A Parallel Architecture for Giga Bit Lossless Data Compression

By
Muhammed Salman Rashed
[2010-NUST-MS PhD-ComE-30]

Submitted to the Department of Computer Engineering
In partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Engineering

Advisor
Dr Shoab Ahmed Khan

College of Electrical & Mechanical Engineering
National University of Sciences and Technology
2013

DECLARATION

We hereby declare that no portion of the work referred to in this Project Thesis has been submitted in support of an application for another degree or qualification of this of any other university or other institute of learning. If any act of plagiarism found, we are fully responsible for every disciplinary action taken against us depending upon the seriousness of the proven offence, even the cancellation of our degree.

COPYRIGHT STATEMENT

- Copyright in text of this thesis rests with the student author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the author and lodged in the Library of NUST College of E&ME. Details may be obtained by the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the author.
- The ownership of any intellectual property rights which may be described in this thesis is vested in NUST College of E&ME, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the College of E&ME, which will prescribe the terms and conditions of any such agreement.
- Further information on the conditions under which disclosures and exploitation may take place is available from the Library of NUST College of E&ME, Rawalpindi.

This thesis is dedicated to my daughter
Mishaal

Abstract

The thesis proposes a high speed data compression architecture for multi-purpose communication applications. The architecture is a systolic array design for hardware implementation, and LZ 77 has been implemented as compression algorithm. Compression is achieved by reducing redundancy inherent in data, by comparing input data stream with previously encoded and transmitted bits. Design has been further optimized by carrying out searches in parallel, and future searches of data stream are carried out in same clock cycle increasing throughput to manifolds. The architecture has been further pipelined to reduce the critical time and further enhance throughputs exceeding 10 Gbits / sec.

The motivation for research is the ever-increasing development in the field of digital communication. Requirements of information transfer as well as storage transferred have increased many folds in last many years. The communication bandwidth, although trying to match the pace of information increase has always been lagging behind, and there has always been a mis-match between the two. And both communication bandwidth and memories are quite expensive and cannot be increased just to match to the ever-growing information across networks. This necessitates measures to reduce the amount of information that needs to be transferred or stored, and hence, the basis of Data Compression.

Various techniques varying from statistical to dictionary based methods exist in literature with a view to remove redundancies in data and achieve data compression. The techniques have been successfully implemented both across software and hardware platforms with far-reaching results. The choice of LZ 77 Algorithm, the simplest of dictionary based schemes, with a systolic architecture, owing to its inherent promising advantages of speed, error-resistant and simplicity forms the basis of this research.

Acknowledgements

First and foremost, I admit that without the blessings of **Almighty Allah**, the most Merciful and the Most Gracious, I would not have been able to complete this research.

Secondly, I am in great debt to my thesis advisor, **Dr. Shoab Ahmed Khan**, whose constant motivation, clear guideline, persistent efforts and words of wisdom were always a beacon of guidance for me. He was there to help and guide me whenever and wherever I looked for him. Despite his never ending assignments of University and Department management, student counseling, project supervision and teaching, his commitments were never been a reason for his non availability whenever I requested for his time and support and he was always there to help me with his vast experience and words of wisdom.

I would also like to highlight acknowledgements to all of my teachers and colleagues, who were source of inspiration for me throughout the course work, especially **Mr. Sajid Gul, Mr Waqas Mazhar and Mr Bilal Tahir** who helped me out to bridge all the gaps and bottlenecks in concluding the research work.

Table of Contents

DECLARATION …………………………………………………………………... 2

COPY RIGHT STATEMENT 3

ABSTRACT…………………………………………………………………..... 5

ACKNOWLEDGEMENTS…………………………………………….......... 6

TABLE OF CONTENTS………………………………………………………. 7

LIST OF FIGURES……………………………………………………………... 11

LIST OF TABLES ……………………………………………………………… 12

CHAPTER 1: INTRODUCTION………………………………………………. 13

1.1 Motivation ………………………………………………………………... 13

1.2 Introduction to Data Compression ……………………………………..... 14

1.3 Traditional Applications of Data Compression ……………………… 14

1.4 Data Compression Unfolded ……………………………………………... 15

1.5 Systolic Arrays …………………………………………………………. 17

1.6 Organization of Thesis…...……………………………………………... 18

CHAAPTER 2 : RELATED WORK ……………………………………... 20

2.1 Data Compression Requirements ……………………………………... 20

2.2 Data Compression Terminologies …………………………………….. 20

2.2.1 Compression Ratio ………………………………………………... 21

2.2.2 Compression Factor ……………………………………………….. 21

2.2.3 Saving Percentage …………………………………………………. 21

2.2.4 Compression Time ………………………………………………… 21

2.2.5 Entropy ………………………………………………………….. 22

2.2.6 Code Efficiency ………………………………………………. 22

2.2.7 Bandwidth ……………………………………………………. 23

2.2.8 Throughput ……………………………………………………... 23

2.3 Lossless Data Compression ………………………………………. 23

2.3.1 Statistical Data Compression Scheme ……………………………. 23

2.3.2 Dictionary Based Schemes …………………………………... 24

2.3.3 Static vs Adaptive Dictionaries …………………………………… 25

2.3.4 Lempel Ziv Family of Algorithms ………………………………... 26

2.4 Implementation Methodologies ………………………………………. 28

2.4.1 Software Solutions …………………………………………….... 28

2.4.2 Hardware Solutions ……………………………………………. 28

2.4.3 Microprocessor Based Implementations ……………………….... 29

2.4.4 CAM Based implementations…………………………………….. 29

2.4.5 Systolic Array Architectures …………………………………….. 30

2.5 Summary ……………………………………………………………. 33

CHAPTER 3: LZ ALGORITHMS AND IMPLEMENTATION ………. 34

3.1 Why Dictionary Based Schemes ……………………………………… 34

3.2 Introduction to LZ 77 Family ………………………………………… 35

3.3 LZ 77 Data Compression Algorithm ………………………………… 36

3.3.1 The Algorithm ………………………………………………….. 37

3.3.2 Encoding Process ………………………………………………. 39

3.3.3 Decoding Process ………………………………………………. 41

3.4 Performance and Limitations of LZ 77 Algorithm …………………. 42

3.5 Summary …………………………………………………………………….. 43

CHAPTER 4: OWN DESIGN AND METHODOLOGY ……………….. 44

4.1 Introduction ……………………………………………………………… 44

4.2 Why Systolic Architectures …………………………………………… 45

4.2.1 Systolic Architectures Elaborated ……………………………….. 45

4.2.2 Issues in Special Purpose Chip Design ………………………….. 47

4.3 Proposed Compression Architecture ………………………………… 47

4.3.1 The Compression Cell ……………………………………………. 47

4.3.2 Column Logic ……………………………………………………..... 50

4.3.3 Pointer and Length Calculation …………………………………. 52

4.3.4 Compression Block ………………………………………………... 52

4.4 Summary ……………………………………………………………………. 54

CHAPTER 5: SUPER UNFOLDED AND PIPELINED ARCHITECTURE ……………………………………………………………. 55

5.1 Unfolding of Architecture ……………………………………………... 55

5.2 Super Unfolded Architecture ………………………………………… 55

5.3 Optimized Super Unfolded Architecture ……………………………. 57

5.4 Summary ……………………………………………………………………. 60

CHAPTER 6: RESULTS, COMPARISON AND FUTURE RECOMMENDATIONS ………………………………………………….. 61

6.1 Results ……………………………………………………………………... 61

6.2 Comparisons ……………………………………………………………… 64

6.3 Summary ………………………………………………………………… **64**

CHAPTER 7: CONCLUSION AND FUTURE RECOMMENDATIONS **65**

7.1 Conclusion ……………………………………………………………….. **65**

7.2 Future Recommendations ……………………………………………. **65**

7.3 Summary ………………………………………………………………… **66**

REFERENCES ……………………………………………………………….. **67**

List of Figures

Figure 2.1: LZ Family of Algorithms …………………………………….. 26

Figure 2.2: Sliding Window of LZ 77 ……………………………………. 27

Figure 2.3: Systolic Array ………………………………………………… 31

Figure 3.1: Sliding Buffers of LZ 77 ……………………………………… 36

Figure 4.1: Simplified Systolic Array …………………………………….. 46

Figure 4.2: Compression Cell ……………………………………………... 48

Figure 4.3: Column Logic with Logarithmic Complexity ……………….. 51

Figure 4.4: Pointer and Length Calculation Block ………………………. 52

Figure 4.5: Compression Block for One Iteration ……………………….. 53

Figure 5.1: High Speed Super Unfolded Architecture …………………... 59

Figure 6.1 Simulated Design with Random Input Data ………………… 62

Figure 6.2 Simulated Design Outputs with Random Input Data ……… 63

List of Tables

Table 3.1: Encoding Process of LZ77 Algorithm ……………………………... 40

Table 3.2: Decoding Process of LZ 77 Algorithm …………………………….. 42

Table 4.1: Parallel Comparisons of LZ 77 Algorithm Implementation ……... 49

Table 5.1: Multiple Iterations of LZ 77 Algorithm with Parallel Searches ….. 56

Table 5.2: Identification of Redundant Comparisons ………………………… 58

Table 6.1: Experimental Results of Proposed Design………………………….. 64

CHAPTER 1
INTRODUCTION

1.1 Motivation

Data compression is fast becoming a necessity to sustain the development and even normal functioning of high data-rate data communication systems and data storages. With the ever increasing awareness and requirements of users, the endless series of questions warranting WHAT and WHY need to be answered and user satisfaction guaranteed for progress of technology. With limited bandwidth of communication channels and limited space for storages, coupled with high costs associated for increasing their capacities, the only viable solution left is to economize the existing capacities available and develop frameworks that can be incorporated in these systems and achieve higher desired specifications.

Considerable work has been done in this field, with evolving efficient designs competing for implementation and promising higher data rates and storages. Various algorithms have been developed, modified and implemented, both in software and hardware platforms, with their pros and cons. However, hardware based systems with only drawback of system complexity vis-à-vis their higher processing rates and reduced costs still promise to be a competitive solution to existing bottle-necks.

This forms the basis of our proposed design which explores a data compression algorithm implementation on an efficient, optimized hardware design, promising exceptionally high throughputs.

1.2 Introduction to Data Compression

In 1940, when Claude Shannon first conceived a communication system that could efficiently transmit the information produced from a source to a destination, he laid down the foundations of an entirely new chapter in field of communication systems, that we now call Data Compression. The field that was limited to a very small number of engineers and scientists is now a leading and enabling technology for multimedia revolution.

Data compression is the science of representing information in a compact form, by removing all possible redundancies inherent in the data and encoding repeated longer strings into shorter codes [1]. In essence, data compression research aims to evolve compression algorithms and their implementations, which are actually ways to reduce the number of symbols required to represent source information. This reduces the amount of space needed to store the information as well as amount of time necessary to transmit that information over available media.

The advancement of technology world over has seen a vast increase in the amount information transferred across media, in terms of communication. The increase in amount of information has also forced media to evolve and develop, in order to meet the challenging requirements of ever increasing data, with advancement in fields of both hardware and software to support communication requirements.

Over a period of time, data compression has become a very common requirement for most software applications as well as an active research area of computer science. Personal communication a decade back involved voice only and texts were considered quite infrequent and uncommon. Today, it encompasses texts coupled with multimedia as well as frequent data exchanges, taking personal communication to a whole new world.

Take another example of High Definition Television (HDTV). Without compression, it requires data transmission rate of 885 MBits per sec, requiring approximately 220 MHz of channel bandwidth. With the help of data compression, the effort is reduced to 20 MBits per sec and a channel bandwidth of 6 MHz. Similarly, without data compression, the speedy transmission over facsimile would also not have been possible. It would take a whole day just to transmit a single page document.

Compressed data thus offloads the choked bandwidth providing effective and efficient communication [2-7]. Similarly, the physical space of storage devices, whether it is static type hard disks or dynamic memories, is also relieved by use of compressed data [8-9].

1.3 Traditional Applications of Data Compression

Analyzing the field of data communications, there are mainly two main applications of data compression. The first is to save storage space; by saving files in such a manner that they

occupy less space, hence, compressed. Within the aspect of saving storage space, the characteristics of the compressors and the decompressor are determined by the accessibility patterns of these files once used. Backup files, which are rarely used, such as in case of emergency recovery, necessitate the compressors and decompressor to have slow speeds. The thing that matters here is compression and not quickness of response.

Similarly, if files are read frequently but updated occasionally, then compressor may be slow, but decompressor has to be comparatively faster. However, if both the reading and updating of files are quite frequent, then both the compressors and decompressor have to be fast.

Another issue is that of robustness; backups stored in a medium prone to introduce small errors indicate use of a compression method that may allow for recovery of some part of the file or files. However, this means introducing redundancy in the compressed files, which goes against data compression that aims to reduce redundancy. For this, error correcting codes are used, which is another very vast and complex aspect, and does not form part of the research we have carried out.

The second application of data compression is where data before transmission is compressed at the source end and on receiving, is decompressed at the destination end. With the explosive growth of digital information transmission, for example, over the internet, there is always problem of matching available bandwidth to required data transfer rates. And in today's fast world of development, getting delayed sometimes results in losing the entire struggle. The aim is to effectively utilize the available bandwidth of the communications channel to one's benefit.

1.4 Data Compression Unfolded

Depending on the application requirement, there are basically two types of data compression; the lossy and the lossless. As the name implies, the lossy data compression attempts to reduce the actual number of bits carrying information, while at the same time compromising on the integrity of the actual information, to an extent that is either not noticeable or can be done away with. A simple example of lossy compression is the gif file system, which tends to ignore the high frequency components not visible to a naked eye, and yet displays a considerably satisfying picture to a viewer. Such like approach is used where there is no harm in compromising on the quality attributes of the information.

Lossless data compression implies achieving compression under strict constraints of retaining actual information, not compromising on the quality attributes. This kind of technique may also be termed as a reversible process, where the data before compression and after decompression is exactly identical. Medical imaging applications require enhancing pictures for the purpose of detection of minor defects/changes and such like images, if compressed under lossy technique, are bound to suffer and sometimes, result in fatal consequences with inability to detect the presence of a flaw.

There are various algorithms that have been developed to accomplish both the lossy as well as lossless data compression. The algorithms take into account the two aspects of data compression, i.e. modeling and coding. The model component analyzes the input data stream and identifies characteristics in the incoming data, such as probabilities, repetitions and redundancies for exploitation. The coder part then takes into account the modeling outcome, such as the probability biases, and uses this information to generate codes for symbols. Huffman Coding is a perfect example of this, where modeling component analyzes the probabilities of symbols, and the coder component then generates codes. Shorter codes are used for frequently appearing symbols and relatively larger codes are used for rarely appearing symbols.

Here, we must understand that a typical model developed for one system may not be practical for implementation on another system. However, having one model, there may be different ways to code the symbols. This lays down the basis for the two basic schemes of modeling, namely the statistical modeling and the dictionary based modeling approaches. The modeling schemes are completely independent of the coding schemes, which are basically means to implement theoretical algorithms on a specific type of model.

Moreover, the implementation of data compression may be done over software platform or hardware architecture. The two types vary in speed of implementation as well as costs and resources involved. Software based platforms are relatively slow, since they require frequent memory addressing, hence introducing latency in the system, which in turn implies reduced throughput. Increase in throughputs through software based solutions means increased memories for larger searches, and increased memories implies increased costs. Hardware

based platforms are difficult to implement, involving complex architectures, yet they guarantee high throughputs with comparatively reduced overheads.

Within the hardware platforms, there are a number of techniques to implement data compression models, the two most distinct approaches being the Content Addressable Memories (CAMs) based architectures and the Systolic Array architectures. Details on these aspects shall be covered in subsequent chapters, however, we shall dwell more on the Systolic Array based architectures as it forms the basis of our research.

1.5 Systolic Arrays

There is continuing drive to create faster and faster computers to address the ever increasing signal processing and communications applications. Faster computers mean faster clock rates, and there is a limit to gaining computational advantage by increasing speed alone. The computers need increased memory to support their input / output requirements, when processing at such high speeds; and memories are quite expensive. The solution to this problem has been found by extracting parallelism, which enables more work to be done in less time.

This concept has introduced computer architectures that have made possible the development and implementation of multi-processor platforms to execute multiple instructions/processes. Flynn's taxonomy gives a detailed account of various types of parallel computing architectures. From parallel computing evolves the concept of systolic array architectures.

Systolic arrays are special purpose processors which are very fine-grained and hence suited for typical applications that are computationally intensive. They a have a very simple structure that enables them to execute limited and very simple computations on very high clock rates. Data between these processors is moved in a synchronous and rhythmic manner, mimicking the systole of a human heart that regularly and repetitively supplies blood to complete body through rhythmic pumping [10].

Unlike typical parallel computing architectures where processors loose speed due to interconnects, systolic arrays are special-purpose parallel architectures that have processors or processing elements connected by very short wires. Moreover, another advantage gained by use of systolic architectures is the reduction in computing complexity. For example an n x n

matrix multiplication would normally require n^n computations. However, systolic architectures reduce this computation to n^2 computations.

Amongst various factors that have contributed to systolic array's choice as a leading choice for handling computationally intensive applications are technology advances, concurrent processing and demanding scientific applications [11].

In succeeding chapters, we will elaborate our research which is proposes a novel architecture for implementing a high speed lossless data compression algorithm on a simple systolic array based design that is capable of handling very computationally intensive data, yielding throughputs exceeding 10 Gbits/sec.

1.6 Organization of Thesis Report

In this chapter, a brief introduction to various aspects of data compression has been covered. Typical applications of data compression in communication systems have been covered followed by a brief account of lossy and lossless data compression has been discussed. Lastly, we have introduced systolic array architectures, an evolving solution to address the modern day to day computationally intensive applications that are performed without compromise on speed of processing. The rest of the thesis is organized as follows.

Chapter 2 gives detailed account of literature review, with focus on related work on the subject, up to and including most recent advances in the pertinent field of LZ 77 hardware based lossless data compression.

In Chapter 3, we have dilated upon LZ 77 Data Compression Algorithm and its working methodology in detail. The concept of Algorithm and its working is discussed in detail, with an example as relevant to our work. Chapter 4 elaborates architectural details of proposed design. The conception of design with different modules in hierarchical order is discussed at length concluding with a block required for single iteration. Chapter 5 takes on the basic design to a higher level, with its unfolded and pipelined versions, which greatly enhance the perceived throughputs.

Chapter 6 comprises results obtained when the design is simulated and synthesized on FPGA families. The results are discussed in detail and compared with some of the recent designs

proposed in literature. Chapter 7 concludes the report with insight into Future Recommendations.

CHAPTER 2

LITERATURE REVIEW

2.1 Data Compression Requirements

Data required to be stored on a storage media or transferred over a communication link contains significant redundancy. This inherent redundancy may be modeled and then exploited by means of a coding strategy that aims to reduce the redundancy, and hence offload thick data communication channels and data storage devices. Same information now requires fewer overheads, either for storage or transmission. And in today's modern age of digital data where information is represented by bits of value 0 and 1, this inherent redundancy is always there. The only thing left is to devise strategies for modeling and coding of data and hence reducing redundancies.

In the scenarios of satellite communications, where an enormous amount of information is required to be transferred between various communication entities, the data rates rises to the orders of Giga Bits/sec. This communication, if not expertly handled by use of optimized systems, may easily breakdown the entire communication channel, resulting in loss of critical information.

Problems associated once common compression methodologies are integrated into computer systems have encouraged increased tendency to use automated data compression. This includes mis-match between execution times of the techniques and data transfer rates, lack of flexibility for implementation on multiple applications once developed and at times, non-optimized compression techniques, resulting in opposite of what was intended to be achieved. Techniques developed for compressing one specific application, when used for another application, sometimes result in expansion of the data rather than opposite.

2.2 Data Compression Terminologies

In order to gauge the performance metrics of any technique for the purpose of implementing data compression, it is necessary to understand various terminologies associated with this field.

Space and time efficiency could be one of the factors that must be analyzed. Speed of compression and ratio of compressed data to uncompressed data could be another. All this in turn depends on the input source of data and the model being used, whether lossy or lossless. Thus, there are various criteria to evaluate any compression algorithm and those associated with lossless families will be briefly dealt with in subsequent paragraphs.

2.2.1 Compression Ratio

Compression ratio is ratio between the size of the compressed file and the size of the uncompressed source file. Mathematically, it is defined as:-

$$\mathbf{Cr} = \frac{\textbf{size of compressed file}}{\textbf{size of uncompressed file}} \tag{2.1}$$

2.2.2 Compression Factor

Compression factor is the inverse of compression ratio, and is ratio of uncompressed source file to compressed file. Mathematically, it is defined as:

$$\mathbf{Cf} = \frac{\textbf{size of uncompressed file}}{\textbf{size of compressed file}} \tag{2.2}$$

2.2.3 Saving Percentage

Saving percentage refers to the percentage reduction in size of the compressed file achieved as a result of compression. Mathematically, it is defined as:

$$\mathbf{S}\,\% = \frac{(\textbf{size before compression} - \textbf{size after compression})}{\textbf{size before compression}} \times \mathbf{100}\,\% \tag{2.3}$$

2.2.4 Compression Time

Compression time refers to time needed for both compression and decompression and requires that both be accounted for, separately. Compression algorithms when implemented may have same time required for compression as well as decompression or even different times. Therefore, it is necessary to take into account both the factors. In storage of video, we may

afford to have slower compression techniques; however, once reading the same video under the same system, decompression has to be faster to accommodate uninterrupted viewing.

2.2.5 Entropy

A key measure of information is known as entropy, which is usually expressed by the average number of bits needed to store or communicate one symbol in a message [12]. The information content of a group of bytes (a message) is termed as *entropy H*, which is a function of symbol frequency. It is the weighted average of the number of bits required to encode the symbols of a message and mathematically defined as:

$$I(s_i) = \log_2 \frac{1}{p_i} = \log_2 p_i \qquad (2.4)$$

where p_i is the probability of occurrence of the symbol s_i

This mathematical notation establishes the fact that higher the probability of a symbol, lower will be the information carried by it, to the extent that if a symbol has pi =1, then information in the communication I(s) = 0 !

The entropy of the entire message is the sum of the individual symbol entropies, given by mathematical relationship as:

$$\mathbf{I} = \sum \boldsymbol{P(x_i)} \times \mathbf{log_2}\, \boldsymbol{P(x_i)} \qquad (2.5)$$

2.2.6 Code Efficiency

Average code length is the average number of bits required to represent a single code word. If the source and the lengths of the code words are known, the average code length can be calculated using the following equation:-

$$\acute{\mathrm{I}} = \sum_{\mathbf{j=1}}^{\mathbf{n}} \mathbf{p_j} \cdot \mathbf{I_j} \qquad (2.6)$$

where p_i is the occurrence probability of jth symbol of the source message,

l_j is the length of the particular code word for that symbol and L = {l1, l2, …. Ln}.

2.2.7 Bandwidth

The maximum data transfer rate that can be achieved or allowed over a medium is referred to as bandwidth of the media. It is a measure of how much data can be transferred in a given amount of time. It does not measure the speed of travel from one point to another, rather amount of data flowing through a specific point at one time.

2.2.8 Throughput

Throughput is analogous to bandwidth. It is essentially the data transfer rate that is practically achieved. Maximum throughput achieved is equal to digital bandwidth of the media. It may also be termed as amount of data processed in one clock cycle of any communication system.

2.3 Lossless Data Compression

Lossless data compression, as we have dwelled upon earlier, implies a reversible process, where the data before and after compression is exactly identical; there is absolutely no room for loss of any information in the data. As compared to lossy techniques, wherein, high frequency information, that is either not noticeable or may be compromised, is done away with, lossless techniques are applied mostly to texts based applications, where integrity of the data is of prime importance. Important databases, executable codes and sensitive biomedical images all are compressed using lossless technique.

A lossless compressor maps all files to different encodings; if it shortens some files, it necessarily makes others longer. We try to design the compressor so that the probability that a file is lengthened is very small, and the probability that it is shortened is large.

There are two main schemes of lossless data compression, as regards to modeling of the systems, namely the statistical scheme and the dictionary based scheme. The two schemes alongwith related work on the relevant method shall be dealt with in subsequent sections.

2.3.1 Statistical Data Compression Scheme

In statistical modeling, data is analyzed for entropy, characters and word frequencies and probabilities. These characteristics of data are tabulated and codes generated for the

characters based on their frequencies. The statistics of input data reveal the inherent redundancy in the input data which is coded using an appropriate algorithm.

Shannon-Fano Codes and Huffman trees are common examples of this model, which are also known as fixed to variable length coding [13]. But such like statistic modeling is particular to the data stream which has been previously analyzed. If the data stream changes, the entire model of compression degrades to unacceptable levels.

Building different models for different streams implies overheads, since the model has to be transmitted to the decoder ahead of the compressed data stream. Moreover, compression performance of these codes may be high, but they are very slow to implement and hence not suited for real-time use.

For most data applications, statistical methods provide excellent compression ratios. However, there are two main disadvantages associated with them. The first disadvantage is that it requires two passes over the data before encoding. Firstly, for calculating the probabilities, and secondly, for encoding the data based on the calculated probabilities. The second disadvantage is non-optimal encoding.

According to Shannon theory [14], the optimal number of bits to be used for each symbol is calculated by log base 2 of (1/p) where p is probability of the given symbol. The coding ceases to be optimal as the number of bits used to encode are integral and cannot be fractional as necessitated by probabilities at times. For example, take a case of a probability of a symbol being 9/10, which would assign 0.15 bits for encoding; however, due to limitation, Huffman coding would assign 1 bit, which would be almost 6 times more than required.

However, despite all these facts, statistical modeling remained the undisposed king of compression techniques and considerable work was done in the field of communications with far reaching results. In fact, some of the schemes are still very effective for some specific applications, either when used alone or in combination with other techniques.

2.3.2 Dictionary Based Schemes

In certain cases of communication systems, the embedded structural regularities in strings of incoming data may be taken advantage of, hence saving time for modeling and achieving better

performance than statistical modeling. Cases where *a priori* knowledge of incoming data source renders statistical modeling to be impossible or unreliable, there is a need to develop and adopt a universal coding scheme that has the learning aspect interlaced within the coding process. This forms the basis of dictionary based modeling for reducing redundancy.

Prior to 1977, research in field of data compression was confined to statistical modeling and coding, and improvements of these approaches. Jacob Ziv and Abraham Lempel presented a novel modeling technique that gave birth to a new concept, i.e. Dictionary based data compression scheme. The scheme was presented in their two successive publications in 1977 and 1978 [15-16].

Dictionary based scheme uses a completely different approach to achieve compression; in fact, the model is opposite to statistical modeling in that, compression is achieved by using variable to fixed length coding. A dictionary is built by analyzing data and matches of incoming stream of data is matched with previously transmitted data; if a match is found, the pointer indicating the location and length of previous data is transmitted as a token. If the match is not found, the data is coded in with tokens, but in a less efficient manner. If the number of bits of tokens is smaller than actual data stream bits, compression is achieved.

All dictionary schemes have almost same statistical scheme and thus achieve the same compression ratios. Whereas higher order context models may achieve better compression ratios in statistical modeling, dictionary schemes are getting popular because of their speed and economy of memory. Moreover, such methods address the limitation of statistical methods in that there is just one pass over the data required for encoding. Data is coded and compressed in real-time, on the fly.

2.3.3 Static vs. Adaptive Dictionaries

A static dictionary is one that is built and synchronized before communication takes place. Such a dictionary is not generalized, but specific to an application. The biggest advantage is its simplicity, that is, it can be tuned up by making small changes to fit the data it is intended for. Thereafter, the compression and decompression is automatic. The disadvantages of such like dictionary are its inflexibility to adapt to changing models as well as how to pass it to the decoder for synchronizing.

Adaptive dictionaries on the other hand are generalized dictionaries that may be conveniently used with lot of flexibility. These dictionaries are empty or contain null characters at the start of the process, and as compression process continues, they are built up sequentially, retaining recent and frequent characters and discarding old and rarely occurring characters. The dictionary adapts itself to the input data as it changes, automatically. At the decoder end, the data is decoded on the basis of recently received data and hence synchronization and updating of dictionary is automatic.

With the advent of this technique, technological advancement has been quite considerable in the field of data compression. Both software and hardware solutions make it practical for implementation.

2.3.4 Lempel Ziv Family of Algorithms

The Lempel Ziv Algorithms, also common known as LZ Algorithms is a whole family of adaptive dictionary based algorithms, branching from two main algorithms proposed by the authors Jacob Ziv and Abraham Lempel in 1977 and 1978 [Christina Seminars], as depicted in Figure 2.1.

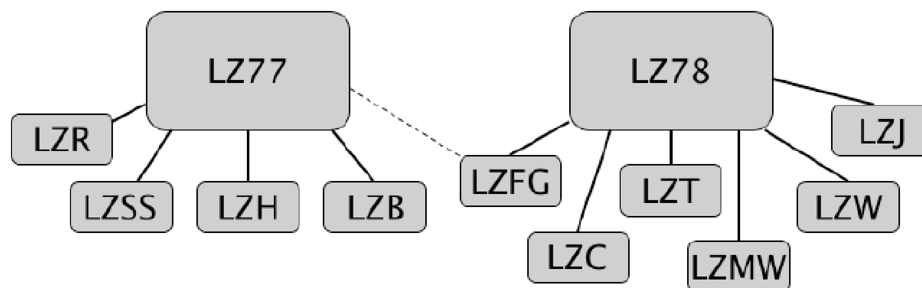

Figure 2.1 – LZ Family of Algorithms [17]

The two main algorithms are in fact based on the same concept, with the difference of dictionary updating technique. Other branches from the two main algorithms are optimized improvements to the main algorithms, as developed from time to time.

In LZ77 Algorithm [15], the dictionary is a sliding window, consisting of a search and history buffer, as shown in Figure 2.2. At the encoder end, the incoming data slides into the search buffer for encoding and transmission, shifting into history buffer after encoding for future comparison and out of it. The symbols in search buffer are compared to the already encoded symbols in the history buffer, and if a match is found, a code or a token, consisting of a triplet

(p,l,C) is output as a code to the decoder. In the token triplet, p refers to the pointer or location of start of matched symbol or string, l denotes the length of the matched string, and C denotes the first unmatched character after the matched string.

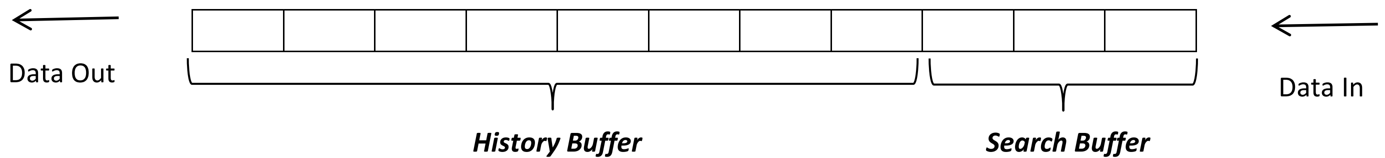

Figure 2.2 - Sliding Window of LZ 77

LZ 77 is also termed as a greedy algorithm, as it attempts to find longest match of strings for encoding. The size of the window, i.e. history and search buffers are the main driving factors affecting the compression ratio achieved. However, the limitation of LZ 77 Algorithm is that if the distance or number of characters between two matching symbols or strings is greater than the search buffer, there is a decline in efficiency of coding. Fixed size window is again another limitation as the matched strings for coding cannot be greater than the size of window, that is, size of search buffer plus history buffer.

In LZ 78 Algorithm [16], no sliding window is used, and encoded text is added to dictionary, which does not have a fixed size. Each time a code or a token is used, the encoded string is added to the existing dictionary. Once the limit to dictionary size is reached, either the dictionary is saved for future use, if the coding efficiency is good, or reset to zero, if coding efficiency is deteriorating.

Moreover, instead of using triples in code or tokens as in case of LZ 77, LZ 78 uses pairs, that is, the pointer p and the symbol C following the matched string; the length is discarded. At the decoder end, since the encoding scheme and pair-format is known, the dictionary can be reconstructed and input stream of pairs decoded to recover the actual data. This is a considerable improvement, however, the algorithm being patented, is the biggest hurdle in its common use for applications.

2.4 Implementation Methodologies

The algorithms are just a theoretical basis for achieving data compression in digital communications. These theoretical guidelines require practical implementation on platforms for achieving the desired results. There are two main platforms available for practical implementation of such like and other algorithms in the field of digital communication, namely the software platform and the hardware architectures. A hybrid methodology also exists incorporating both the software and hardware platforms as well as Reduced Instruction Set Computer (RISC) also known as microprocessor based designs. Related work in these fields will be elaborated in subsequent sub-sections.

2.4.1 Software Solutions

Data compression on software platforms imply implementing these algorithms or their guidelines in the form of software developed for the application on a general purpose computer. The advantage of this approach is that frequent improvements and up-gradations for optimized solutions may be incorporated without much of a hassle. Compression ratios may be improved by increasing the size of the dictionary, which enhances the computation power of the CPU for carrying out larger searches. However, this also means increase in CPU's processing time as well as memory usage, introducing latency. And we know that latency directly affects throughput.

A wide range of software applications and programs have been developed based on LZ based algorithms, for example, compress, lha, zoo, pkzip, gzip, V.42, gif, etc. Multiple memory accesses in software solutions are a bottleneck in high speed performance requirements, and thus, software solutions do not support high throughput data compression requirements. Despite the fact that several fast string matching techniques have been applied to accelerate compression speeds [18-20], the achieved results are far too slow for real-time applications such as wireless data networking and high speed mass storage transfers.

2.4.2 Hardware Solutions

The exploding traffic over the wired and wireless networks for data transfers and storages in orders of terabytes have deemed hardware solutions of data compression necessary [21]. The

advent of VLSI technology and parallel computation methods have further eased real-time high speed data compression to be implemented on hardware platforms, so that data may be compressed and decompressed on-the-fly [22].

In dictionary based algorithms, like LZ 77 algorithm, the most time consuming part is searching for the longest possible search, which if done by the CPU itself, bogs down all other applications. However, if the same is made to be done by a co-processor or an FPGA, the CPU is off-loaded for other computations and an accelerated increase in compression speed is achieved. A dedicated hardware designed for a specific algorithm can thus achieve faster and reliable compression as well as high throughputs. Hardware implementations on FPGAs or ASICs facilitate real-time compression and decompression of data, with throughputs in order of gigabits/sec.

2.4.3 Microprocessor Based Implementation

Reduced Instruction Set Computer (RISC) based approach has also been researched for purpose of removing redundancies in data and achieve data compression [23]. A stand-alone microprocessor is programmed and used to implement a selected algorithm. However, the approach has not been very promising and successful as large-scale implementation is not feasible owing to increased costs of memories, increased power consumptions and complex inter-connects. Moreover, such like approach also has a higher complexity in hardware-testing issues. Due to simpler designs in lower hierarchy, this approach is largely employed in scenarios where limited data is required to be handled and as such, its development remains confined to these scopes only.

2.4.4 CAM Based Implementations

In this technique, string matching is carried out by employment of Content Addressable Memories (CAMs) which results in higher speed of matching. Repeated strings of input data may be matched to parallel lookups within the CAMs, thus resulting in high-speed comparison and high throughputs. With pipelining, the CAM based architectures have been reported to achieve throughputs as high as 100 Mbits/sec. However, CAM based approach is not very popular since CAMs, after all being memories, are quite expensive.

Content Addressable Memories (CAMs) based architectures provide constant time to search the matching strings, which is a great advantage. Designs based on these architectures have proven to be quite effective [24], as a CAM-based LZ 77 data compressor can process one input symbol per clock cycle, regardless of the sizes of buffer and length of string.

A CAM based design has been discussed in [25]. This approach suggests a two-stage compression-decompression architecture, based on combined features of Parallel Dictionary LZW [26] and Adaptive Huffman (AH) algorithms. The resultant design is 4 times hardware efficient in terms of cost and beats Adaptive Huffman in most of cases. Moreover, the architecture also excels when compared to Adaptive Huffman algorithm applied on software platform, such as compress utility of UNIX system. It operates at a clock of 100 MHz, achieving compression throughputs ranging from 16.7 - 125 MB/sec and decompression rates ranging from 25-83 MB/sec. The design can be easily adapted to any FPGA. However, the mis-match between the compression and decompression rates makes it unsuitable for ideal real-time applications.

Another CAM based design has been suggested in [27]. Identifying the critical path in the architecture, pipelining of the design in matching portion and portioning of Content Addressable Memories into blocks for fully parallel searches has been been applied to gain speed at cost of area and hence, cost. The architecture has clock rate of 50 MHz, implying 50MSamples/sec can be processed. In fact, the Titan-R was termed as the fastest compressor in 2008 [28] achieving throughputs upto 8 Gbits per sec.

However, there are two disadvantages associated with this approach. First, memories are expensive. Increasing the size of window to enhance compression ratios, by allowing larger strings to be available for matching implies increased costs. Second, memories constantly consume static DC power, which, for larger memory banks, means increased power consumption and associated power dissipation problems.

2.4.5 Systolic Array Architectures

A systolic array is an arrangement of processors in an array where data flows synchronously across the array between neighbors, usually with different data flowing in different directions. Each processor at each step, takes in data from one or more neighbors, processes it and, in the

next step, outputs results in the opposite direction. The idea is to form an extended layout of identical processing elements, each having simple interconnections and capable of carrying out simple tasks. A simple layout of a systolic array is shown in Figure 2.3.

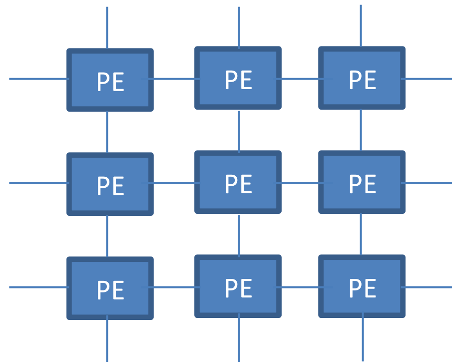

Figure 2.3: Systolic Array

Systolic architectures have the properties of being extremely fast, scalable and convert some exponential problems into linear ones. The order of computations of a serial match problem can be reduced from n^2 to n by use of n systolic processing elements, n being the length of longest match.

A number of factors support use of systolic arrays for highly intensive computational problems that are likely to degrade or even fail a software system. Firstly, VLSI technology and Moore's law has made possible smaller and faster gates, which in turn make possible high rates of communication within the chip, thus increasing processing speed. Then, these chips do not have power consumption and dissipation issues and hence are ideally suitable for hardware implementation of computationally intensive applications. Economical design and manufacture processes have made possible a considerable decrease in their prices. Finally, advanced simulation techniques ensure a fault-free chip is tested before fabrication, ensuring error free hardware. A fault-tolerance measure taken in one cell is bound to propagate to all other cells in design of a systolic design, every cell being identical to the prototype.

Whereas CAM based designs perform string matching by fully parallel searches, systolic array architectures accomplishes the same by pipelining. CAM based designs may be faster but systolic designs are cost-effective and error resistant [29]. The only de-merit of systolic architectures is that being highly specialized architectures, they are difficulty to design, build and implement.

The earliest systolic array based data compression hardware is found publication titled “High Speed VLSI Designs for Lempel-Ziv Based Data Compression” [24], published in IEEE transactions on Circuits and Systems in 1993. The proposed architecture employs thousands of processing elements (PEs) and achieves high speed and throughput for text based compression by exploiting pipelining and parallelism. Using n-processors array, it has been practically proven that the number of computations are reduced from n^n to n^2. Operating at a frequency of 40 MHz, the architecture achieves a compression rate of 13.3 Mbytes/sec.

A wrap architecture [30] based proposed design is a significant improvement upon this earliest architecture, and is based on a variant of LZ 77 Algorithm, namely the LZSS Algorithm [31]. Maximum matched length is achieved for every clock cycle and structure is able to compress and decompress the data on the fly for real-time data communication systems. Reaching an operating frequency of 91 MHz, each character can be encoded in one clock cycle. The improvements included compression time being linearly proportional to input length and a simple and modular architecture for ease of implementation and expansion. Despite being hardware complex, the design was a significant improvement over previous proposed designs.

An improvement is suggested in “Unified VLSI Systolic Array Design for LZ Data Compression” published in IEEE Transactions on VLSI Systems in 2001 [29]. The research proposes a number of processor array designs and concludes one as the most optimal. Capable of operating at clock of 150 MHZ (simulated) and 100 MHz (practically for prototype chip), throughputs of 10 MBits/sec is claimed. A number of processors may be used in parallel to exploit the parallelism in the data stream multiplying yields in throughputs, for example, 10 x processors can achieve a compressed throughput of 100MBits / sec.

Another efficient systolic array design is proposed which is high speed and area efficient in research “Design and Implementation of FPGA-based Systolic Array for LZ Data Compression” published in IEEE journal in 2007 [21]. Using 16 PEs, it was the most cost effective systolic array architecture. The systolic approach eases designing and routing in that only PE is laid out, and the rest 15 x PEs are replicated. A comparison is presented with a recent architecture on the same lines, i.e with Design-I [29] and results show a comparative improvement in terms of hardware resources once implemented on SPARTAN II XC 200 FPGA. Achieving clock speeds of 105 MHz, a single chip containing 10 PEs can process and achieve throughput of 130 MBits/sec.

Moreover, the design being flexible allows implementation of variants of LZ 77 algorithm, such as LZ 78, LZW or LZSS.

The latest research proposes a novel high speed data compression architecture embedded in an enterprise network communication system, to cater for multiple communication interfaces [32]. The systolic array arrangement is used for parallel comparisons of multiple symbols in a single clock cycle. With a layered structure catering for LAN, WAN and SAN of the enterprise network system, the scheduler optimizes these layers by directing different data streams on different layers as per their workloads. With different types of data being handled by respective layers, the architecture caters for future iterations of the algorithm in the same clock cycle and producing more than one codeword. The architecture is also subjected to pipelining and unfolding techniques to enhance throughputs to multi-giga bit rates. Operating at clock of 100 MHz and unfolding factor 2, throughputs of order of 9.17 GBit/sec have been achieved. Once the architecture is pipelined, throughputs upto 8.7 GBits/sec are achieved at a clock of 165 MHz. Synthesizing these architectures at higher clocks, such as 500 MHZ can result in throughputs greater than 10 GBits/sec.

2.5 Summary

In this chapter, we have discussed the requirement of data compression vis-à-vis the current and future data communication systems requirements. We have briefly stated various terms associated with lossless data compression, before moving on to types of lossless data compression schemes. We have briefly referred to the LZ family of Algorithms comprising the dictionary based schemes. Implementation methodologies of this family of algorithms and related work done in the field has been discussed momentarily, up to and including latest research work done so far, both on software and hardware platforms.

CHAPTER 3

LZ 77 ALGORITHM

3.1 Why Dictionary Based Schemes

Certain structural regularities are always there in data streams, especially when we deal with them in discrete domain. These regularities can be exploited to reduce the number of bits needed for either communication or storage. Prior to 1977, lossless data compression techniques were confined to the statistical modeling, where the data stream was analyzed for statistical patterns. Redundancies were deduced from these patterns and based on probability of symbols in data streams, codes were generated. Quite some work has been done in this field as already covered in Chapter 2.

These models assumed that prior knowledge of incoming symbols in data stream is known and not expected to change much during the processing time. However, when no *a priori* knowledge of source is available, a large memory is required to interlace the learning aspect of model with the coding scheme. The statistical schemes thus become either very expensive or unreliable and at times, impossible.

When Jacob Ziv and Abraham Lempel created the breakthrough in 1977 with their research about a universal compression algorithm for sequential data compression [15], a new dimension was added in the field of data compression. This concept, which was so simple and yet was waiting to be discovered and realized, was ideally suited to systems where there was no *a priori* information of the source. The concept behind algorithm was to maintain a dictionary that may be static, suited for specific applications or adaptive, that is updated continuously with the changing patterns in incoming data stream.

Take for example the text preceding this paragraph, where frequently used words are stream, scheme, data, compression, etc. The incoming symbols in the data stream are matched to those in the dictionary, and if a match is found, an index to the dictionary is transmitted instead of the steam.

If considerable matches are found from dictionary, and the number of bits for index transmission is less than number of bits required to transmit the data as it is, compression is said to have been achieved. And certain redundancies are always inherent in the data streams.

Dictionary based schemes exploit temporal locality of reference, i.e. patterns in current data stream are most likely to recur in near future. And this concept has proven to be quite true, as evident from the results obtained when applying this technique to various lossless applications.

Lossless data compression requires the solution to be strictly complying to the algorithm, whether hardware based or software based. The increasing efficiency and effectiveness of the model requires exploiting architectural optimization techniques in hardware, whereas software based platforms need to be simpler and adaptable for compiler translation.

3.2 Introduction to LZ Family

The LZ family of algorithms are named after their authors, Abraham Lempel and Jacob Ziv and based on the two researches published in 1977 and 1978 [15-16]. There have been considerable improvements to the two basic algorithms over past 35 years, and improvements such as LZW, LZSS, LZR, LZT, LZH etc have been published. The basic idea of all these algorithms is the dictionary concept, where frequently occurring strings of symbols are stored in buffers, and future matches are indexed to this dictionary. A large number of applications today are based on these algorithms, and developed as a standalone hardware or software solution, or in combination with a statistical scheme such as Huffman or Adaptive Huffman Coding. A few examples such as GIF, compress, lha, arc, zip, gzip, stacker and V.42 are important to be mentioned here.

The LZ 77 or LZ1 algorithm essentially works with a sliding window consisting of a history and a search buffer as shown in Fig 3.1. At the source end, incoming data slides in from the right, into the search buffer, coded, and shifted into history buffer and out of it. The window size is kept fixed and size of search buffer is comparatively much larger that the search buffer. The algorithm aims to find longest match of a string of symbols in search buffer starting at cursor position, from the already encoded and transmitted symbols in

the history buffer. Depending on the match found, a triple comprising the index position of match in history buffer, length of match and the first non-matching symbol after the match are transmitted. If the size of this triple is comparatively small than the symbols itself, compression is achieved.

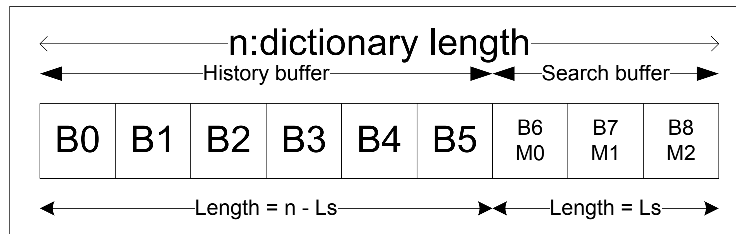

Figure 3.1: Sliding Buffers of LZ 77

In LZ 78 or LZ2, no sliding window is used; instead a coded symbol or string is iteratively and incrementally added to the dictionary. Once the preset size of dictionary is reached to the limit, the dictionary is either discarded, if compression is not optimal, or saved, if the compression is optimal. Moreover, the code-word comprises a double instead of a triple as in case of LZ 1, with only index to location and non-matching symbol being transmitted and length being discarded. However, the LZ 78 did not become as popular as LZ 77 because parts of it were patented after few years of its publication.

We shall focus on the details of LZ 77 as it forms the basis of our proposed architecture.

3.3 LZ 77 Data Compression Algorithm

The LZ 77 Data Compression Algorithm also called LZ 1, is the most popular of its family and belongs to a class of universal algorithms with results comparable to certain optimal codes. It can be easily applied on any discrete source. The underlying concept is encoding future segments of the source data by maximum-length copying from recently coded segments. The longest match found in the current string is coded by a triple, essentially an index to the dictionary and comprises buffer address, length of the matched segment and usually the first unmatched character.

Thus, a variable length segment of the input data is coded to a fixed length codeword. Using this code word and recently received symbols, the decoder reconstructs the segments being received. The transmitted code-words contain sufficient information for to

allow the decoder to recover the data on the fly. These code-words are overhead in the initial stage of the compression, as the coder builds up its dictionary. However, with subsequent transfers, less and less information is sent, enabling compression accomplishment. Data is decoded right away; as whatever code-word is received, the decoder has to reconstruct the data from its own local dictionary. There is no need for large storage buffers to retain the received code-words for processing, until data is recovered by the decoder; the dictionary automatically does it, in real-time.

Referring back to definition of Compression ratio, as long as the ratio of output to input data stream is less than 1, compression is said to be effective. There are a number of variants to LZ 77 algorithm as proposed in literature, however, we shall constrain our discussion and implementation to the basic simplified version of the LZ77 algorithm processing as mentioned in research paper [32].

3.3.1 The Algorithm

LZ 77 uses a dictionary that is a portion of recently processed data stream. The incoming string is imagined to be passing through a sliding window, from right to left. As the data slides through the window, it is compressed and the dictionary is updated, imitating a compressor scanning and processing the data segments from start till end.

This sliding window is divided into two portions, the window commonly referred to as buffer, meaning a temporary storage. The first portion is called the History Buffer and the second portion is called the search buffer. The history buffer contains recently processed symbol segments. The search buffer contains the data segment immediately following the processed data in the stream, and required to be compressed. The size of each buffer is fixed as per the designed architecture, and is an important factor affecting compression ratios achieved. The size of history buffer is comparatively much larger than that of the search buffer. A detailed study in this regard has been carried out in [33].

For each encoding step, the incoming data characters are pushed into the window, say of length N characters, from one side and basing on the number of characters encoded, out of it from other side. The window is further divided into two history and search buffers as already shown in Fig 3.1. The input data required to be coded for say E characters (s_0-s_2)

occupies the search buffer and the already encoded data of say F characters (h_0-h_7) occupies the search buffer. The first character of the look-ahead buffer (y_0) is compared with entire character string (s_0-s_7) in the search buffer, and once a match is found, the next character in search buffer (s_1) is concatenated to the first character (s_0) and the entire matching is done again to find a string-match. This process is repeated for all characters of the look-ahead buffer (s_0-s_2) if successive matches are found. Accordingly, codes are generated depending on the matched string position in history buffer and length of matched-string. The first non-matching character is appended to the code-word complete one iteration.

At the beginning of compression process, the entire window comprising history and search buffers is normally filled with zeroes. Initial steps yield nulls as characters as code-words, as characters do not match with the pre-filled zeroes of the history buffer. With each clock cycle, the data is pushed into the window sequentially, and code-words generated as per the match. The window slides by number of characters encoded in each step plus 1, in direction of incoming data stream, throwing out the same number of characters from history buffer and entering new characters of same length into the search buffer. This is the concept of the sliding window encoding method, which will be elaborated further with the help of an example when we discuss the details of encoding.

For the purpose of implementation, we will denote the starting point of history buffer where the match occurred with character of look-ahead buffer as pointer ***pntr,*** the length of the matching characters as ***len*** and the first non-matching character is denoted as ***symbl***. The three outputs are concatenated as (***pntr len symbl***) and output as a codeword. The window thus slides len +1 characters followed by a match.

Following compression, the total number of bits required for the code-word can be easily calculated by using the following notations.

Let the size of History Buffer be denoted as H.

Let the size of Search Buffer be denoted as S.

Let the size of the source alphabet / character be denoted as C. Then, size of codeword N for such a compression scheme is defined as:

$$\mathbf{N = [\log_2 H] + [\log_2 S] + [C]} \tag{3.1}$$

To sum up, the algorithm executes following 3 x steps in loops:-

a. Search and find longest match of a segment of incoming data, starting at cursor and completely contained in the search buffer to a segment contained in the history buffer.
b. Output a codeword, essentially a triple (pntr, len, 'c'), where p is the pointer indicating the location of match in the history buffer, len is the length of the matched segment and 'c' is the first un-matched symbol after the matched segment.
c. Move the cursor n + 1 characters forward.

The pseudo code for the algorithm implementation whether in hardware or software is as under:-

```
while (SearchBuffer not empty) {
get a reference (position, length) to longest match;
if (length > 0) {
output (position, length, next symbol);
shift the window length+1 positions along;
} else {
output (0, 0, first symbol in the search buffer);
shift the window 1 character along;
}
}
```

3.3.2 Encoding Process

The encoding scheme employs applying the above mentioned pseudo code to hardware architecture or a software platform. Loop handling is easy in software but quite tricky and cumbersome in hardware. Quite precision has to be taken care of while implementing such kind of recursive operations in a hardware system. In this research, we shall explain the encoding technique employed for the simplified version of the algorithm on our architecture as already mentioned.

Extending the concept outlined in the section above, let us take an example and unfold the encoding process as it happens in our proposed architecture. We take window length W = 11, search buffer H = 8 and Look-ahead buffer S=3. We assume a data string of characters

comprising the sentence “this thesis deals with data compression” as our input string and subject it to compression technique.

The coding process is explained in Table 3.1 below; each row simulates a step for entire compression process, including comparison, calculating the required code-word and shifting of window before this process is repeated. h0 – h7 depicts the characters in the search buffer whereas s0 – s2 depict the characters in the history buffer. Window length is same as used in our research and already mentioned above. The steps are shown as under and are self-explanatory.

h0	**h1**	**h2**	**h3**	**h4**	**h5**	**h6**	**h7**	**s0**	**s1**	**s2**	**Code Out (p, l, “c”)**	**Shift by (l+1)**
0	0	0	0	0	0	0	0	t	h	i	0,0,t	1
0	0	0	0	0	0	0	t	h	i	s	0,0,h	1
0	0	0	0	0	0	t	h	i	s	t	0,0,i	1
0	0	0	0	0	t	h	i	s	t	h	0,0,s	1
0	0	0	0	t	h	i	s	t	h	e	4,2,e	3
0	t	h	i	s	t	h	e	s	i	s	4,1,i	2
h	i	s	t	h	e	s	i	s	i	s	2,3,o	4
h	e	s	i	s	i	s	o	n	d	a	0,0,n	1
e	s	i	s	i	s	o	n	d	a	t	0,0,d	1
s	i	s	i	s	o	n	d	a	t	a	0,0,a	1
i	s	i	s	o	n	d	a	t	a	c	0,0,t	1
s	i	s	o	n	d	a	t	a	c	o	2,1,c	2
s	o	n	d	a	t	a	c	o	m	p	7,1,m	2
n	d	a	t	a	c	o	m	p	r	e	0,0,p	1
d	a	t	a	c	o	m	p	r	e	s	O,0,r	1
a	t	a	c	o	m	p	r	e	s	s	0,0,e	1
t	a	c	o	m	p	r	e	s	s	i	0,0,s	1
a	c	o	m	p	r	e	s	s	I	o	1,1,i	2
o	m	p	r	e	s	s	i	o	n		8,1,n	2

Table 3.1 : Encoding Process of LZ 77 Algorithm

Instead of the string characters being transmitted as such, the code-words are transmitted. We have assumed the text to be in ASCII (American Standard Code for

Information Interchange), hence requiring 8-bits for each character. The length "**N**" for code-word can be easily calculated by recalling Equation 3.1 as under:

$$\mathbf{N = [\log_2 H] + [\log_2 S] + [C]} \qquad (3.1)$$

Having History Buffer Size H = 8, Search Buffer Size S = 8 and Character C=8, we get N as:-

$N = [\log_2 8] + [\log_2 8] + [8]$

$N = 3 + 3 + 8 = 14$ bits

This result in use of 14 bits-encoding as against transmission of 32 bits, 3 x 8-bit characters encoded and 1 x 8-bit character appended, resulting in maximum compression ratio of 44 %. Experimenting with different sizes of History and Search Buffers, depending on type of input data, considerable compression ratios may be achieved. These codes may be used as such, or further subjected to other techniques of encoding and/or encryption, enhancing integrity of data and efficiency of channel bandwidth.

