPGA

  Another future situation that could add to the results could be the download of the code to a FPGA and run on the spot. This could offer a real size of the resources that a decompressor would need. Always a real time use of a project can give many results that the simulation is not offering.

References

[1] "LZ4 - Extremely fast compression". In: (). URL: https://lz4.org/.

[2] "LZ4 GitHub repository". In: (). URL: https://github.com/lz4/lz4.

[3] "OpenBenchMark.org". In: (). URL: https://openbenchmarking.org/test/pts/compress-lz4&eval=7d09b74acec40f5dd2415b5af7d64c344d0a882b#metrics.

[4] "nvCOMP". In: (). URL: https://developer.nvidia.com/nvcomp.

[5] "Xilinx Vitis library". In: (). URL: https://xilinx.github.io/Vitis_Libraries/data_compression/2020.1/source/L2/lz4.html.

[6] "AMD Vitis library". In: (). URL: https://docs.xilinx.com/r/en-US/Vitis_Libraries/data_compression/source/L2/lz4.html.

[7] "The Datapath". In: (). URL: https://tucgr-my.sharepoint.com/:u:/g/personal/ggalatianos_tuc_gr/EcEgSnxQ4CVBio9SLLDSOFoBJd9aAMhu6UpqJZgTjS-gjw.

[8] "Finite State Machine". In: (). URL: https://tucgr-my.sharepoint.com/:u:/g/personal/ggalatianos_tuc_gr/EWcyZEeRkXhHqDH6ZiIhHSABanTxeRG5KSz9utIi

[9] "LZ4 compress example". In: (). URL: https://github.com/lz4/lz4/blob/dev/examples/frameCompress.c.

[10] "The Canterbury Corpus". In: (). URL: https://corpus.canterbury.ac.nz/descriptions/.

[11] "WinMerge software". In: (). URL: https://winmerge.org/?lang=en.

[12] "Zynq". In: (). URL: https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview.