FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	High throughput FPGA implementation of LZ4 algorithm
Student:	Bc. Tomáš Beneš
Supervisor:	Ing. Matěj Bartík
Study Programme:	Informatics
Study Branch:	Design and Programming of Embedded Systems
Department:	Department of Digital Design
Validity:	Until the end of winter semester 2020/21

Instructions

Analyse LZ4 algorithm in term of general principles of lossless compression.

Analyze existing software and hardware solutions, which uses similar algorithms (especially LZ77) and suggest possible optimizations for the LZ4 hardware implementation.

Implement both compression and decompression blocks following these requirements:
- processing data block sizes up to 9000 bytes and minimal latency between individual blocks of data,
- test and/or simulate resulting architecture and perform an experimental evaluation of a data throughput,
- AXI4-Stream with 64-bit width data interface with optional control signals.

References

Will be provided by the supervisor.

doc. Ing. Hana Kubátová, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 20, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

High throughput FPGA implementation of LZ4 algorithm

Bc. Tomáš Beneš

Department of Digital Design
Supervisor: Ing. Matěj Bartík

May 7, 2019

Acknowledgements

I would like to thank Matěj Bartík and Karel Hynek for their moral and professional support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 7, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Beneš, Tomáš. *High throughput FPGA implementation of LZ4 algorithm.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Diplomová práce se zabývá návrhem a implementací kompresní a dekompresní architektury číslicových jednotek určených pro FPGA obvody. Návrh klade důraz na využití v systémech s vysokou propustností a nízkou latencí.

Práce obsahuje důkladnou analýzu kompresních algoritmů rodiny LZ77 a LZ78 pro dosáhnutí optimalizované implementace algoritmu LZ4 pro hardwarovou architekturu. Dále práce popisuje návrh kompresní jednotky v jazyce VHDL. Poslední část práce se věnuje simulaci, testování a experimentálnímu vyhodnocení navrhnuté jednotky.

Navhrnutá architektura byla úspěšně implementována, simulována a otestována pomocí Ethernetového rozhraní na FPGA platformě od firmy Xilinx

Klíčová slova Komprese, Decomprese, LZ4, Nízká latence, 1Gbit, 10Gbit, IP Core, FPGA

Abstract

This master thesis presents a design and an implementation of hardware compression and decompression units designated for use in FPGAs. The design focuses on high-throughput and low latency systems.

The thesis contains a thorough analysis of LZ77 and LZ78 families of compression algorithms for implementation of optimized LZ4 algorithm for hardware architecture. Then it describes the design process of the compression unit written in VHDL. Lastly, it concerns with simulation, testing, and experimental evaluation of the designed architecture.

The architecture has been successfully implemented, simulated and tested using Ethernet interface on the Xilinx FPGA platform.

Keywords Compression, Decompression, LZ4, Low latency, 1Gbit, 10Gbit, IP Core, FPGA

Contents

List of Figures

List of Tables

Introduction

With the increasing popularity of video streaming services such as Youtube or Netflix, multimedia traffic has become the greatest part of the worldwide network data transfers [1]. It is challenging to satisfy the demand of every user with the use of compressed multimedia traffic and it would be nearly impossible to do so with the uncompressed data.

The main principle of compression is to reduce the amount of data needed to be transmitted. Lossy compression does not require the reconstructed data to be identical to the original data, unlike a lossless compression.

The same analogy can be used in hardware architectures, where the limitations are again in data transfers between multiple hardware units. In these transfers, we have to retrieve the data without any data loss unlike the multimedia transfers, where some parts of the information about every pixel can be omitted because the difference between similar pixels is unrecognizable for the human eyesight. This is where the lossless compression can be used for increasing the overall performance of the hardware architecture by lowering the demand on the weakest link in the architecture by using more of the computation resource of the modern technologies.

Numerous well-known compression algorithms could be used for these purposes. For example, the Ziv-Lempel coding or the Burrow-Wheeler transform based coding [2], however, these algorithms have a large latency and overhead, therefore they cannot be used in low latency systems. On the other hand, the LZ4 compression algorithm is designed for high-speed compression and can be implemented with minimal latency by trading the compression ratio for increased compression/decompression speed. Due to these features, LZ4 can be used for a design of a logic core which might be suited for application in low latency systems.

Chapter 1

State-of-the-art

In this chapter, there are described all of the principles, architectures and the technologies, which are required for the understanding each step during the design and development of the low latency compression/decompression unit.

1.1 Compression techniques

The compression techniques or compression algorithm are consisting of two algorithms. There is a compression algorithm, which takes an input X and transforms it to X_c which is the compressed representation of X. Size of X_c should be lower than X, however, most of the techniques must be applied to a given format which leads to size increment which can result in $X_c > X$. The reconstruction algorithm uses as input the compressed representation of the data (X_c) and reconstructs the data Y.

We can divide the compression techniques into two groups, which differentiate in requirements on the reconstruction algorithm. The group of lossless algorithms requires $X = Y$ and the group of lossy algorithms which does not have this requirement. The lossy algorithms are widely used in multimedia files, where each file in its original size is unusable by an average consumer. The lossy techniques have generally much higher compression ratio than the lossless one due to the loss of data[2].

One of the performance indicators of a compression technique is a data compress ratio. The data compress ration is defined as the ration between uncompressed data size and the compressed data size.

$$Compress\ ratio = \frac{Uncompressed\ size}{Compressed\ size}$$

1.1.1 Lossy compression

Using the lossy compression techniques involves loss of information, this fact, however, is not an issue in certain cases used for human interaction, instead of

a computer which needs an exact form of the data to operate on. This can be seen on the multimedia information, where a human does not need the exact video or voice, human senses can not tell the difference. This behavior enables to reach higher compress ratio in multimedia compression.

1.1.2 Lossless compression

Lossless compression techniques do not involve any loss of information. The data processed by these techniques can be from its compressed state reconstructed exactly. These techniques are used in an application where we cannot tolerate any information loss. It is used mostly for computer processing such as compression of executables, documents, and others. In the case of a binary file, even small change in the data can cause to these data to have completely different meaning and change the behaviour of the computer processing them.

1.1.2.1 Dictionary techniques

The input data can be preprocessed to find some properties which can be then used to enhance or construct a compression algorithm. The preprocessing can be used to find frequently occurring patterns and using them as a structure called dictionary.

This assumption is based on the data format which the algorithm is processing from a strictly statistical point of view where each part of the data is statistically independent. This is not a very robust solution from the statistical point of view. However, most of the data used in common situation have certain patterns which can be used and compressed into smaller instances.

Dictionaries are used by the compression algorithm to represent repeatedly occurred patterns just by their position in the data instead of the full pattern. The dictionaries can be divided into multiple categories by the way they are constructed:

- Static dictionary - This is the most appropriate approach when dealing with prior knowledge of the input data. (The exact words of the input are known.)
- Adaptive dictionaries - These dictionaries are build using input analysis, these can be then used even on unknown data input with relative success. Most of the adaptive-dictionary-based techniques have their roots in papers by Jacob Ziv and Braham Lempel in 1977[3] and 1978 [4]. The algorithms based upon 1977 article are part of LZ77 or LZ1 family and algorithms based upon 1978 article are part of LZ78 or LZ2.

1.1.2.2 LZ77

The dictionary of the LZ77 is simply a portion of a previously encoded sequence. The compression algorithm is examining the input using a sliding

window technique with a fixed size. The window consists of two parts, a *search buffer* that contains recently encoded sequence and a *look-ahead buffer* that contains the next part of the input stream to be encoded.

The *search buffer* is being searched for an occurrence of a sequence located at the start of the *look-ahead buffer*. The length of the matched sequenced is then calculated. This process can be repeated to find the longest match in the *search buffer* to achieve the highest compress ratio. Some of the successors of the LZ77 are tuning this process to trade the compress ratio for the compression speed and another way around.

The distance between the selected match and the start of the *look-ahead buffer* is called a *offset*. The selected match sequence is then encoded together with informations: *offset*, *match length* and number of words following this sequence. These pieces of information are using by the reconstruction algorithm to reconstruct the original data.

As shown in the original article[3], the LZ77 algorithm is a simple scheme which does not require any prior knowledge of the data input and asymptotically achieves the performance of scheme with full knowledge about the statistics of the input[3].

There are many algorithms based upon the LZ77 scheme, for example, LZSS[5] or DEFLATE. The LZSS further increases the performance of the LZ77 scheme. The DEFLATE combines the LZ77 principles and Huffman coding principle[6]. It is used in popular tools, such as gzip, png image files and zip.

1.1.2.3 LZ78

The LZ77 algorithm assumes that the repeated sequences are closer together. However, if the occurrence period is longer than the sliding window it will not be encoded. On the other hand, the LZ78 algorithm does not use the sliding window as the dictionary, it is keeping an explicit dictionary. This dictionary has to be identical in both encoder and decoder. The dictionary can be constructed by both sides, stored as part of the compressed information or can be separated and stored prior to the data transfer to reduce the amount of information needed to be transmitted.

Additional improvements to the LZ78 scheme were made by Terry Welch in 1984 which is called LZW [7]. The main improvement introduced by the LZW was the dictionary construction method. The method constructed the entries in the dictionary in streaming fashion instead of inserting the whole match. It is widely used in GIF image format and can be optionally used in TIFF or PDF files.

Magic num.	Descriptor	Data block	(...)	End mark	Content Checksum
4 bytes	3-15 bytes	upto 4MB		4 bytes	0-4 bytes

Table 1.1: LZ4 Frame structure

1.1.2.4 LZ4

LZ4 is a member of the LZ77 family which focuses on speed by lowering the compress ratio[8]. Authors of LZ4 states that by the benchmarks it is the fastest compression and decompression algorithm from the commonly used ones[9].

Format of LZ4 is defined by the structure which consists of *frames*. The frame structure depicted in (Tab. 1.1) is composed from a magic number, frame descriptor with information about the inner format, *data blocks*, *end mark* and optional *content checksum*. These frames can be concatenated inside of the data source, this feature allows to append to existing compressed file another frame without the need to re-frame it.

The *data blocks* consist of the *block size*, *sequences* and a optional *block checksum*. Additionally, the highest bit of 4-byte *block size* is used as a flag for the uncompressed blocks. These blocks are present in the format for encoding uncompressed data with minimal overhead, otherwise there would be significant overhead by the sequence encoding.

The Compressed *data block* is composed of the *sequences*, it is a structure of *literals*(not-compressed bytes), followed by a *match copy*. It contains additional information about the length of both the *literals* and the *match copy*. It also contains the *offest* from which should be the *match copy* taken.

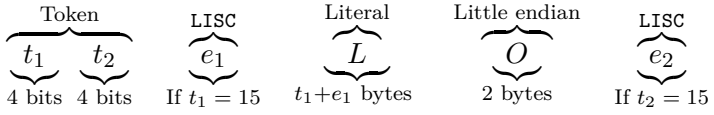

Figure 1.1: LZ4 Sequence structure

Source: [10]

The *sequence* structure depicted in (Fig. 1.1), starts with the *token*, which is a one-byte field separated into two 4-bit fields. The higher field t_1 is used for storing the length of literals. For values higher then 15 it uses a Linear small-integer code(LSIC) e_1. The processing scheme of the LSIC format is depicted in (Fig. 1.2), it loads another byte after *token* adds it to the length and checks if the value of the read byte is equal to $0xFF$ this indicates that there is an additional byte of the code after the read one and process repeats.

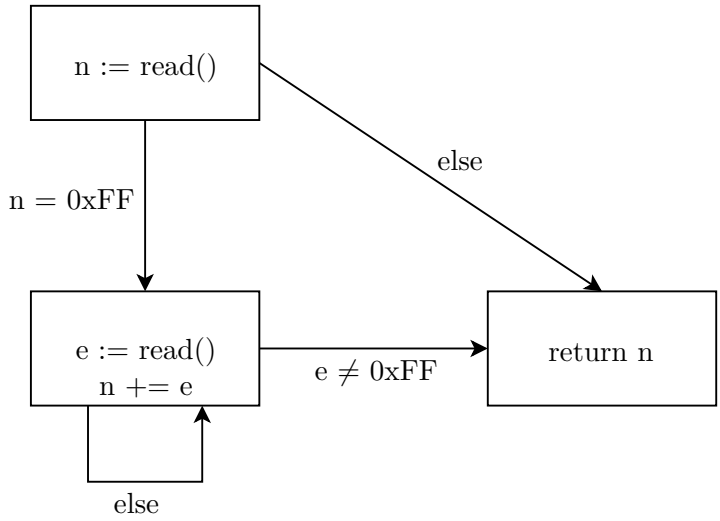

Figure 1.2: LSIC - Decoding flowchart

Source: [10]

This allows LZ4 to keep the number of bytes used by the sequence minimal in case of a short match. the literals L are placed after the LSIC value which determines their length. Then there is the *match offset* O which is taken from LZ77 family and the *match length* LISC value e_2 which is used with lower part of the *token* t_2. Additionally to the calculated *match length* value is added number 4 because shorter matches do not introduce any compression into the format.

The decoding algorithm is processing the compressed input in the following order: It loads the *token*, which t_1 determines whether it should decode the LSIC e_1 of the *literal length*. The token also determines whether the sequence contains any literal bytes, which should be copied to the front of the decoded stream. Afterwards, it loads the *match offset* O which points to already decoded stream. Then it again determines by the token t_2 whether is should decodes the LSIC e_2 of the *match length*. Then it uses the *match offset* to copy bytes from already decoded stream to the front of it. This is where the compression takes effect and actually reduces the size of the input data. This process is also called deduplication and it is depicted in (Fig. 1.3).

It is obvious that the decompression of LZ4 format is simple, very efficient and easy to implement. That is the reason why the decompression speed of LZ4 format is far beyond other compression algorithms. The main focus of this thesis is to construct a compression unit because it is the more complex part of the LZ4 algorithm.

The most computationally expensive part is finding the matches to encode in LZ4 compression algorithm. This can be done by using multiple principles

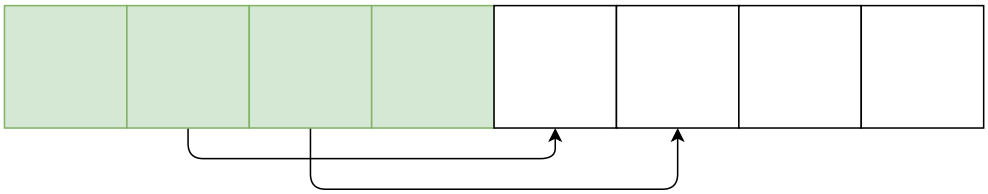

Figure 1.3: Deduplication principle diagram

Source: [10]

which effects the overall speed, latency and the compression ratio of the algorithm. This task is handled in a unit called *Match search unit* in this thesis. The official LZ4 reference design is using a hash table whereas the key to this data structure is used 4-byte set of data for finding matches which have length equal or greater than four bytes. The referenced design is software design which uses advanced features of the hash table to resolve collisions introduced by the use of a hash table. It also uses an advanced technique to choose the selected match from multiple candidates. Using this method LZ4 can encode the input in a single pass over the input data. There is a variant of LZ4 which operates by fully analyzing the data before encoding it in multiple passes which greatly increases the compress ratio of input data, however, it also greatly decreases the compression speed. This can be used for the static data resources which are compressed once and served used many times. This also adds extra latency into the compression algorithm and mainly does not guarantee minimal latency.

1.1.3 Comparison of different compression techniques

One of the most challenging tasks during the development of every compression algorithms is a performance measurement. This measurement heavily depends on a data set which is used for evaluation. Therefore for lossless comparison of multiple compression techniques exists multiple data compression data sets called corpora such as Silesia[11], Cantebury[12] and Calgary[13]. These corpora consist of files with various properties and types. These corpora are designed to compare performance between multiple compression algorithms for the same data input. For example English text, executable files, HTML files, XML file, database files even images are in these sets. This can give an insight into the compression ratio performance of a given technique. However, these corpora are not representing all of the cases which the technique encounters during its real production cycle. The different types are preventing from making an optimized algorithm for given corpora.

The situation is even worse for lossy compression techniques. There are no corpora which could be used to compare their performance. The main

reason is that the decompressed output differs from the compressed one. This fact causes a subjective impression of the output of a given lossy compression technique. Which means, we cannot compare these techniques using exact methods available in modern computers. One of the techniques to compare the lossy compression techniques is the ABX test. The ABX test is a general statistical tool to compare two choices and find detectable differences between them using human subjects to make the rating of both choices.

1.2 Pattern matching for Ziv-Lempel

The first and easiest method for matching pattern located at the beginning of the *look-ahead buffer* is a linear search. The linear search is comparing each of the N positions in the *search buffer* with the pattern. It has a complexity of $O(NM)$ where M is the maximum length of the pattern, however, in most cases, the linear search is much quicker and finishes in $O(M + N)$[14].

Another method designed by Knuth, Morris, and Pratt[15] known as the KMP algorithm is using information about the match to skip some of the searched position. This is done in a moment of pattern match fail. It is possible to choose the next position to check based on the repetitions within the searched pattern. This algorithm does not outperform the linear search in a text such as English text. However, in moments when it is used to search in a text where a lot of repetitions occurs it greatly increases the performance of the pattern matching. The behavior of KMP is $O(M + N)$ in the worst case.

One of the most advanced patterns matching algorithms is an algorithm designed by Boyer and Moore known as the Boyer-Moore algorithm. It is based upon the KMP and further expand the idea of preprocessing the matched pattern. It introduces a new idea of comparing the tail of the pattern against the *search buffer*. This allows a further increase in the number of skipped positions. In the original paper, it has worst-case behavior of $O(M + N)$ only in case of searching for a pattern that does not exist in *search buffer*[16]. Richard Cole proved proof the upper bound of $3n$ comparisons in the worst case in 1991[17].

All of the advanced algorithms described above perform better than the linear search for larger values of N and M. However for smaller values of M the advantages of these algorithms are lost.

1.2.1 Data structures for pattern matching

Another approach of finding patterns in the *search buffer* involves the development of data structures to index the data inside the *search buffer* and should allow fast identification of matches. They should also allow us to insert and delete elements which should allow us to move the *search buffer* as a sliding window.

Thus three operations are performed on the data structure:

1. Insert
2. Delete
3. Search

It is critical for a fast pattern matching algorithm to use these structures for achieving the fastest available operations. Some of these operations can be merged by special properties of advanced data structures.

1.2.1.1 Trie

One of the most iconic data structure to store strings of data is a Trie also called digital or prefix tree. It is a tree with where each nodes stores information about pattern end and each edge contains information about the character it represents. A path within Trie consists of a series of edges and a node with the end mark. The ordered set of these edges then represents the pattern made of their assigned characters. This allows for a very efficient way of storing patterns which have common prefix as it is depicted in (Fig.1.4). It also allows implementing the search operation in $O(M)$. This is accomplished simply by traversing the Trie by each character contained in the pattern. To delete a pattern from the Trie it simply removes the leaf which contains the end mark of the pattern if the parent node has a single child it is removed also.

In the case of storing a large number of varying patterns the usage of space is not ideal and can be wasteful. This can be solved by modified structure Patricia trie[18] which is based upon the Trie. It represents the Trie in a more compact way by merging nodes with a single child with their parent.

A very similar data structure called suffix tree[19], which have increases performance of the insert operation although it decreases the performance of the delete operation. It is used in a variant of the LZ77 compression algorithm called LZR[20]. It utilizes the increased performance of the insert operation and solves the downside of the suffix tree by discarding it and constructing a new one when needed.

1.2.1.2 Hash tables

The hash tables can be used to store information by using a key (hash) generated by the hash function from input data. This can be utilized by storing the patterns of the *search buffer* and processing each block with a single hash table. This can be done for matches of any size if correct hashing function is available.

By utilizing multiple hash tables we can construct a hash table list. This list would be constructed by storing matches of length N in the first one and next would store matches of length N+1 and so on. This allows us in linear

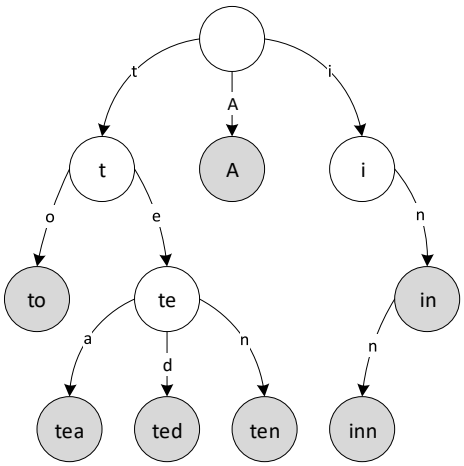

Figure 1.4: Example Trie data structure

time to check if the *search buffer* contains match of a given size. The search algorithm would use the value from *lookahead buffer* with a length of N and on the successful match it would try N+1 and so on. This is a very efficient way of search for patterns it, however, introduces the problem of collisions and overwriting already stored patterns. Both of these can be solved by using either rehashing or by the implementation of hash chains.

1.2.1.3 Binary tree

When it comes to searching algorithms the binary tree will always come into play. The balanced binary tree has an insert, delete and search operation with complexity $O(log(N)M)$, however, there is always a possibility of degeneration into a linked list which have complexities $O(NM)$. They are always possible solutions to this problem by implementing some of the advanced trees such as AVL-Tree (Delson-Velsky and Landis)[21], RB-Tree(Red-Black) or Splay-Trees[22] which tries to keep the depth of the tree minimal. This should prevent the tree from degenerate into a linked list.

1.2.2 Match search unit

Every lossless compression algorithm has a code segment or dedicated unit which has a specific purpose of locating repeated occurrences inside the text. This unit may be called a Match search unit (MSU). MSU is usually optimized for an application in a given compression algorithm. In the case of LZ4 and goal of this thesis, its focus is on finding 4-byte matches in the *search buffer*

by ingesting the top of the *lookahead buffer*. It should be optimized for high throughput, low latency, and constant latency.

For achieving our goal, the lowest latency possible, the main principle which will be used is a single pass, because the multi-pass approach increases latency. The unit has been inspired by the LZ4 reference design which is using a hash table.

1.3 Data structures in a hardware

A data structure is one of the most important things in hardware design. Without such structures, it would be impossible to store and process any large and complex data sample. There are many structures which are carefully designed to fulfill a certain task. Most of them are used to store a piece of information, however, each has a different way of accessing the information. The most simple one is a traditional memory, which uses addresses to choose the specific data to be read or to be written inside the memory bank. These memory types are often used to construct more sophisticated structures such as FIFO (first in first out), LIFO (last in first out), content addressable memory (CAM) or higher-ported memory out of lower-ported memory blocks.

1.3.1 Content addressable memory (CAM)

Content addressable memory, as its name suggests, is a memory which uses as its address a data content. One of the implementations of content addressable memory uses one to one mapping function of data to addresses. If the memory is fully associated then the memory size is equal to the number of data variants which it is able to accept. Each of the data variants has its predefined place (address) in the memory. This feature allows the user to find data in the memory in constant time simply by addressing the memory by the result of the mapping function. That is why the CAM memory is mostly used in applications requiring very high-speed searching capabilities. However, the fully associative memory is very large for the given data set in most application. For example, in case we want to store a piece of 32-bit information in fully associative memory, it requires to have a memory block of size 2^{32} which is almost impossible to implement inside modern hardware or it is very expensive.

When using these memories the application has to be very efficient with every single memory slot. It should utilize every possible slot. This is very unlikely with the use of higher memory sizes because there is only a handful of an application which accepts data from the full range of 32-bit data.

1.3.2 Hash table

Hash tables are a special class of CAM, which are not fully associative. This means there is a subset of a given size which is mapped to the same address space in the memory. The mapping function is called a hash function. Properties of a hash function determine the performance of the hash table. This feature has advantages and disadvantages. The main advantage is a lower memory size than a fully associative CAM memory. The main disadvantage is of a hash table are collisions (depicted in Fig.1.5) which have to be solved by the application or by the additional functions of the hash table. The number of collisions depends on the data set which is used to access the hash table and on the performance of the hashing function. One of the performance indicators of the hashing function is the spread of the input data to the whole memory space.

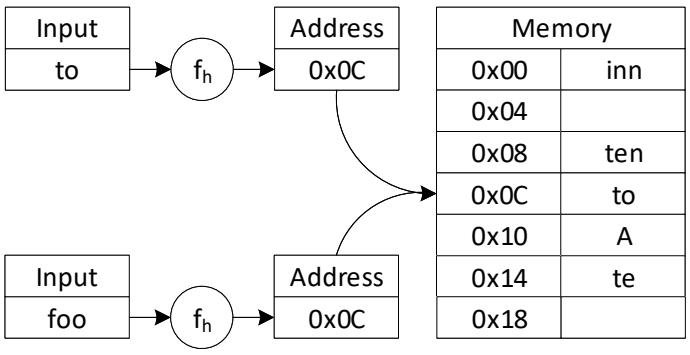

Figure 1.5: Example hashtable data structure with collision

1.3.2.1 Hashing function

Hashing function f_h is a function which accepts an input data D of given size N_D and it calculates an output H_D which is called hash of size N_H. The sizes N_H and N_D can differentiate. In the case of $N_D = N_H$ the function can be used for fully associative memory. However, in most cases is N_H is smaller than N_D this implicates that some of the input data have to be mapped to the same hash. The N_H determinates the size of the hash table, therefore we can use smaller memory to store the data. The subset of the input data which is mapped to the same hash creates collisions. This means that in given time only one element of this subset can be present in the memory.

$$f_h(D) = H_D$$

This function is expected to fulfil some properties or at least approach the ideal [23][24].

- Determinism - This property must be always accomplished each input D has to have exactly one H.
- Uniformity - Every good hashing function have a probability of each H roughly the same. This helps with the spread of the input data across the whole output space. This, however, does not prevent clustering of similar input data.
- Irreversible - Hash function does not have inversion function which would determine for a given H_D original data D. This property is however mostly used in cryptography, which is not a crucial property for hashing function used to create a hash table.

1.3.2.2 Multiplicative Hashing

The multiplicative hashing method is based on the division method, which simply uses remainder modulo M:

$$h(K) = K \ mod \ M$$

The performance of this method heavily depends on the value of M. In most cases use of a prime number as M has been found to be sufficient [25]. The multiplicative hashing is in way generalization of the division method. To understand it we have to imagine working with fractions instead of integers. Let the w be word size of the computer, A an integer constant which is relatively prime to w [25].

$$h(K) = \left\lfloor M\left(\left(\frac{A}{w}K\right) \ mod \ 1\right)\right\rfloor$$

The multiplication is often faster than the division in modern computers and hardware, which leads to significant performance gain[25].

1.3.2.3 Fibonacci hashing

The Fibonacci hashing is based on the value of the golden ratio $\phi = 1.6180...$ which is related to the Fibonacci numbers. The basic principle can be explained by an example: When you walk around 360° circle with steps of given size $360°/n$ you will end up in the same spot after kn steps where k is the number of laps you take around the circle. However, with the golden ratio ϕ you can choose to step by the value of $360°/\phi$ which will result in an infinite number of steps (and laps) and you will never end up in starting position. This can be utilized for hashing because with the large step we will distribute similar numbers across the whole space. This results in using constant $2^b/\phi$

where b is the number of bits used during the computation. The whole hashing function is:

$$h(K) = \left(K * \frac{2^b}{\phi}\right) \gg S$$

S is equal to $b - log_2(M)$, where M is the size of our hashing space which is the power of 2. The symbol $\gg$ is representing binary shift with a size of S bits which causes usage of the higher bits from the multiplication which results in most accurate steps around the circle[26].