3.3.3 Decoding Process

At the receiver end, the code-words are received as transmitted by the transmitter. If the code-words were further subjected to another technique of encoding and/or encryption, the reverse of these techniques are applied to recover the actual code-word required for recovering the string of characters for LZ 77 decoding.

Code-words are then used to recover the actual characters of the string on the fly. It may not be out of point to mention here that the decoding process is automatic; there is absolutely no delay in recovering the characters from the code-words, rather the characters are generated as received. The dictionary is built, starting from all nulls initially, and maintained at the receiver end. Using the last received and decoded characters present in the dictionary and the new code-words received, the decoder reveals the new characters, simply replicating them from the pointers and lengths received and regenerating the data stream.

For example, let us take the initial few steps of decoding, starting from the code-words received and inferring the characters referred to by them as in Table 3.2 below.

Code Out (p, l, “c”)	s0	s1	s2	s3	s4	s5	s6	s7	h0	h1	h2
0,0,t	0	0	0	0	0	0	0	0	t		
0,0,h	0	0	0	0	0	0	0	t	h		
0,0,i	0	0	0	0	0	0	t	h	i		
0,0,s	0	0	0	0	0	t	h	i	s		
4,2,e	0	0	0	0	t	h	i	s	t	h	e
4,1,i	0	t	h	i	s	t	h	e	s	i	

Table 3.2 : Decoding Process

3.4 Performance and Limitations of LZ 77 Algorithm

The LZ 77 data compression algorithm produces optimal results on most text-based lossless data compression applications. The results of-course varies with the size of the window and in turn the size of the history and search buffers used. The disadvantage of LZ 77 algorithm is that it is fairly efficient only for long files or messages; shorter messages or strings usually result in overheads in transmission or storage, since considerable matches cannot be found and hence the number of bits required after processing are more than actual. If the repeating characters in the string occur with a period greater than the size of the search window, the performance obviously degrades. In most practical applications, size of the history buffer is usually 8 192 bits and size of the search buffer is about 10 to 20 bits.

The code-words initially tend to expand the transmission or storage requirement as dictionary is building up. Consider for example, as in Table 3.1, the replacement of a 14-bit codeword against an 8-bit character in a string. However, as the dictionaries build-up, the overheads decrease and more and more characters are encoded and thus less information is required to help the receiver decode the code-word. The process then approaches optimization and hence achieving compression.

3.5 Summary

In this chapter, we have introduced and discussed in detail the advantage of dictionary based schemes over the traditional statistical based schemes. We have introduced the LZ family of algorithms with details on structure, encoding and decoding process of the basic LZ 77 Data Compression Algorithm. The scheme has been explained in detail using an example as relevant to our implementation methodology. We have also touched upon the performance and limitations of the LZ 77 algorithm which has been used in our proposed hardware.

CHAPTER 4

ARCHITECTURAL DESIGN OF PROPOSED METHODOLOGY

4.1 Introduction

Having discussed the structure of LZ 77 algorithm in detail in the preceding chapter, we now move on to its implementation. The algorithm as already discussed may be implemented in hardware or software, each having its merits and demerits. However, by and large, we saw in Chapter 2 that hardware based implementation is relatively beneficial, efficient and more reliable. Data compression implementation is relatively easy in hardware as dedicated FPGAs may be employed to carry-out the relatively complex and time-consuming task of comparisons, off-loading the general purpose CPU of the computer. In this way, the main processor is available for other tasks, ensuring saving of precious clock cycles and contributing to transmission and processing efficiency.

To achieve high throughputs, we have explored architectural optimizations to convert serial processing into parallel comparisons. Processing of existing data as well as future data strings / character concept will be used within the same clock cycle. Previous researches have mostly focused on a single match per clock cycle whereas our research extends this concept to future matches within same clock cycle, hence contributing to increased throughputs.

We have developed our design using the bottom-up approach, starting from basic comparison architecture, and building up hierarchical modules to evolve the complete architecture. Our design is systolic array based architecture, replicating the basic comparison architecture in multiple iterations. We utilize the advantages of systolic architectures to achieve simplicity and flexibility in design in order to have increased throughputs.

This chapter starts with a brief introduction to systolic architectures, their pros and cons vis-à-vis their application, before moving on to unfold own design including the proposed basic hardware architecture performing the comparisons for the purpose of data

compression. The architecture is scalable and flexible to be implemented in various applications that require economy in bandwidth and storage. We shall dwell upon our basic compression architecture in this chapter, and cover the details of unfolded and pipelined architectures in next chapter.

4.2 Why Systolic Architectures

Having already discussed in detail in Chapter 2, within the orbat of hardware platforms, systolic architectures are an obvious choice when compared to CAM based and other such like approaches. CAM based design carryout fully parallel string matching whereas Systolic designs do the same through pipelining. Moreover, CAM based designs are comparatively faster yet systolic designs are not only cheaper to implement but also error resistant when tested.

With the development in field of digital systems and Very Large Scale Integration (VLSI) and Wafer Scale Integration (WSI) technologies increasing the gate density of chips, their production cost are reduced dramatically. Pipelining and parallelism have further eased use of such specialized chips and ever increasing throughputs are now achievable. The complexity of such architectures remains a problem, but nevertheless, it is workable. Systolic architectures tend to further increase the ease and efficiency of specialized VLSI chips, developed to handle computationally complex problems. Before we go into the finer details of systolic architectures, let us first go through the underlying process of chip designing and implementation of algorithms that necessitate their use.

4.2.1 Systolic Architectures Elaborated

The concept of systolic architectures was first introduced by Kung in [11] as a means to implement computationally intensive repetitive applications requiring local communication. Such architectures exploited features of regularity, modularity, rhythmic, synchronous and concurrency of processes in application required to be implemented.

The term systolic is coined from physiology of a living heart that contracts in a rhythmic fashion to regularly supply blood to the body. Analogous to this, systolic elements perform specified tasks in a rhythmic, repetitive and regular manner, taking inputs and processing them to produce desired outputs.

A simple definition of systolic array architecture as given from literature is, *“Imagine n simple processors arranged in a row or an array and connected in such a manner that each processor may exchange information with only its neighbors to the right and left. The processors at either end of the row are used for input and output. Such a machine constitutes the simplest example of a systolic array.”*[34]

Systolic architectures have grid-like multiple processing elements (PEs) interconnected by simple communication paths and networked to form an array, as shown in Fig 4.1 The PEs are balanced, uniform and identical, each capable of performing a specified simple task. Each PE is interconnected to other PEs through simple wired connection. Simple interconnections do not only contribute to increased computations but also facilitate design and implementation.

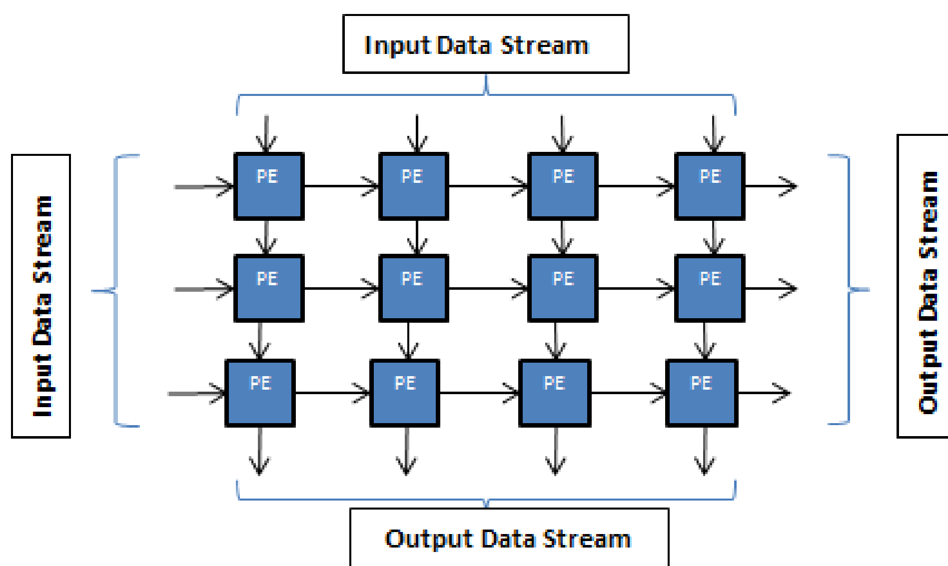

Fig 4.1 Simplified Systolic Array

Data within the architecture flows at multiple speeds in multiple directions. Speed is achieved in the design through register-to-register transfer of data. And then all PEs are processing at a central common clock, and each PE including those at boundary of array is I/O capable. Thus, systolic array architectures have a very high I/O rate and aptly suited to computationally intensive applications, implemented in a very simplistic manner.

4.2.2 Issues in Special Purpose Chip Design

VLSI chip designing is a complex problem and needs deliberate and detailed discussion of various issues, starting from conceiving its design to its test and production. We shall not go into details of these at the moment, but just briefly touch upon the important ones that shall form the basis for understanding our proposed design.

The first thing that must be taken into consideration before designing a VLSI chip specialized for a specific task is identification of that task that needs to be implemented. Then the design must be divided into few simple modules with relatively simple interfaces, so that costs on designs and testing can be reduced. The modularity helps modifying the design and adapt to changing environment. Then we focus onto the inter-module communication and ensure concurrency in communication and coordination between different modules.

The last step is to balance the chip computations with input and output (I/O) of the design. Frequent I/O accesses imply latency in the design, whereas, recursive computations with the same data imply data storage within the design, hence increased memory requirements, which in turn imply increased cost. This is where systolic architectures come as a rescue.

4.3 Proposed Compression Architecture

We start unfolding our design, explaining the various constituent blocks that have been used. Using a bottom up approach, we have proposed small basic building blocks, and connected and re-used the same for developing a higher-order architecture that carries out parallel matching and achieves multi-gigabit throughputs. The various blocks include namely Compression Cell block, Column Logic block, Pointer and Length Calculation block, and explained in succeeding paragraphs below.