1.3.3 FPGA memory

The Xilinx is providing support for memory blocks constructed from a Block RAMs (BRAM), Lookup tables (LUT) or implemented in logic each has its benefits and uses different resources of an FPGA chip. The memory implemented in logic is not restricted by the number of memory ports. However, these memory blocks are up to a few Kb and consume a large number of FPGA resources. This is the reason to prefer BRAMs, they are better and they have much larger storage capacity. The main disadvantage is they have exactly two read/write ports. This disadvantage can be very problematic for parallelization of some algorithms, which require access to a coherent memory space from multiple processing units. The distributed memory which uses LUTs as small memory block is between the memory implemented in logic and with use of BRAM. It is restricted by the number of ports, however, it can have up to four ports. Its sizes are larger than the logic implementation. Very similar support can be found in FPGAs by other manufacturers such as Intel[27], Lattice[28], etc.

1.3.4 Multi ported memory

A multi-ported memory is a memory which is capable of operating multiple ports independently with a data coherency preserved between all of the ports. These memory blocks are either designed in a silicon with this functionality in mind or they can be constructed out of lower ported memory by using principles described bellow[29][30].

1.3.4.1 Principles

There are three fundamental principles used for construction multi-ported memory from a lower ported memory[30].

Replication (1W/nR - 1 Write n Read ports) which is placing n memory next to each other and connecting single write port into all of them. The write transaction is then handled by all of the memory and each has the same data stored inside. This allows using n read ports to access coherent memory space, where each read port is connected to a different memory depicted in (Fig. 1.6).

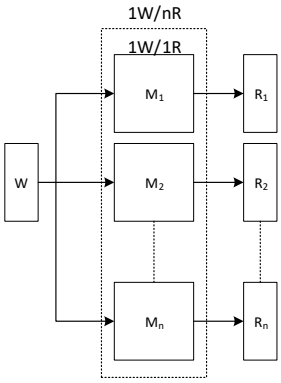

Figure 1.6: Memory replication

Source: [30]

Banking (nW/nR) approach uses n memory blocks next to each other into a single block which then have n Write ports and n read ports this, however, does not provide coherent memory space depicted in (Fig. 1.7).

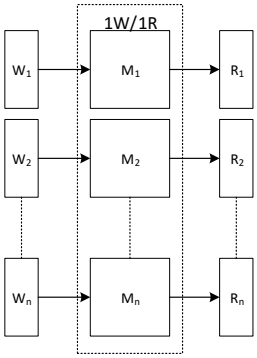

Figure 1.7: Memory banking

Source: [30]

Multipumping (mW/nR) principle uses only a single memory. It introduces new internal clock for the memory which is usually $f_{in} >= max(M*_{out}, N * f_{out})$. The main limitation is determined by the maximum frequency of the used memory. This allows the memory to execute all the requests on every port in sequential order clocked by the internal clock[30]. The memory appears for the slower clock domain as multi-ported memory, which handles

each transaction in one clock cycle depicted in (Fig. 1.10). This principle theoretically does not have a limit with the use of high-performance memory. However, in FPGAs and SOCs the multipumping factor is usually 2, which is still great performance increment. A typical design operates at 200 MHz[31] even thought BRAM resources available in FPGA are capable of functionality with frequency 600 MHz, the logic is not.

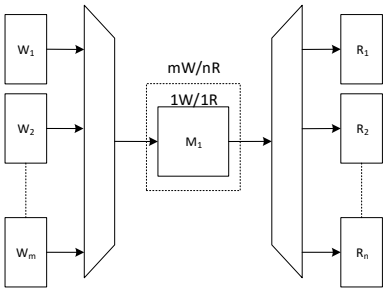

Figure 1.8: Memory replication

Source: [30]

1.3.4.2 Live Value Table technique

One of the traditional implementations of multi-ported memory is using a Live Value Table (LVT), which is a small multi-ported memory implemented in logic which holds a piece of information about latest write port which has accessed the given address. This principle has constant efficiency regardless of the width of the stored data inside the multi-ported memory. It does, however, scale with the number of ports and the memory size of the multi-ported memory. The size of the memory increases in a linear manner with the memory size and in an exponential manner with the number of ports. The information stored in LVT is used during each read transaction by any read port to select an appropriate memory output the latest value written into the memory.

1.3.4.3 XOR Technique

Another technique of the implementation of multi-ported memory is known as XOR technique. It is based upon a xor property $A \oplus A = 0$. This property is able to select a single unique element in a subset where the maximal occurrence of an element is equal to one.

$$(A\ B\ B\ C\ C\ D\ D\ E\ E\ F\ F) \oplus = \ A$$

Where $\oplus =$ represents an operator xor of all elements in the given subset in any order. Using this property we can eliminate all previously written values in multi-ported memory by storing the latest value after applying xor of all the values from other ports memory. During the read transaction, all of the values from all ports are processed by xor operation which results in the latest value written. This technique, however, requires during write operation additional read operation, for standard usage, it adds additional ports required for implementation. In terms of the frequency the xor technique is fairly comparable with LVT technique without multipumping, with multipumping it is falling behind the LVT technique[32]. In terms of used resources, this technique saves a very significant part of the logic resources[32]. This is achieved by omitting the LVT memory and a large number of multiplexers required by the LVT technique for each read port.

1.4 Hardware interconnection methods

One of the most important things in hardware design are interconnected interfaces. These interfaces are designed to exchange information between multiple hardware blocks, which are usually designed by multiple hardware engineering groups. Some of these interfaces are proprietary and disclosed solutions. In case of the commercially available cores, some of these interfaces are public for general usage and become standard for inter-corporation integration. These interfaces are designed to cover most of the interconnect functionality for a certain type of hardware blocks.

1.4.1 Advanced Microcontroller Bus Architecture

Advanced Microcontroller Bus Architecture (AMBA) is a family of interconnect interfaces which were developed by ARM [33]. They are widely used in integrated circuits, Systems on Chips (SOC) and even adopted by FPGA manufacturers. FPGAs often comes with Hard IP cores beside a programmable logic. It is easier for manufacturers to adopt an already existing AMBA interface in their development tools and their proprietary IP cores. The AMBA 3 and AMBA 4 standards have been adopted by the main manufacturers such as Xilinx and Intel in their FPGA IP cores [34].

1.4.2 Advanced eXtensible Interface

The Advanced eXtensible Interface 3 (AXI) was first introduced in 2003 in the standard AMBA 3. The latest version (v4) was introduced in 2014 and it is adopted by Xilinx for use in their latest IP cores. Three basic types of AXI interface are existing called AXI–Full, AXI-Stream [35] and AXI–Lite. Each variant is designed for a specific application and has advantages and

disadvantages. They cover almost all of the required cases for a hardware block interconnection.

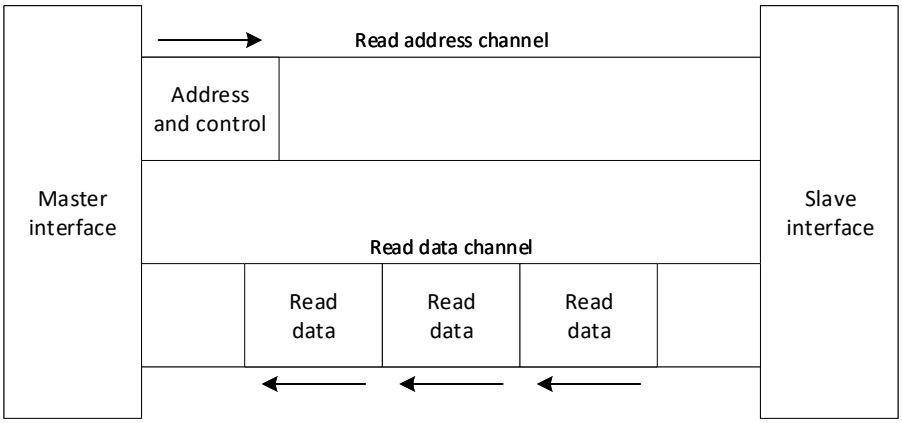

Figure 1.9: AXI4 Read transaction

Source: [33]

1.4.2.1 AXI–Full

This interface is based upon address based protocol with responses. It is a master-slave communication which uses an address to determine the slave to which the master wants to communicate. It provides information about the incoming transaction for the slave. The interface consists of write, read channel, where each channel consists of an address interface, data interface and response interface and additional signals which provide a wider range of functionality which includes cache control, slave locking, quality of service, etc.

Master can be connected to multiple slaves by using AXI Interconnect block, which uses predefined address information about slaves to route incoming transaction by its address to the destination slave. The transaction begins with master exposing read or write transaction address which is routed to given slave. The slave then acknowledges the address by signaling that it is ready to receive/transmit data on the data channel. The master then sends/receives the data through the interconnect. This request and response based communication also allows executing cross clock domain crossing which is usually present in larger designs incorporating multiple IP cores.

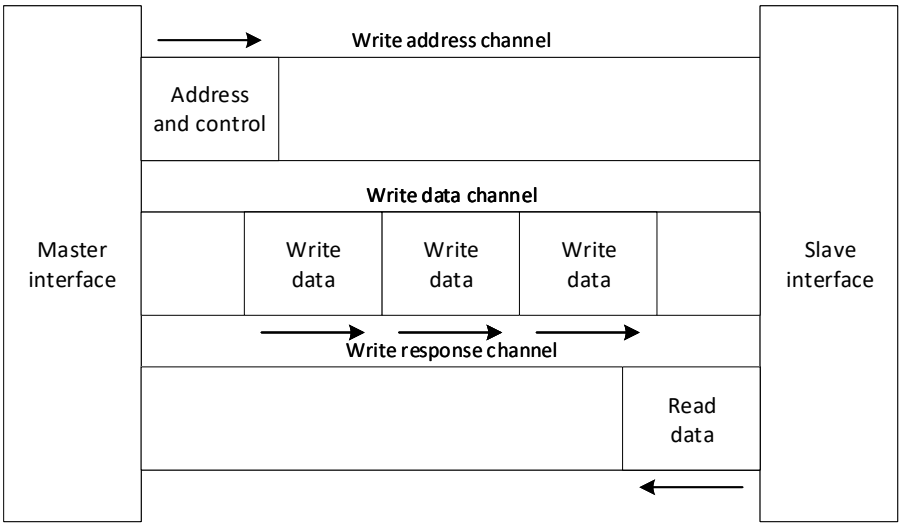

Figure 1.10: AXI4 Write transaction

Source: [33]

1.4.2.2 AXI–Lite

AXI–Lite, as its name suggests, is the lite version of AXI–Full which omits the advanced functionality of AXI–Full. It implements only the write and read channel which is the main functionality. This interface is usually used for simpler blocks which do not have special requirements.

1.4.2.3 AXI–Stream

AXI–Stream is an interface used for IP block which transmits/receive a stream of data, without an overhead of the address handshake. It contains data and control signals (data valid and ready signal). The protocol is quite simple. The master side signals on the data valid signal that the data are ready and slave interface signals by signal ready that it successfully received the data. Thanks to this exchange it can be implemented in most of the widespread IP cores which exchange a large amount of data between each other and it does not require additional signals.

1.5 Hardware optimalizations

Optimizations are one of the most crucial aspects of low latency and high-performance designs. They allow the hardware design to push the limits of the

architecture they are based upon. Some of the basic principles are described in this section.

1.5.1 Parallelism

Parallelism is one of the fundamental and most powerful technique in hardware, which is not commonly using in CPU (Central processing unit) on the elementary level. The parallelism used in modern CPUs is mostly dependent on software. There are several threads in the software domain which shares the resources of the CPU. Even multiple threads can be executed at once in case of a multi-core system. These threads are then synchronized using a complex operation to achieve cache coherency and to be executed in the correct order. The operations do not have to be executed sequentially if they are not dependent on the previous one in hardware design. This is a simple exchange of resources for performance.

A lot of software applications are faster in hardware only because of carefully designed architecture which is only executing required operations in parallel and the synchronization are then precisely planned. In the software design, the CPUs are designed for general usage where each synchronization is very expensive because it solves a large number of issues which are not present in hardware design.

1.5.2 Pipeline

An execution pipeline is mostly known for its usage in modern CPUs. These pipelines are very complex and design to synchronized multiple specialized units. The basic principle is to separate a single operation into smaller steps where each of the steps has its own storage. This allows the hardware design to operate on higher frequency because the combinational critical path is split into multiple parts and shorten. Using this technique the throughput is then increased by the increment of the frequency.

1.5.3 Predictive execution

Predictive execution is a principle[36] which assumes or predicts results of some previous operations which then determines which part of the code will be executed next. Based on this prediction then the CPU executes the code before the initial operation finished. If the prediction was successful then the CPU just gain some performance loss. If the prediction was wrong then the CPU is at its expected performance because in standard behavior it would have to wait until the previous operation finished to continue. This optimization can have in some cycles with one terminal condition large impact on performance because if there would be cycle which would be executed one million times and each time it would wait for the operations to finish when only one time on

the end it will resolve as positive the execution would be dramatically slowed down.

Chapter 2

Design process

Details of the hardware design process of LZ4 compression and decompression block are described in this chapter. This work is based upon previous work, which adapted basic software implementation to work on FPGA platform[8]. Necessary steps to improve [8] implementations are described and properly explained.

2.1 Compression block

Previous design [8] uses the sequential principle of the software implementation of LZ4. This design is suitable for modern CPU, however, it is not ideal for an FPGA design, because it does not fully utilize its resources.

The basic sequential steps are described below and a simplified pseudo-code is available below.

1. Search for a match using a hash table
2. Get length of the match
3. Encode match
4. Search from the end of the match

These four steps (parts of the design) can be optimized by use of the pipelining principle. The main disadvantage is the sequential process is executing only one part at the time. The parallelization of such design is not a trivial process because each step depends on the previous one.

$p_i \leftarrow 0$ ▷ Pointer to the input data
$p_o \leftarrow 0$ ▷ Pointer to the output data
$ht \leftarrow 0$ ▷ Hash table
while $p_i < size_i$ **do**
  $d_{in} \leftarrow read_u32(p_i)$
  $h_{in} \leftarrow f_h(d_{in})$
  $h_{data}, h_{addr} \leftarrow ht[hash_{in}]$ ▷ Retrieve last address and data
  $ht[hash_{in}] \leftarrow (p_i, d_{in})$ ▷ Store current data and address
  if $h_{data} \neq d_{in}$ **then** ▷ Check for collision
    $p_i \leftarrow p_i + 1$
  else ▷ No collision encode sequence
    Find length of the Match
    Encode Token
    Encode Literals length
    Copy Literals
    Encode Offset
    Encode Match length
    $p_i \leftarrow p_i + Matchlength$
    $p_o \leftarrow p_o + Sequencelength$
  end if
end while

2.2 Compression block - Hardware design

The basic design of the unit is composed of four blocks *Input buffer*, *Match Finder*, *Encoder* and *Output buffer*. This design allows us to use the sequential principle and extend it into a fully parallelized version. Main goals for this design are high-throughput and low latency. To achieve these goals we have to ensure that each block has sufficient throughput by itself and that each block has defined stable latency.

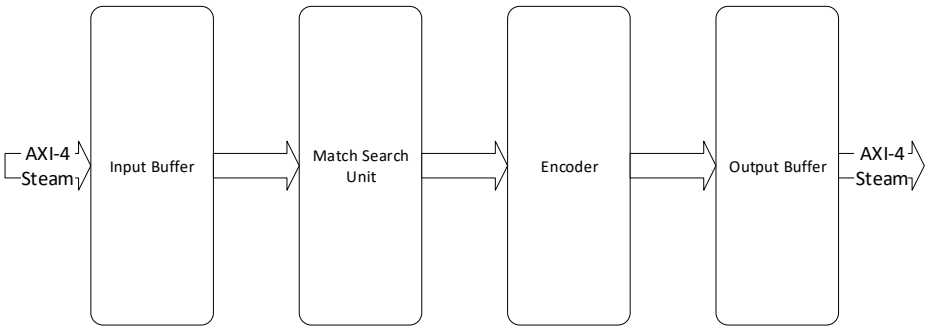

Figure 2.1: Initial hardware design to be optimized

The initial hardware design (Fig. 2.1) which needs to be optimized is consisting of:

Input Buffer - Should implement input interface (AXI4-Stream) with variable word width and it needs to be capable of holding entire data block, which will be processed.

Match Finder - Unit which will process each byte from input buffer together with surrounding bytes to create an entry in the hash table and discover any repeated sequences in the input buffer.

Encoder - Unit which in the moment of a match will find the length of the match and encode it with uncompressed literals to the output buffer.

Output Buffer - Unit which should implement output interface (AXI4-Stream) with variable word width it also needs to be capable of holding entire data block + optional overhead in case of zero compression.

For the proper sequential function of this design each block has to wait until the next block will be ready to receive the data from the previous block. This causes the throughput to rapidly decrease. The basic principle for achieving given throughput is to determine constant which describes the number of bytes which should be ready with each cycle on the output. This excludes time it saturates the pipeline. In this case, it would be ideal to achieve processing speed of 8 bytes per cycle which should allow for a clock of 156.25Mhz throughput of 10 Gbps.

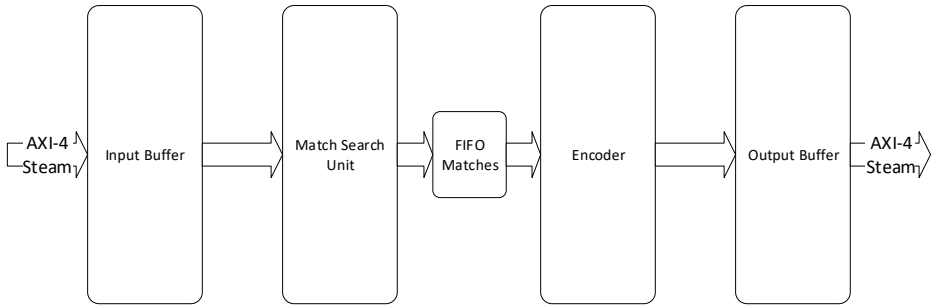

Figure 2.2: Hardware design improved for pipelining

Next phase of improving the design is to remove the dependency of each block to wait on the other this will have an effect of creating pipeline across the whole design. Input and output buffers will be always ready to process data so the only place to solve this problem is between encoder and match finder. A FIFO structure for the found matches is inserted between the encoder and the Match finder depicted in (Fig. 2.2) to solve this problem. This solves

the problem with throughput which was introduced by the previous design, however, pipelining introduces a new problem.

The sequential principle the fourth step adjusts place from which the Match finder continues to search additional matches. However, in this design, this is not possible because it does not wait for the encoder to finish encoding the last match. This causes to create multiple matches from a single match of length > 4.

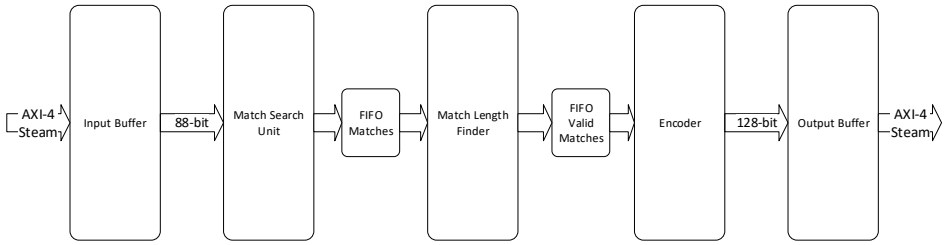

Figure 2.3: Hardware design improved for final pipelining

The problem with multiple matches generated from a single match with length > 4. Can be solved by separating the Encoder block into two blocks. The first will validate and filter matches by their length and by using a greedy strategy of accepting matches to never overlap. This block will produce valid matches with all information needed for encoding. The second block will do only simple encoding for the output buffer. The placement of these blocks in the hardware design is depicted in (Fig. 2.3). The designed unit is utilizing fully the pipeline principle and each block should be able to process 8 bytes per cycle, this should be sufficient to accomplish the requirements of 10 Gpbs throughput.

2.2.1 Latency

A latency of the hardware design is given by two properties. The first one is the LZ4 format, which contains information about the size of the upcoming block in front of each block. This fact limits our latency to the size of the largest block because our unit needs to buffer the whole block before being able to output the size of upcoming it in the correct format with size in front. The second one is the implementation itself, which needs to be fully pipelined and only latency increment must be given by the LZ4 format.

The pipeline design should allow for the block to operate independently on the others. This is achieved by the separating FIFOs which signals (Full signal) whether the previous block should stall. This allows for the next block to catch up. The separating FIFOs should be large enough to keep the stalls at a minimum. This pipeline design should accomplish the requirement of the minimum latency from input to output.

2.2.2 Input/Output buffer

The output buffer is fairly simple single ported memory implemented as FIFO with AXI4-Stream finite state machine which allows AXI4-Stream slave to receive data from the FIFO.

The input buffer is implemented as memory as input FIFO with AXI4-Stream finite state machine which allows the master to write data to be compressed into the memory in sequential order with variable data width to assure the data throughput requirement. However, the input buffer also provides multiple read ports for other parts of the design. This is achieved by banking the memory inside the input buffer. This allows us to use as many ports to the input data buffer as the design requires and the only cost are BRAM resources.

The main issue for these buffer is the fact that the LZ4 algorithm is byte oriented algorithm. This results in the need of using byte addressing, however, this feature is not supported by Xilinx BRAMs. BRAMs can address only whole words. A common word width of used buses inside the compression block is 64-128 bits. Due to this special property needs to be manually enabled by using one of two principles.

Two BRAMs are used to create a memory with unaligned read feature. This principle is depicted in (Fig. 2.4) where the output word for the read operation is constructed from two BRAMs where each serves as a source for lower and higher address. In case of unaligned read the lower BRAM is used to fill M lower bytes and the higher BRAM is used to fill $N - M$ higher bytes. This approach is very resource efficient however the output multiplexers are very vast with combination logic which leads to the long critical path and it may cause problems with clock timings.

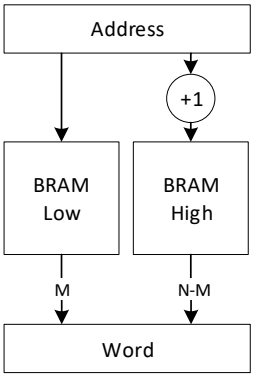

Figure 2.4: Word concatenating from two BRAM memory

Another approach which has a shorter critical path is to create this word

memory from byte memory depicted in (Fig. 2.5). Two multiplexers are used to create output word in a similar fashion as in approach higher. This results in extensive use of BRAM resources which are very valuable in FPGAs, however, it also may allow skipping some of the pipelining stages which in some designs are very problematic.

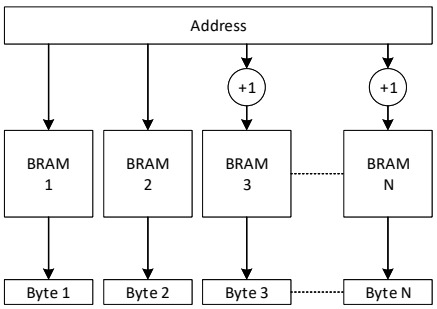

Figure 2.5: Word concatenating from N BRAM memory

2.2.3 Match search unit

The hash table has been chosen out of the above-mentioned pattern matching methods and data structures because it has a very efficient implementation in FPGA. The other data structures do not guarantee constant search latency, which would have a negative influence on the latency of the whole design. The other reason is the inspiration in the official LZ4 implementation which is also using a hash table.

The Basic sequential Match search unit (MSU)[37] is executing the following steps to find a match:

1. Load four bytes of input data
2. Hash four bytes
3. Access hash table with hash
4. Validate content of table with input data
5. Move to next byte

A pipeline principle is used to get a throughput of one byte per cycle which is eight times lower than the requirement. We can simply parallelize this scheme to load simultaneously eight bytes to create eight hashes and to access the hash table with these eight hashes to increase the throughput of this design. The main issue of the design is the requirement of memory block with eight ports.

One port for each of the parallel hashing units. The multi-ported memory is required otherwise the compression ratio would be significantly reduced if each of the hashing units would match only matches for its index. The designed unit is depicted in (Fig.2.6)

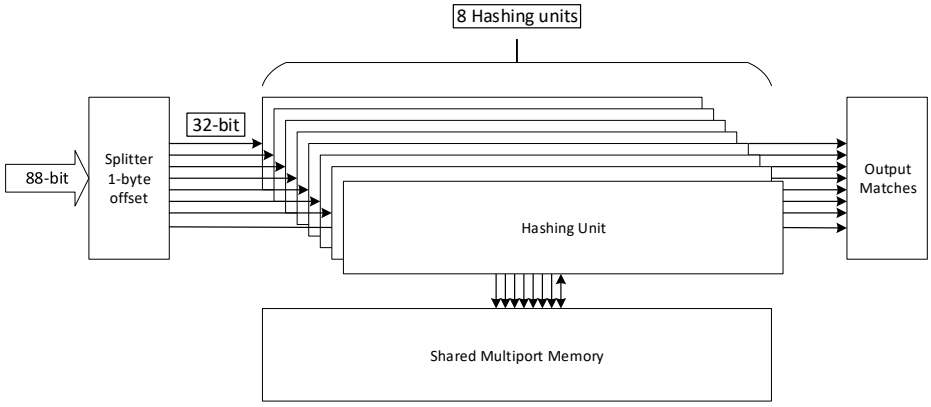

Figure 2.6: Hardware design of the Match search unit (MSU)

The FPGAs DSP48 blocks[38] have been used for the design of a hashing unit depicted in (Fig.2.7) which have the capability to multiply a four-byte number with constant given by Fibonacci hashing principle. These multipliers (hashing units) have a latency of six clock cycles. The match has to be validate if it is a genuine one and not a collision there is shift-register for input data alongside each hashing unit.

The hash calculated by the multipliers is used as an address for the multi-ported memory which has stored inside either input value and its address from previous hashed values or nothing if no value for this hash has not been processed. If the data are equal with currently processed input data both addresses are then used as output for the following block.