4.3.1 The Compression Cell

Our most basic module is a simple comparison block that reduces the comparison of two characters to a single bit, i.e. the compression cell. We have realized the sequential process of comparisons as parallel comparisons for achieving increased throughput. We

have considered our input to design to be 8-bit American Standard Code for Information Exchange (ASCII) or Extended ASCII characters. This means there are 2^8 or 256 possible characters that can be fed to the design for the purpose of compression. The input string thus comprises of characters from these 256 different characters.

During the process of compression, each of these characters in search buffer shall be compared with those of history buffers. This comparison is carried out by a comparator that compares each of the 8-bits of a character in search buffer with the corresponding 8-bits of the characters in history buffer, and basing on the result of the comparisons, decides whether a match has occurred or not.

The bit-to-bit comparison is done using Exclusive-NOR (XNOR) gate which gives an active HIGH (1) output if both the inputs are matched. This ensures that the each of the 8-bits of both the characters of the search and the history buffers are truly compared. The 8-outputs of individual XNOR gates are then fed to an 8-input AND gate, to achieve a single bit output. The output of the AND gate shall always be active high (1) if and only if, both the characters are exactly matched. In net effect, the two characters are compared in totality, and a single-bit output is produced, which is then used for further processing. For characters s_0 and h_0 in search and history buffers, this is illustrated in Figure 4.2 below.

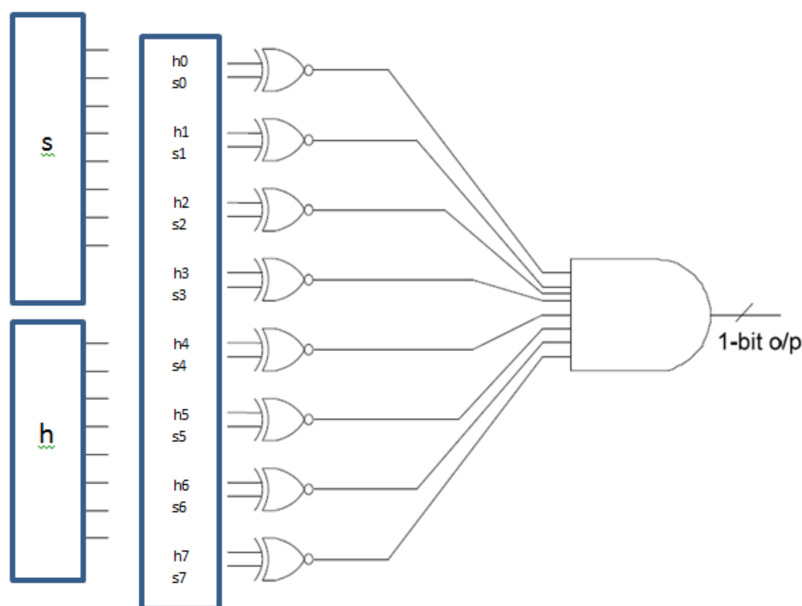

Figure 4.2 – Compression Cell

This comparator has been used in our proposed design, and in fact, forms the basis of a systolic architecture used here onwards. Replicating this compression cell and carrying out matching in parallel, multiple characters of the search buffer are matched with entire contents of already encoded history buffer simultaneously.

Using a sliding window with history buffer size h = 8, and a search buffer size s = 3, we carryout multiple searches for longest match. Let the characters in history buffer be h_0-h_7 and characters in search buffer be s_0-s_2. All the characters of the search buffer are compared with 3 characters of history buffer at a time, and iteration is repeated for all sequential 3-characters combinations of history buffer. The various matching schemes are as shown in Table 4.1 below.

Ser	**Character Matching Pairs in each Iteration**										
	History Buffer Characters								**History Buffer Characters**		
0	h0 s0	h1 s1	h2 s2	h3	h4	h5	h6	h7	s0	s1	s2
1		h1 s0	h2 s1	h3 s2							
2			h2 s0	h3 s1	h4 s2						
3				h3 s0	h4 s1	h5 s2					
4					h4 s0	h5 s1	h6 s2				
5						h5 s0	h6 s1	h7 s2			
6							h6 s0	h7 s1	s0 s2		
7								h7 s0	s0 s1	s1 s2	

Table 4.1 – Parallel Comparisons in LZ 77 Algorithm Implementation

There are a total of 24 x essential 2-character comparisons that have to be made in order to search for the longest match of search buffer characters from already encoded history buffer characters. 3 x compression cells are employed in each of the rows for this purpose. All these searches are performed in a single cycle of a clock and produce lengths of matches that occurred and the location of match in the row where it occurred. A simple mechanism may then be employed to select the row that generates the optimum codeword.

4.3.2 Column Logic

To manage execution of all these iterations for obtaining an optimum codeword of longest possible match, we have incorporated a circuit that gives us occurrence of a match and its location in a specified row. When a match is found in any iteration, the architecture computes its length (0 for no match, 3 for longest match) and its pointer (0-7), or specific row where the match occurred. Each comparator in a row compares two characters and the output if led to a circuit that carries forward the match result and calculates the pointer to the row where the match occurred.

As we can see from Table 3.1, the first character s_0 of search buffer is being compared with all the preceding characters present in history buffer, from h_0-h_7 and thus forms the first sub-column of our architecture. Similarly, s_1 is compared with all 8 x characters preceding it, h_1-h_7 ,s_1 and thus forms the second sub-column of our architecture, and so on for s_2 . Moreover, the output of each comparator is carried forward across the row over a cascaded AND gate incorporated so that maximum length in a row may be calculated. This is explained in Fig 4.2.

The outputs of each of these comparators are then given to a column logic circuit, which carries forward the status of matches in the first sub-column and the row in which the match occurred. The column logic circuit essentially carries forward the status of a match in a sub-column, which is set to 1 if a match is found, and its location in the row where the match occurred, or a pointer.

A number of designs for column logic may be conceived, each having its own merits and demerits, with the aim of carrying forward the occurrence of a match and its location. A

serial optimized logic implementation exists in [32]. We have proposed a novel optimization where the serial logic is made to work in pairs and results in logarithmic complexity, as shown in Figure 4.3.

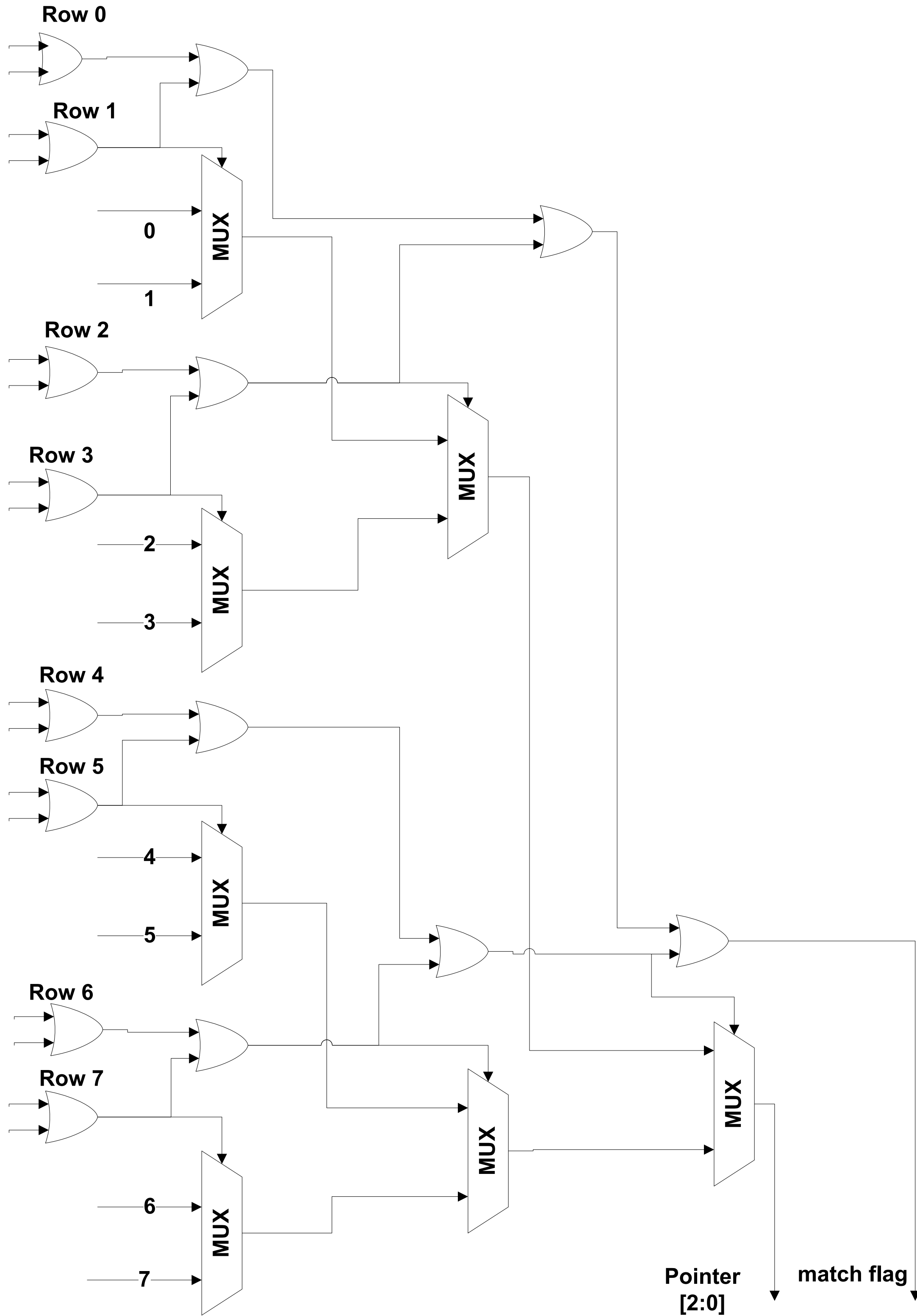

Figure 4.3 Column Logic with Logarithmic Complexity

4.3.3 Pointer and Length Calculation

Having executed all the above mentioned iterations in parallel, we need to select the optimum pair of length of pointer that gives us the longest possible sequential match. Each of the sub-columns gives us a match flag and a pointer. One iteration of the algorithm with the given window size gives us three match flags and three pointers, which are then selected through a Pointer and Length Calculation block to give us a pair for optimized codeword . Figure 4.4 shows the proposed block for this purpose, which is one of the most important blocks of our proposed design.