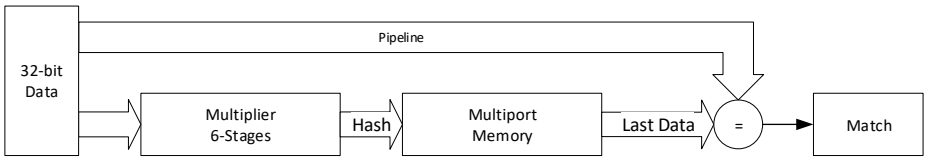

Figure 2.7: Hardware design of a single Hashing unit

The input data for each multiplier is given by the shortest possible match which is 4 bytes and each match can start on each byte. This results in an overlap of data between the hashing units. The overlap (see Fig.2.8) of the

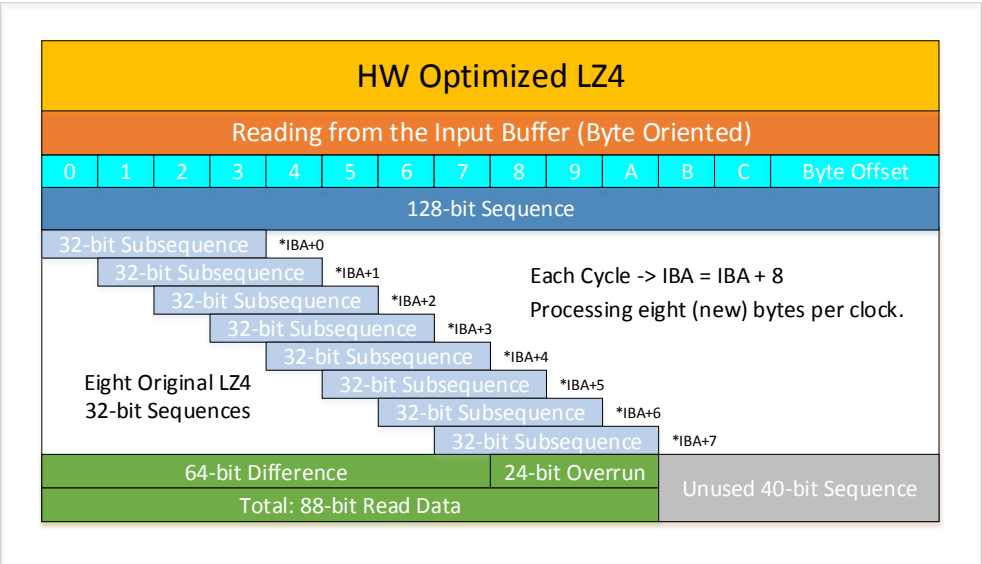

Figure 2.8: The order of processing input data by the hashing units

32-bit sub-sequences which have offset of 8-bits between each other. The total width of used data is summarized to 88-bit which consists of 64-bit of processed data and 24-bit data overrun which will be yet processed. This results in memory with an interface width of 128-bit, which is the closest power of 2. This should ease the implementation in the hardware where implementing non aligned width to the power of 2 could cause resource wasting and in some cases implementation is impossible.

- Source address - Address which is always higher then destination address. It addresses which is currently processed.
- Destination address - Address which contains the same data as the data currently processed.

The size of the (multi-ported) hash memory is directly proportional to the compress ratio of the unit itself. A software model of the hash table is used to determine the number of collisions which occurred inside the hash table to determine the optimal size for the memory for the given data set. The number of collisions is also affected heavily by the data set used for hash table testing.

2.2.3.1 Multi ported memory

The Implemented multi-ported memory N/M (N write ports and M read ports) for the MSU unit is using LVT approach with Replication and Banking principle. The LVT is a design using general programmable logic. It has N ports for keeping information about port which has written the latest value

into the memory. This information is used during each read operation to determine the source of the read information.

Each of the N writes ports during each write operation writes into M memory, each memory represents a possible port which can be used for a read operation. This is necessary in case of addressing the same address with all read ports. The memory should be able to handle all of the read ports from same memory slot. This results in a matrix of a memory block of size NxM. These memory blocks hold the content of the multi-ported memory.

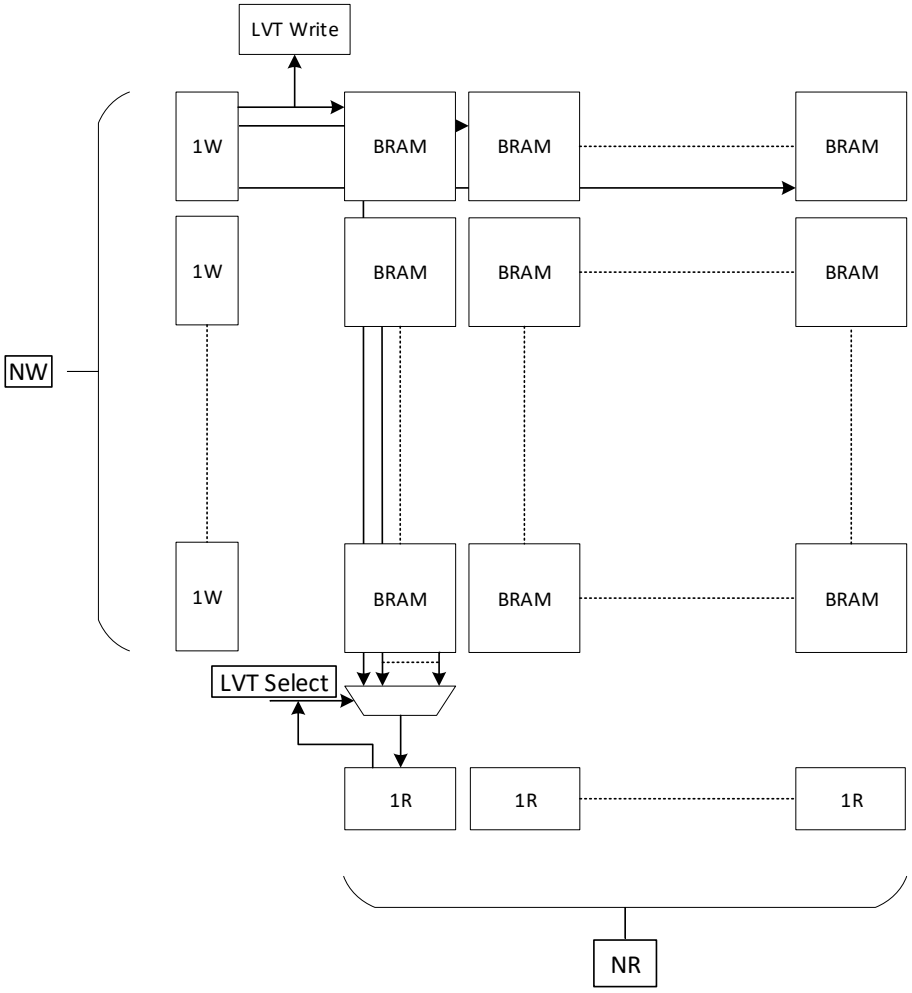

Figure 2.9: Hardware design of the Multiported memory

Each of the M read ports have the option to read from N memory. One of these memory blocks is then selected by a multiplexer controlled by the LVT.

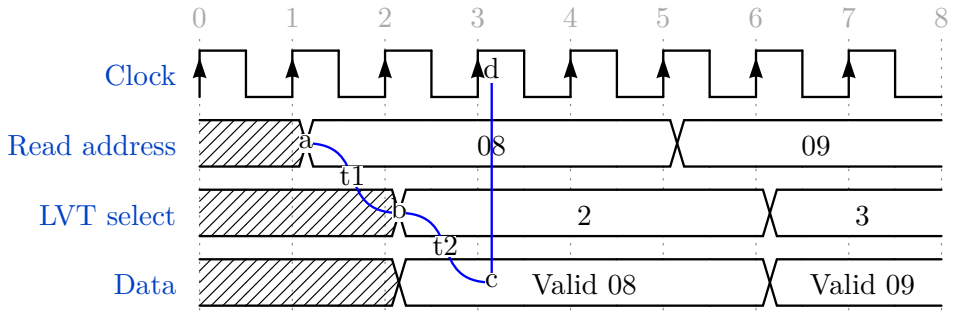

Figure 2.10: LVT Read operation waveform

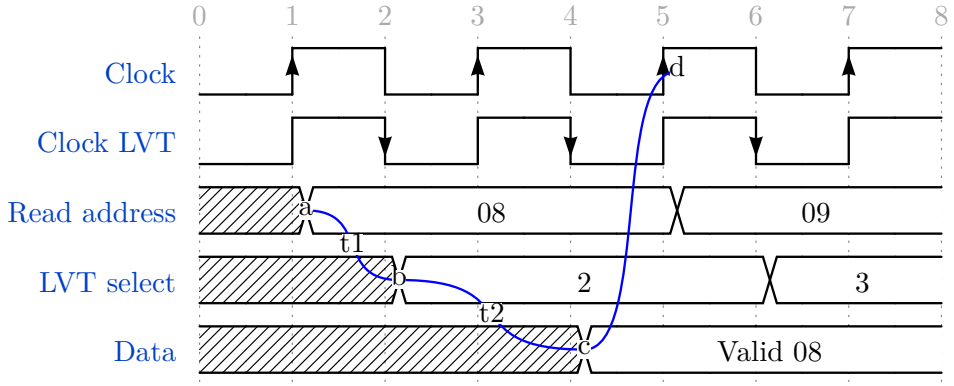

Figure 2.11: LVT Read operation waveform with LVT clocked by negative edge

Control signals from LVT are selected by the requested address of the read operation. The LVT contains the latest index from the N write ports which have written last to the requested memory address.

This operation for a single clocked system will take two clocks depicted in (Fig. 2.10). There is in *a* start of the read operation by giving address 08 this results on the next clock *b* the LVT (it is simple synchronous memory) *t1* select of for output multiplexer which has to process this select with less time than *t2* and finally in *d* the data can be used by the issuer of the read operation. The output multiplexer may cause problems in critical path length and could not meet timing which we set due to its size and complexity.

We can increase time for the output multiplexer of the read operation by using clock source for the LVT which is shifted by 180°. This can be also achieved by programming the logic to work on the negative edge instead of a positive edge. As depicted in Fig. 2.11 we can see that by this principle the *t2* has increased its length by one clock cycle. This should help resolve issues during timing analysis. The critical path has not caused any issues during the implementation of the multi-ported memory.

2.2.4 Match length finder

The main purpose of Match Length Finder (MLF) block is to process matches found by the MSU unit. For each match which does not overlap with already validated matches, the block must find the length of the match. For this purpose, the block has two read ports into the input buffer to allow him to access the data to determinate the length of the match. Features of the match length finder:

Determine length - For given match determine length by searching input buffer and comparing the data.

Cancel current operation - When the block is determining the length of the match should be able to cancel the current operation to stop wasting time for determining the length of a match which will not be used.

Output processed address - Block has to have processed address as its output because the next encoder block has to know which literals (won't be referenced) can already copy to the output buffer.

Throughput - It has to be able to skip matches which will not be encoded to the stream due to longer match overlapping them.

The read delay from input buffer for this block is two clock cycles due to the nature of buffers used inside the block. This can be solved by the use of pipelining to maintain the required throughput. However, if we do not know which address from the input buffer we want to load next due to the delay in the pipeline. These results are used to predict execution to prevent pipeline flushing. The prediction of execution is using simple heuristic which always is expecting the match to be shorter than longer because the longer matches are much less common. Another advantage of this heuristic is the combination with the internal bus of width 16 bytes the longer matches are not limiting the throughput. We always use the next match address from the MSU unit using this heuristic. Following this way we can handle one of the worst cases which are consecutive matches with a length of 4 bytes. This way the 8 bytes processing in a single clock cycle meets the requirements. This would not be possible if we would not assume that the length is shorter then 16.

In the other case of a match being > 16 we have to flush pipeline, this will result of processing 32 bytes in 4 clocks until the operation is resumed. This also allows us to maintain our desired thought put.

Another improvement is the unit can access more than the first element in the FIFO from the MSU. This allows it to process more matches then one at once. This is mostly when the unit is dealing with consecutive matches caused by matches > 4. These matches have to be interactively skipped during the operation of the block.

The canceling of the current operation is done by comparing the current start of the match and current length with borders of previously validated matches and the end of the input buffer space. The currently processed match is discarded in this case. There has to be also implemented filter after this block to prevent race condition which can be introduced by the delay of propagation to these border registers.

The output of this block is again inserted into FIFO to allow pipelining for the following block, which is an LZ4 encoder. Information for this block is validated matches which do not overlap and contains three information:

- Source address - Address which is always higher then destination address. It addresses which is currently processed.
- Destination address - Address which contains the same data as the data currently processed.
- Match length - Length of the match

2.2.4.1 Alternative approach

2.2.5 LZ4 Encoder

The unit which forms the stream of validated matches, currently processed address by MLF and access to the input buffer. Correctly encodes format of LZ4. The unit also is capable of formatting an uncompressed stream which can in some cases be input to the unit. This feature also presents additional advantage into the compression unit which in case of any errors in the previous block will still encode valid LZ4 format the only downside is that without the proper function of previous blocks the compression ratio decreases.

Unit based on the currently processed address from MLF and currently processed valid match will fill its internal buffer with literals which are to be copied into the sequence by speed 16 bytes per cycle. This allows the unit to function using fill flush principle which guarantees throughput of 8 bytes per cycle after amortization.

2.3 Decompression block

The decompression block is by the design simple sequential state machine, which can be implemented in modern computer with throughput in the order of Gbps. This sequential state machine is operating by the following scheme.

1. Load Token
2. Load Literal LSIC
3. Copy literals to the output buffer

4. Load Match Offset
5. Load Literal LSIC
6. Copy Match to the output buffer

By sequential principle, we cannot again achieve high throughput because of many steps required to process a single sequence with the use of the 156.25Mhz. The worst case for this decompression many short sequences where each represents 8 bytes of the data. The most computationally intense part of this process is the decoding of the data this prevents fully pipelining the memory reads.

For a parallel approach, we will utilize the fact that the sequences are independent of each other except the copy match stage. This allows for multiple independent units to function independently and then only serialize the last stage of the decompression process.

Another property which we can utilize is to make the copy parts very high throughput to save cycles for the decoding parts. This will amortize our final decompression throughput too much higher values because the worst case is purely theoretical because the real application data will always have longer matches than 4 and always have some unmatched literals to be copied.

The implementation of the decompression block has not been finished due to the lack of time and the fact that the software implementation is achieving speeds of several Gbps even on low-cost (embedded) CPUs.

Chapter 3

Simulation and Testing

The verification is one of the most important processes during a hardware design cycle. The verification process can be done with multiple techniques. Each of these different techniques is having advantages and disadvantages, thus finding the balance to fit the goal. For verification and development process of LZ4 compression, the unit has been chosen verification by simulation. During this process, we simulate the whole LZ4 block which processes a block of data. The output of this simulation is then compared to the software model of our hardware design. Then the output is also validated against the publicly used decompression program, which should give us exact behavior which we would get on the hardware platform.

3.1 Simulation software

Multiple support application were developed to ease the analysis of errors inside the unit during the development of LZ4 compression unit.

LZ4 Unpacker - Tool written using python to visualize and validate basic LZ4 format. Whenever there is an error inside LZ4 compression output this tool is used to diagnose if the error is in the LZ4 format.

LZ4 Performance - Tool written in C++ which is used as a software model which follows the functionality design of LZ4 compression block. This tool was used to provide insight into changes in the functionality of our unit on compression ratio.

LZ4 Tool - Official program for LZ4 compression and decompression which was modified to provide more detail information on errors during decompression of LZ4 format. The official tool does not provide enough details by default to determine the cause of these errors. There are few exceptions in LZ4 format which was not included in LZ4 Unpacker.

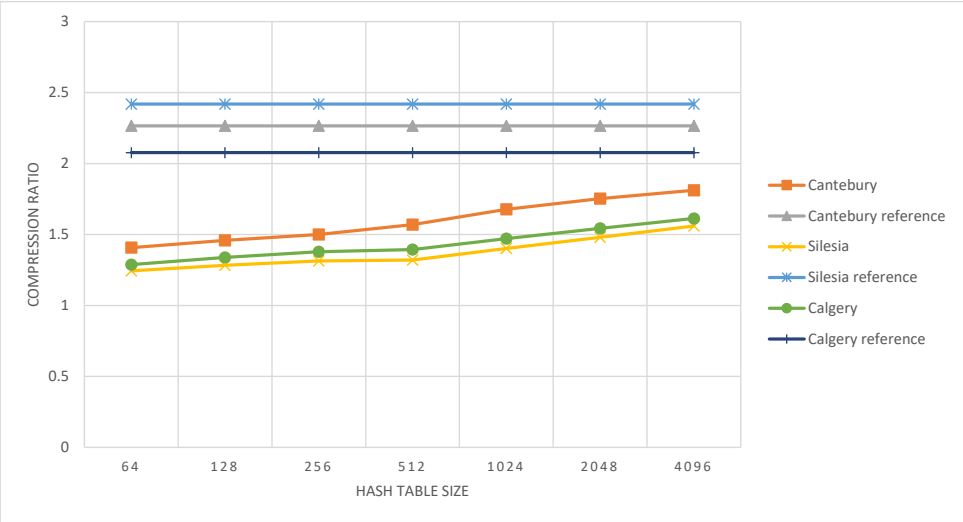

Figure 3.1: The average compress ration with the increasing hash table size

File	64	128	256	512	1024	2048	4096	Reference
dickens	1.02	1.03	1.04	1.04	1.06	1.08	1.11	2.58
mozilla	1.43	1.51	1.57	1.61	1.66	1.71	1.74	1.93
mr	1.24	1.24	1.25	1.25	1.25	1.26	1.27	1.83
nci	2.18	2.25	2.25	2.25	2.54	2.81	3.11	6.05
ooffice	1.11	1.14	1.17	1.19	1.21	1.24	1.26	1.41
osdb	1.03	1.04	1.05	1.07	1.11	1.15	1.21	1.9
reymont	1.11	1.12	1.16	1.16	1.27	1.33	1.39	2.08
samba	1.41	1.52	1.58	1.58	1.75	1.89	2.01	2.79
sao	1	1.01	1.01	1.01	1.01	1.02	1.02	1.06
webster	1.07	1.1	1.14	1.14	1.23	1.32	1.43	2.05
xml	1.33	1.43	1.54	1.54	1.74	1.95	2.17	4.35
x-ray	0.997	0.997	0.998	0.998	0.998	0.998	0.999	1.00

Table 3.1: Compression ratio of LZ4 Performance Tool for the Silesia

File	64	128	256	512	1024	2048	4096	Reference
alice29.txt	1.04	1.05	1.07	1.07	1.1	1.14	1.17	1.71
asyoulik.txt	1.03	1.06	1.07	1.07	1.09	1.13	1.16	1.57
cp.html	1.1	1.12	1.15	1.15	1.28	1.39	1.49	2.06
fields.c	1.21	1.3	1.35	1.35	1.52	1.66	1.75	2.13
grammar.lsp	1.29	1.4	1.44	1.44	1.56	1.66	1.72	1.92
kennedy.xls	1.35	1.36	1.42	1.95	2.16	2.27	2.31	2.74
lcet10.txt	1.05	1.07	1.07	1.07	1.1	1.15	1.19	1.82
plrabn12.txt	1.01	1.01	1.02	1.02	1.03	1.04	1.07	1.47
ptt5	4.1	4.3	4.5	4.7	5	5.1	5.2	5.9
sum	1.21	1.24	1.27	1.3	1.44	1.51	1.57	2.03
xargs.1	1.09	1.13	1.14	1.14	1.18	1.24	1.3	1.57

Table 3.2: Compression ratio of LZ4 Performance Tool for the Canterbury

File	64	128	256	512	1024	2048	4096	Reference
bib	1.04	1.09	1.11	1.11	1.17	1.23	1.29	1.96
book1	1.01	1.02	1.02	1.02	1.03	1.05	1.08	1.47
book2	1.05	1.06	1.07	1.07	1.11	1.16	1.21	1.83
geo	1.01	1.01	1.01	1.01	1.02	1.02	1.02	1.04
news	1.07	1.11	1.14	1.14	1.19	1.25	1.31	1.69
obj1	1.28	1.33	1.37	1.39	1.45	1.48	1.5	1.66
obj2	1.21	1.31	1.4	1.46	1.54	1.62	1.66	2.09
paper1	1.06	1.09	1.1	1.1	1.14	1.19	1.25	1.83
paper3	1.02	1.04	1.04	1.04	1.07	1.1	1.14	1.64
paper4	1.03	1.05	1.06	1.06	1.1	1.16	1.21	1.56
paper5	1.06	1.08	1.09	1.09	1.13	1.18	1.24	1.60
paper6	1.06	1.09	1.1	1.1	1.15	1.22	1.28	1.84
pic	4.1	4.3	4.5	4.7	5	5.1	5.2	5.9
progc	1.13	1.17	1.21	1.21	1.28	1.36	1.44	1.89
progl	1.25	1.27	1.34	1.34	1.47	1.62	1.76	2.65
progp	1.33	1.47	1.53	1.53	1.68	1.82	1.99	2.63
trans	1.45	1.57	1.67	1.67	1.88	2.1	2.29	3.11

Table 3.3: Compression ratio of LZ4 Performance Tool for the Calgery

3.2 Performance analysis

The compress ratio of our unit was tested using LZ4 Performance tool, which is used as a software model implementation of LZ4 compression with equal functionality. This should determine our final compress ratio on given corpora very precisely. The improvement with increasing the size of the hash table used in the Match search unit is depicted in (Fig. 3.1). As reference point is again used official lz4 compression tool which uses much larger blocks and advanced collision resolution which leads mostly to a better compress ratio than our simple approach. The compress ratio also approaches to some stable value which is not further increased by the size of the hash table. In some cases for example file, dickens from Silesia corpus the compress ratio improves with the use of even larger table values 64K from 1.11 to 1.60. Exact values for the measurements are in tables 3.6, 3.1, 3.2.

Simulation setup was made of the LZ4 compression unit. The input of the unit was connected to a block which would load input file from disk storage to the buffer which represents a single block of input data, which then transmitted using AXI4-Stream into the compression unit. The output of the unit was connected to a block which would output AXI4-Stream store to another file on the disk storage. This simulation setup can be used to compress a single file.

An additional property of these block is the ability to operate using streaming files instead of buffering whole files into the operating memory. This results in increased performance during the initialization phase of the simulation using Xilinx ISIM[39].

The simulation architecture depicted in (Fig.3.2) accomplishes two main features. The first one is validation a each block of the hardware design. This can be fairly easily used to track errors to the specified block. The second is loading FIFO buffers from the software domain into the hardware one. This allows skipping the simulation to the problematic block.

The results of the simulation are depicted in (Tab.3.4). The simulation was performed on the Silesia corpus which is the most recent corpus. These results show that the hardware simulation approaches the performance of the software model. The issue with the compression ratio, which is lower then compress ratio of the software model, was traced to the Match Finder Unit. The other blocks worked as expected and it resulted in high throughput and to achieve some compress ratio.

3.3 Testing

The designed units cannot be tested on an FPGA by itself, therefore testing design was developed which uses 1G ethernet interface as UDP packet mirror, where each packet should represent single data block to be compressed or

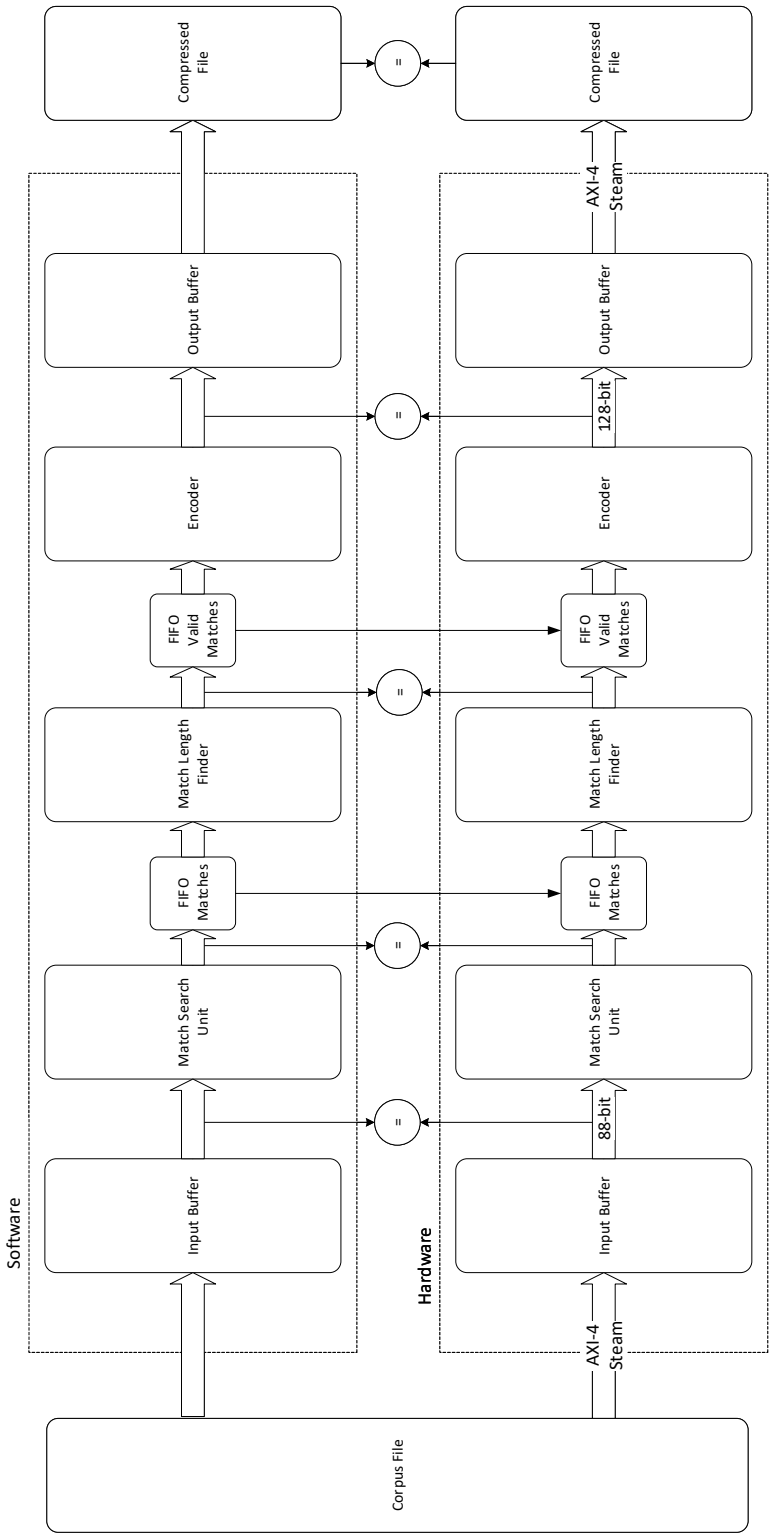

Figure 3.2: Simulation architecture of the compression unit

File	256 Model	256 Sim	Sim Speed(Gbps)
dickens	1.04	1.01	8.43
mozilla	1.57	1.31	4.77
mr	1.25		
nci	2.25	1.30	4.57
ooffice	1.17	1.1	6.27
osdb	1.05		
reymont	1.16		
samba	1.58	1.35	5.58
sao	1.01		
ml	1.54	1.28	5.51
x-ray	0.998	0.999	7.45

Table 3.4: Compression ratio and throughput of LZ4 Compression simulation and the software model

decompressed. This setup was used to validate proof of concept that the simulation is representing the actual function on FPGA, because there is available 1G ethernet UDP/IP core under BSD license at OpenCores[40] and due to previous experience with the 1G ethernet IP cores from Xilinx, which made the implementation easier. The 10G version does not have available open-source IP cores and it is very difficult to implement and it is out of the scope of this thesis.

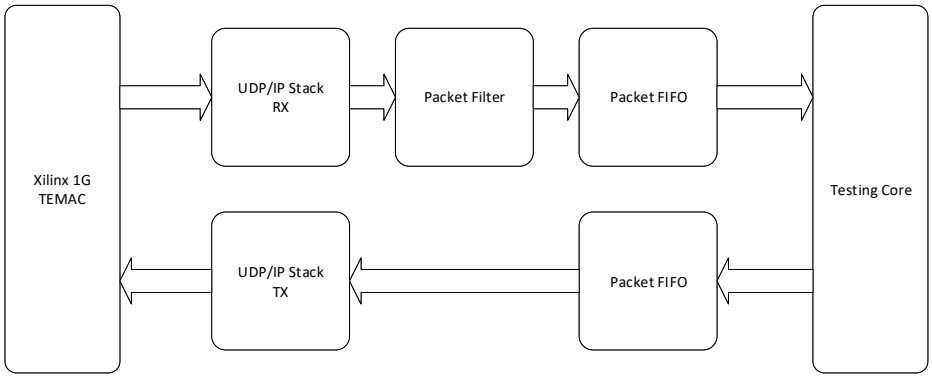

Figure 3.3: Architecture of the testing setup

The architecture of the testing setup consists of Xilinx 1G TEMAC, UDP/IP Stack, Testing core and buffering FIFOS. The data path of the entire architecture is depicted in (Fig. 3.3) and described:

Xilinx 1G TEMAC - RX - Which is used to communicate with the on-

board physical (PHY) layer.

UDP/IP Stack - RX - Core that processes incoming UDP packets and separates the header into multiple signals.

Packet FIFO - Buffer FIFO to ensure packet data continuity, which is not guaranteed by the TEMAC, which contains no buffer only provides an interface for the physical layer.

Testing IP Core - An Ip core with AXI Stream interfaces, which processes the incoming packet and the result is then saved via output AXI Stream to output FIFO.

Packet FIFO - Buffer FIFO to ensure packet data continuity to the UDP Stack.

UDP/IP Stack - TX - This core is handling communication with the TEMAC to output the data packet with correct UDP header. It also handles the translation of the IP address to MAC address using ARP queries.

Xilinx 1G TEMAC - TX - Which is used to communicate with the onboard physical (PHY) layer.

Address	**Register purpose (32bit)**
0	Input - IP Address
1	Input - MAC Address (32 - 0) bits
2	Input - MAC Address (48 - 33) bits
3	Output - IP Address
4	Output - Destination Port
5	Output - Source Port
6	Filter Port

Table 3.5: AXI–4 Register map of the testing setup

The testing setup has AXI–4 Lite interface for configuration, which is operated in our testing setup by Xilinx JTAG to AXI core[41]. This core is very useful for testing or configuring hardware units without the need to go through the implementation process to set a few registers via the AXI interface. The register map is depicted in (Tab. 3.5).

Each IP core was placed in the place of the Testing IP Core. The files which were processed by the simulations were validated that the testing setup successfully produces identical output. Even the test does not cover the full throughput of the unit it at least demonstrates that the simulation which covers the higher throughput is behaving correctly.

Solution	LUTs	LUTMEMs	Regs.	BRAM	DSP	URAM
My architecture	14076	433	2803	82	32	0
Xilinx LZ4	71934	16333	68201	154	1	48

Table 3.6: LZ4 compression unit comparison of FPGA resource usage

3.3.1 Comparison

The official IP core from Xilinx is implemented using High-Level Synthesis (HLS), which is used for comparison, is using significantly more resources than our design. It features throughput of 1.73 GB/s and average compression ratio of 2.15 on Silesia corpus[42].

The designed architecture was synthesized using Vivado design suite 2017.2[43] and FPGA Kintex UltraScale XCKU040-1FBVA676[44] on board AES-KU040-DB-G by Avnet[45].

3.4 Design improvements

Based on analysis of the test results from testing the finalized design series of design problems were discovered which may be addressed in future work and further optimized. These problems are caused by the nature of the implementation which tries to be as close as possible to the official software reference implementation. The first problem is not maximal match size, this requires the use of Match Length Finder unit which is sequential and has to validate these matches after they are found. This could be resolved by the use of a maximal length limit. This would allow determining the length of the match by the Match search unit, which would instead of storing only four bytes, which are currently used for match validation, more bytes up to the maximal length limit. This would lead to a reduction of the Match Length Finder unit which would be replaced only by match filter which would be implemented by simple max/min function to filter overlapping matches. This would lead to great throughput increment, however, this has also impact on the compress ratio of the unit and the Match search unit would also consume more of BRAM resources. This would be most likely compensated by the removed pipeline FIFO.

The effect of limiting the maximal length of the match for the hash table of size 256 is depicted in (Fig. 3.4). It shows the difference between the unlimited match length and the limited match length that the compression ratio is no longer increasing for matches longer than 64. The sufficient length is 32 or even 16 for the other files, this could be generic design parameter specified during implementation to fit the needs of the system. Lowering the maximal length it would save FPGA resources and increase would increase the compression ratio. The most interesting fact is in some cases the limitation improved the

File	4	8	16	32	64	128	256	512
dickens	-0.03	-0.01	0	0	0	0	0	0
mozilla	-0.48	-0.27	-0.15	-0.11	-0.09	-0.09	-0.09	-0.08
mr	-0.17	-0.03	0.05	0.08	0.11	0.13	0.14	0.14
nci	-1.15	-0.81	-0.39	-0.16	0.13	0.14	0.14	0.14
ooffice	-0.14	-0.06	-0.03	-0.02	-0.02	-0.02	-0.02	-0.02
osdb	-0.05	-0.04	-0.03	-0.03	-0.03	-0.03	-0.03	-0.03
reymont	-0.1	0	0.02	0.03	0.03	0.03	0.03	0.03
samba	-0.5	-0.31	-0.16	-0.08	-0.05	-0.03	-0.03	-0.02
sao	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01
webster	-0.12	-0.05	-0.01	0.01	0.01	0.01	0.01	0.01
xml	-0.45	-0.24	-0.11	-0.04	0	0.01	0.03	0.03
x-ray	-0.001	-0.001	-0.001	-0.001	-0.001	-0.001	-0.001	-0.001

Table 3.7: Difference of compression ratio of LZ4 Performance Tool with limitation on the match length and unlimited length for the Silesia.

compress ratio of the data. This is most likely caused by the usage of a greedy heuristic for match selection.

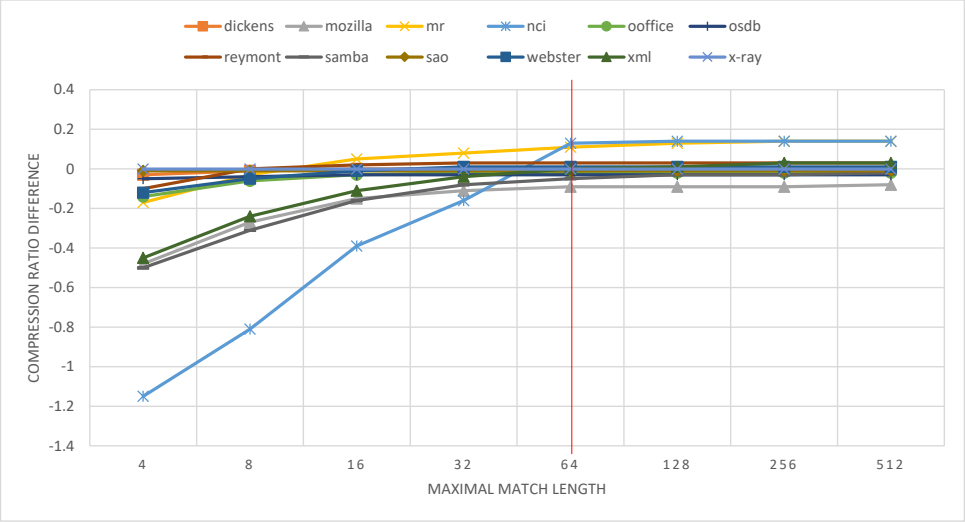

Figure 3.4: Difference of compression ratio of LZ4 Performance Tool with limitation on the match length and unlimited length for the Silesia.

The second problem is in the nature of LZ4 format, which is designed mostly for decompression purposes. The size of the block is always at the beginning of the data block. This limits the minimal latency to be equal to the size of the compressed block because the size needs to be written first. Some of the other hardware implementations[46] of LZ4 changes this format

accordingly to increase their performance. This has a negative effect on the decompression speed.

This LZ4 format modification would lower the minimal latency limitation but it would greatly simplify the Encoder block and it would the lower amount of BRAM resources which are used for the storing the whole block.

Conclusion

An analysis of the LZ4 algorithm and its predecessors LZ77 and LZ78 algorithms have been performed to introduce further improvements to the existing architecture[8]. The improved architecture is capable of compressing data stream with speed up to 8.5 Gbps and can be customized using generic parameters to increase or decrease the compression ratio by increasing or decreasing the size of the internal match searching data structure (hash table). The improved architecture features standardized input and output interface AXI–4 Stream which allows the architecture to be connected into existing infrastructure provided by Xilinx or other IP core manufactures using this interface.

The decompression part of the LZ4 algorithm has not been unfortunately implemented due to the lack of time. This was due to the complexity of the compression which caused the design to be very complex and time-consuming.

During the development, multiple tools for analysis of the LZ4 format has been created to provide additional information about the errors in compressed files, because the official LZ4 compression tools do not provide any option for LZ4 format analysis nor sufficient information about errors in the compressed file.

A software model of the improved architecture has been developed, which was used to select appropriate parameters of the architecture and it has been incorporated into the simulation process of the improved architecture to validate the output of each block in the pipeline.

The improved architecture has been experimentally testing using 1G Ethernet to validate the simulation behaviour in a real environment. There is further testing to be done using 10G Ethernet interface which is by itself a complex problem, which was out of the scope of this thesis.

Based on the analysis of the test results there has been proposed further architecture improvements which could be implemented for increasing the throughput and the compress ratio performance.

Bibliography

[1] *Half of All Internet Traffic Goes to Netflix and YouTube [online]*. [cit. 2019-03-04]. Available from: `https://testinternetspeed.org/blog/half-of-all-internet-traffic-goes-to-netflix-and-youtube/`

[2] Sayood, K. *Introduction to DataCompression*. Morgan Kaufmann, third edition, ISBN 978-0-12-620862-7.

[3] Ziv, J.; Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, volume 23, no. 3, May 1977: pp. 337–343, ISSN 0018-9448, doi:10.1109/TIT.1977.1055714.

[4] Ziv, J.; Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, volume 24, no. 5, Sep. 1978: pp. 530–536, ISSN 0018-9448, doi:10.1109/TIT.1978.1055934.

[5] Storer, J. A.; Szymanski, T. G. Data Compression via Textual Substitution. *J. ACM*, volume 29, no. 4, Oct. 1982: pp. 928–951, ISSN 0004-5411, doi:10.1145/322344.322346. Available from: `http://doi.acm.org/10.1145/322344.322346`

[6] Steven Pigeon. Université de Montréal. *Huffman coding [online]*. [cit. 2019-03-04]. Available from: `http://stevenpigeon.com/Publications/publications/HuffmanChapter.pdf`

[7] Welch, T. A. A Technique for High-Performance Data Compression. *Computer*, volume 17, no. 6, June 1984: pp. 8–19, ISSN 0018-9162, doi:10.1109/MC.1984.1659158. Available from: `https://doi.org/10.1109/MC.1984.1659158`

[8] Bartík, M.; Ubik, S.; et al. LZ4 compression algorithm on FPGA. In *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, Dec 2015, pp. 179–182, doi:10.1109/ICECS.2015.7440278.

[9] *LZ4 Implementation GitHub[online].* [cit. 2019-03-04]. Available from: https://github.com/lz4/lz4

[10] *How LZ4 Works[online].* [cit. 2019-03-04]. Available from: https://ticki.github.io/blog/how-lz4-works/

[11] Doerowicz, S. Canterbury Corpus text compression corpus. Available from: http://www.data-compression.info/Corpora/CanterburyCorpus/index.html

[12] Amold, R.; , T. Canterbury Corpus text compression corpus. Available from: http://www.data-compression.info/Corpora/CanterburyCorpus/index.html

[13] Witten, I.; Bell, T.; et al. Calgary Corpus text compression corpus. Available from: http://www.data-compression.info/Corpora/CalgaryCorpus/

[14] Bell, T.; Kulp, D. Longest-match String Searching for Ziv-Lempel Compression. *Softw. Pract. Exper.*, volume 23, no. 7, July 1993: pp. 757–771, ISSN 0038-0644, doi:10.1002/spe.4380230705. Available from: http://dx.doi.org/10.1002/spe.4380230705

[15] E. Knuth, D.; H. Morris Jr, J.; et al. Fast Pattern Matching in Strings. *SIAM J. Comput.*, volume 6, 06 1977: pp. 323–350, doi:10.1137/0206024.

[16] Boyer, R. S.; Moore, J. S. A Fast String Searching Algorithm. *Commun. ACM*, volume 20, no. 10, Oct. 1977: pp. 762–772, ISSN 0001-0782, doi:10.1145/359842.359859. Available from: http://doi.acm.org/10.1145/359842.359859

[17] Cole, R. Tight Bounds on the Complexity of the Boyer-Moore String Matching Algorithm. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '91, Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1991, ISBN 0-89791-376-0, pp. 224–233. Available from: http://dl.acm.org/citation.cfm?id=127787.127830

[18] Kniesburges, S.; Scheideler, C. Hashed Patricia Trie: Efficient Longest Prefix Matching in Peer-to-Peer Systems. 02 2011, pp. 170–181, doi:10.1007/978-3-642-19094-0_18.

[19] McCreight, E. M. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM*, volume 23, no. 2, Apr. 1976: pp. 262–272, ISSN 0004-5411, doi:10.1145/321941.321946. Available from: http://doi.acm.org/10.1145/321941.321946

[20] Rodeh, M.; Pratt, V. R.; et al. Linear Algorithm for Data Compression via String Matching. *J. ACM*, volume 28, no. 1, Jan. 1981: pp. 16–24, ISSN 0004-5411, doi:10.1145/322234.322237. Available from: http://doi.acm.org/10.1145/322234.322237

[21] Sedgewick, R. *Algorithms*. Addison-Wesley, 1983, ISBN 0-201-06672-6.

[22] Sleator, D. D.; Tarjan, R. E. Self-adjusting Binary Search Trees. *J. ACM*, volume 32, no. 3, July 1985: pp. 652–686, ISSN 0004-5411, doi:10.1145/3828.3835. Available from: http://doi.acm.org/10.1145/3828.3835

[23] Sobti, R.; Ganesan, G. Cryptographic Hash Functions: A Review. *International Journal of Computer Science Issues, ISSN (Online): 1694-0814*, volume Vol 9, 03 2012: pp. 461 – 479.

[24] Bento, L. M. d. S.; SÃ¡, V. A.-c. G. A. P. d.; et al. SOME ILLUSTRATIVE EXAMPLES ON THE USE OF HASH TABLES. *Pesquisa Operacional*, volume 35, 08 2015: pp. 423 – 437, ISSN 0101-7438. Available from: http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0101-74382015000200423&nrm=iso

[25] Knuth, D. E. *The art of computer programming*. Addison-Wesley, second edition, c1998, ISBN 978-0-201-89685-5.

[26] Skarupke, M. Fibonacci Hashing: The Optimization that the World Forgot. 2018. Available from: https://probablydance.com/2018/06/16/fibonacci-hashing-the-optimization-that-the-world-forgot-or-a-better-alternative-to-integer-modulo/

[27] Intel. *Intel® Stratix® 10 Embedded Memory User Guide[online]*. 2019, [cit. 2019-04-18]. Available from: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-memory.pdf

[28] Lattice Semiconductor. *Memory Usage Guide for iCE40 Devices[online]*. 2019, [cit. 2019-04-18]. Available from: https://www.latticesemi.com/-/media/LatticeSemi/Documents/ApplicationNotes/MO/MemoryUsageGuideforiCE40Devices.ashx?document_id=47775

[29] Laforest, C. E.; Li, Z.; et al. Composing Multi-Ported Memories on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, volume 7, no. 3, Sept. 2014: pp. 16:1–16:23, ISSN 1936-7406, doi:10.1145/2629629. Available from: http://doi.acm.org/10.1145/2629629

[30] LaForest, C. E.; Steffan, J. G. Efficient Multi-ported Memories for FPGAs. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10,

New York, NY, USA: ACM, 2010, ISBN 978-1-60558-911-4, pp. 41–50, doi:10.1145/1723112.1723122. Available from: http://doi.acm.org/10.1145/1723112.1723122

[31] Xilinx, Inc. *Kintex-7 FPGAs Data Sheet: DC and AC Swirching Characteristics [online]*. 2017, [cit. 2019-02-18]. Available from: https://www.xilinx.com/support/documentation/data_sheets/ds182_Kintex_7_Data_Sheet.pdf

[32] Eric LaForest, C.; G. Liu, M.; et al. Multi-ported memories for FPGAs via XOR. 05 2012, pp. 209–218, doi:10.1145/2145694.2145730.

[33] ARM. *AMBA AXI and ACE Protocol Specification [online]*. 2011, [cit. 2019-02-18]. Available from: https://silver.arm.com/download/download.tm?pv=1198016

[34] Xilinx, Inc. *Vivado Design Suite - Vivado AXI Reference [online]*. 2017, [cit. 2019-02-18]. Available from: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

[35] ARM. *AMBA 4 AXI4-Stream Protocol [online]*. 2010, [cit. 2019-02-18]. Available from: https://silver.arm.com/download/download.tm?pv=1074010

[36] Jarvis, S. A.; He, L.; et al. The impact of predictive inaccuracies on execution scheduling. *Performance Evaluation*, volume 60, no. 1, 2005: pp. 127 – 139, ISSN 0166-5316, doi:https://doi.org/10.1016/j.peva.2004.10.015, performance Modeling and Evaluation of High-Performance Parallel and Distributed Systems. Available from: http://www.sciencedirect.com/science/article/pii/S0166531604001373

[37] Bartik, M.; Benes, T.; et al. Design of a High-Throughput Match Search Unit for Lossless Compression Algorithms. 01 2019, pp. 0732–0738, doi:10.1109/CCWC.2019.8666521.

[38] Xilinx, Inc. *UltraScale Architecture DSP Slice [online]*. 2017, [cit. 2019-02-18]. Available from: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf

[39] Xilinx, Inc. *ISIM User guide [online]*. 2012, [cit. 2019-02-18]. Available from: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/plugin_ism.pdf

[40] *OpenCores[online]*. [cit. 2019-03-04]. Available from: https://opencores.org

[41] Xilinx, Inc. *JTAG to AXI Master v1.2[online]*. 2016, [cit. 2019-02-18]. Available from: https://www.xilinx.com/support/documentation/ip_documentation/jtag_axi/v1_2/pg174-jtag-axi.pdf

[42] Xilinx, Inc. *Xilinx LZ4 [online]*. [cit. 2019-04-30]. Available from: https://github.com/Xilinx/Applications/tree/master/data_compression/xil_lz4

[43] Xilinx, Inc. *Vivado Design Suite User Guide [online]*. 2017, [cit. 2019-02-18]. Available from: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug910-vivado-getting-started.pdf

[44] Xilinx, Inc. *UltraScale Architecture and Product Data Sheet: Overview [online]*. 2019, [cit. 2019-02-18]. Available from: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[45] *Kintex UltraScale KU040 Development Board[online]*. [cit. 2019-03-04]. Available from: https://www.avnet.com/opasdata/d120001/medias/docus/13/aes-AES-KU040-DB-G-User-Guide.pdf

[46] Liu, W.; Mei, F.; et al. Data Compression Device Based on Modified LZ4 Algorithm. *IEEE Transactions on Consumer Electronics*, volume 64, no. 1, Feb 2018: pp. 110–117, ISSN 0098-3063, doi:10.1109/TCE.2018.2810480.

Appendix A

Compress ratio visualization

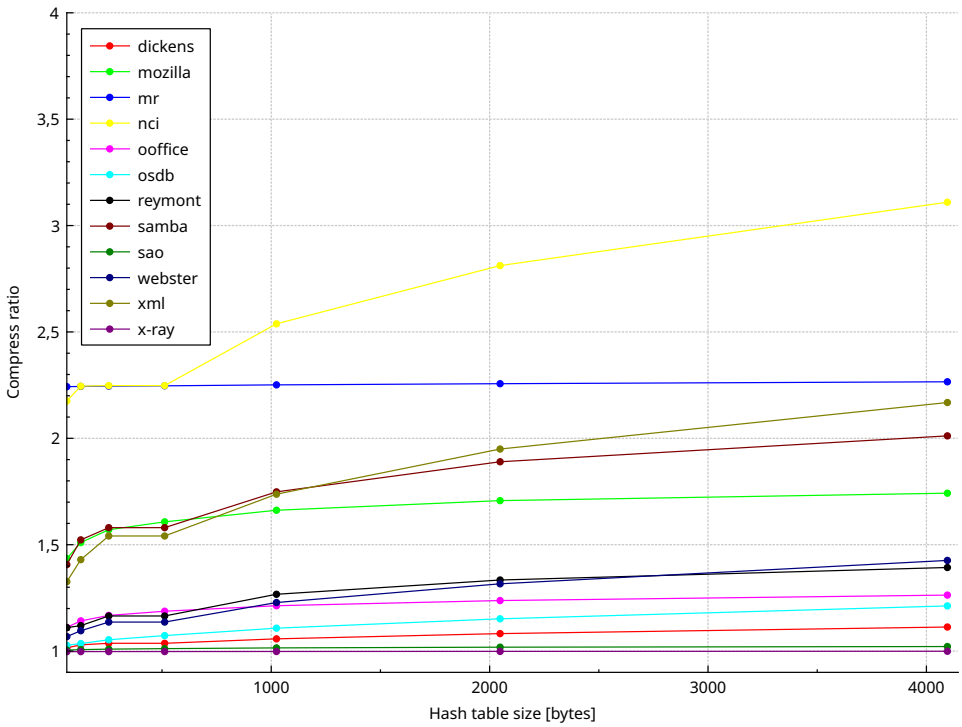

Figure A.1: The average compress ratio with the increasing hash table size for Silesia corpus

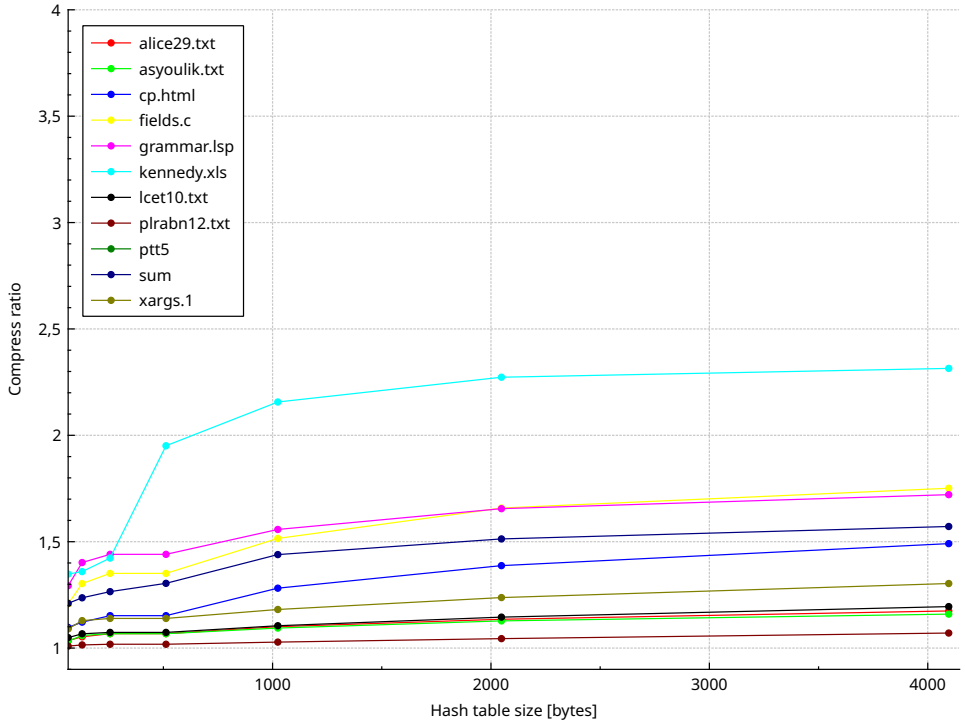

Figure A.2: The average compress ratio with the increasing hash table size for Cantebury corpus

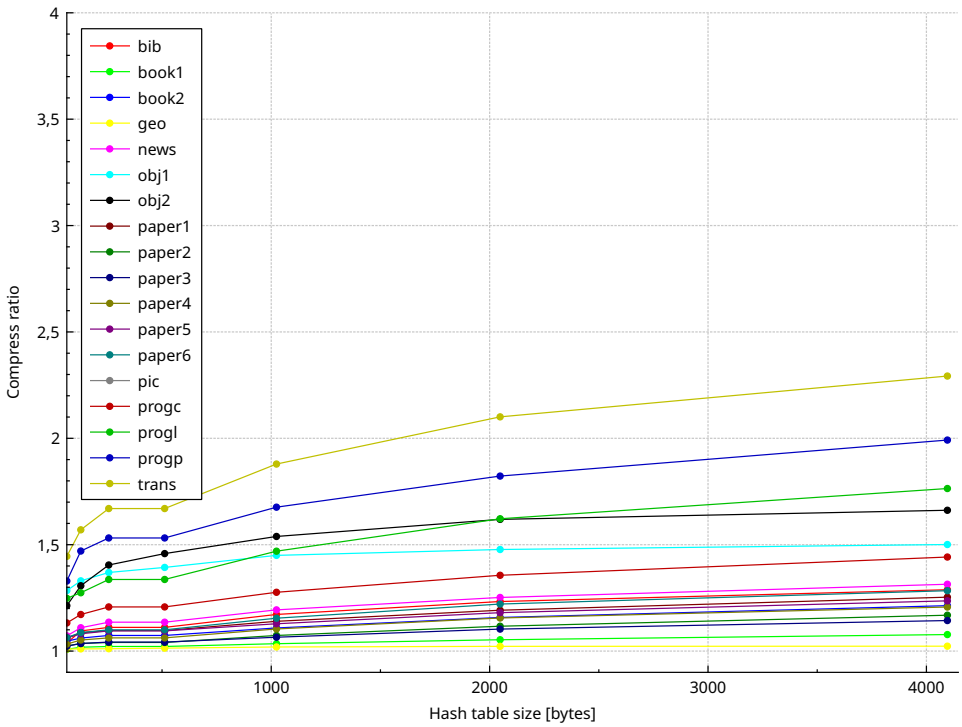

Figure A.3: The average compress ratio with the increasing hash table size for Calgery corpus

Appendix B

Acronyms

AMBA Advance microcontroller bus architecture

AXI Advanced eXtensible Interface

BRAM Block random access memory

BSD Berkeley Software Distribution

CAM Content addressable memory

CPU Central processing unit

DSP Digital signal processing

FIFO First in first out

FPGA Field programmable gate array

HLS High-Level Synthesis

HTML Hypertext markup language

IP Intellectual property

JTAG Joint Test Action Group

LIFO Last in first out

LSIC Linear small-integer code

LUT Look-up table

LVT Live value table

MSU Match search unit

MLF Match length finder

PHY Physical layer

RAM Random access memory

RX Receive

SOC System on chip

TEMAC Tri-Mode Ethernet MAC

TX Transmit

UDP User datagram protocol

URAM Ultra random access memory

XML Extensible markup language

Appendix C

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `LZ4Tool` ..... the modified official LZ4 tool
  - `LZ4PerformanceExp` ..... the software model of the LZ4 ip core
  - `LZ4LoopBack` ..... the testing architecture
  - `LZ4Compression` ..... compression ip core
- `corpora` ..... the directory of corpora
  - `calgary` ..... the Calgary corpus
  - `canterbury` ..... the Canterbury corpus
  - `silesia` ..... the Silesia corpus
- `text` ..... the thesis text directory