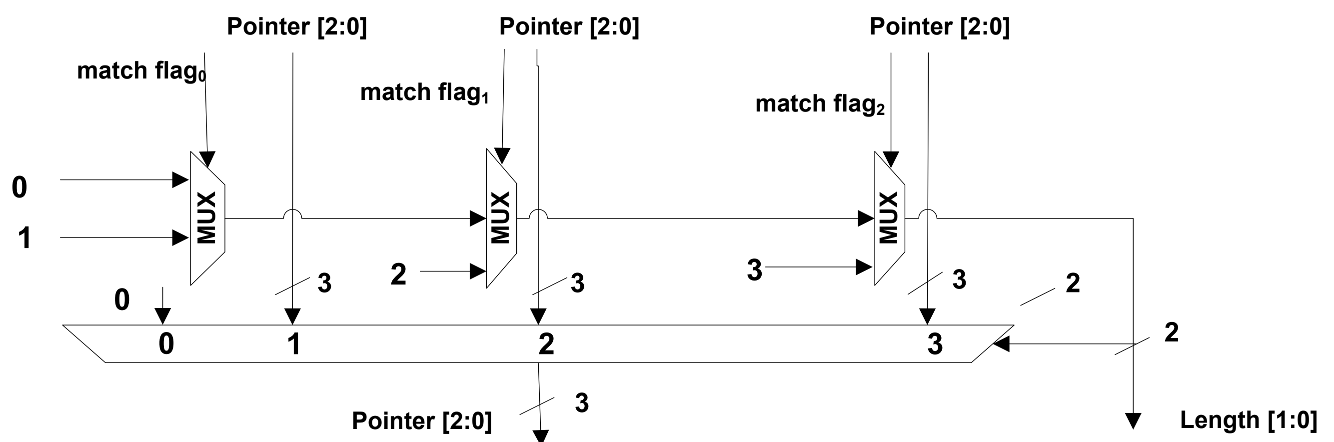

Fig 4.4 Pointer and Length Calculation Block

The lengths of history buffer may be increased to achieve better compression ratios, however, larger history buffers have their effect on the critical path of Pointer and Length Calculation block. A typical length of 8-32 characters of history buffers produce optimal results as quoted in [27], [29], [35].

4.3.4 Compression Block

Together, the Compression Cell (CC), Column Logic (CL) and Pointer and Length Calculation (PLC) blocks constitute a Compression Block (CB) of proposed architecture for executing a single iteration of the LZ77 Algorithm. The various interconnects are shown in Fig 4.5 below. Characters from search and history buffers are fed to the parallel CCs, through CLs and finally to the PLCs, which calculate the desired pointers and lengths of matches. This architecture is fully parallel and designed to run on a single clock cycle.

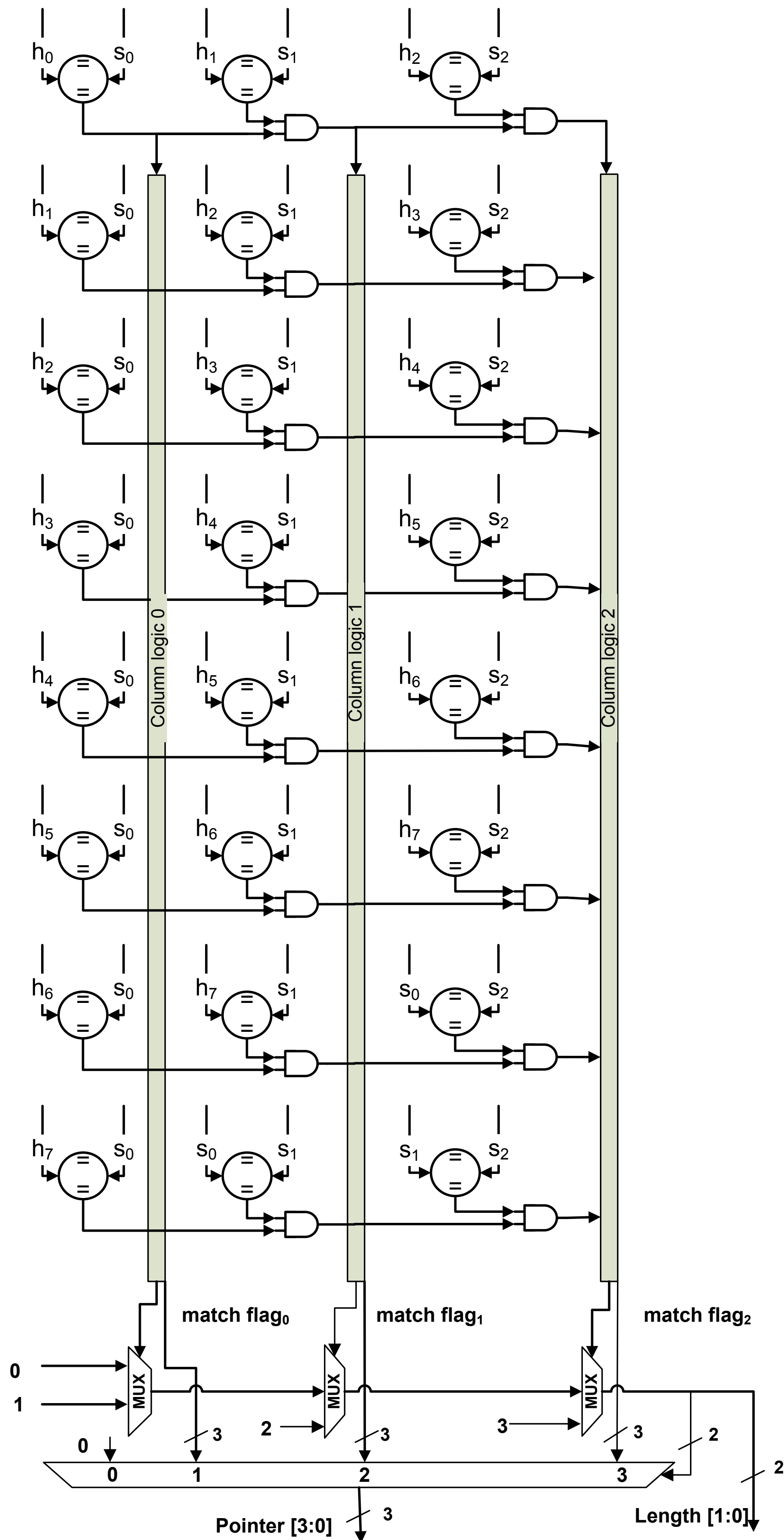

Fig 4.5 Compression Block for Single Iteration

The proposed design compares contents of entire search buffer with that of history buffer in a single clock cycle. The length of code-word for proposed design is 13 bits, with 3-bits for pointer, 2-bits for length and 8-bits for the appended character. This code-word can achieve a maximum compression of 41 % Compression Ratio if all 3 characters in search buffer are matched.

Various lengths of history and search buffers may produce different results, with considerable decrease in performance for larger buffer sizes, due to increased critical paths and hence execution timings. This architecture may be unfolded using barrel shifters to shift encoded characters from search buffer into history buffer and out of it, as the coding and compression process continues. At the same time, new characters are fed into the search buffer from incoming data stream.

4.4 Summary

In this chapter, we have introduced our proposed design for implementing LZ77 Data Compression Algorithm. The proposed architecture has been designed using a bottom up approach and various constituent blocks have been discussed in detail. The entire Compression Block for single Iteration execution has been revealed in the last portion of the chapter. The architecture optimizes the searches, and converts serial sequential matches into parallel matches, saving a lot in execution time. This architecture is a fully parallel architecture and works to compress the entire contents of the search buffer in a single clock cycle.

CHAPTER 5

SUPER UNFOLDED AND PIPELINED ARCHITECTURES

5.1 Unfolding of Architecture

Systolic architectures as we have already elaborated on in Chapter 4, are regular arrays of simple finite state machines (FSM) where each FSM in the array is identical. The underlying algorithm in the architecture is same and relies on data from different directions at regular intervals and outputs are combined [36].

The sliding window in the proposed architecture has specifications as history buffer h=8, search buffer s=3 and a window size W=11. The design is capable of encoding three 8-bit characters from the search buffer at a time in one clock cycle. The replicating of the design to form a fully parallel systolic architecture can increase its throughput many-folds.

Depending on the number of characters encoded, the data fed into search buffer varies from one character at minimum (case when there is no match) to 4 characters at maximum (when all 3 characters are encoded and 4th character is appended in the codeword). In simple terms, length of the match would force the window to slide "l+1" characters ahead, where "l" is the length of match found. This analogy may be further utilized to unfold the proposed design to include future matches of the window, based on the present matches, and processed in the same clock cycle.

5.2 Super Unfolded Architecture

The idea mentioned in the paragraph above is used to extend this design into multiple parallel stages, where the search buffer is extended to include future iterations of the matching in present clock cycle. This is the proposed systolic architecture that we shall be using for purpose of LZ 77 data compression and forms one of the complete designs of this research.

Consider Table 5.1 below, using the same specifications W=11, h=8 and s=3. If all 3 characters are matched, the window slides 4 characters forward. The six columns cater for

an unfolding by factor of 2 and eight search characters s_0-s_7. For a higher order unfolding, for example 3, the number of columns would increase to 11 and search buffer characters include s_0-s_{10}. This search scheme may be further extended to include further future searches, and all searches are carried out in parallel and in same clock cycle. The unfolding factor is simply determined by lengths of history and search buffers.

Col 1	Col 2	Col 3	Col 4	Col 5	Col 6
$h_0h_1h_2$ $s_0s_1s_2$	$h_1h_2h_3$ $s_1s_2s_3$	$h_2h_3h_4$ $s_2s_3s_4$	$h_3h_4h_5$ $s_3s_4s_5$	$h_4h_5h_6$ $s_4s_5s_6$	$h_5h_6h_7$ $s_5s_6s_7$
$h_1h_2h_3$ $s_0s_1s_2$	$h_2h_3h_4$ $s_1s_2s_3$	$h_3h_4h_5$ $s_2s_3s_4$	$h_4h_5h_6$ $s_3s_4s_5$	$h_5h_6h_7$ $s_4s_5s_6$	$h_6h_7s_0$ $s_5s_6s_7$
$h_2h_3h_4$ $s_0s_1s_2$	$h_3h_4h_5$ $s_1s_2s_3$	$h_4h_5h_6$ $s_2s_3s_4$	$h_5h_6h_7$ $s_3s_4s_5$	$h_6h_7s_0$ $s_4s_5s_6$	$h_7s_0s_1$ $s_5s_6s_7$
$h_3h_4h_5$ $s_0s_1s_2$	$h_4h_5h_6$ $s_1s_2s_3$	$h_5h_6h_7$ $s_2s_3s_4$	$h_6h_7s_0$ $s_3s_4s_5$	$h_7s_0s_1$ $s_4s_5s_6$	$s_0s_1s_2$ $s_5s_6s_7$
$h_4h_5h_6$ $s_0s_1s_2$	$h_5h_6h_7$ $s_1s_2s_3$	$h_6h_7s_0$ $s_2s_3s_4$	$h_7s_0s_1$ $s_3s_4s_5$	$s_0s_1s_2$ $s_4s_5s_6$	$s_1s_2s_3$ $s_5s_6s_7$
$h_5h_6h_7$ $s_0s_1s_2$	$h_6h_7s_0$ $s_1s_2s_3$	$h_7s_0s_1$ $s_2s_3s_4$	$s_0s_1s_2$ $s_3s_4s_5$	$s_1s_2s_3$ $s_4s_5s_6$	$s_2s_3s_4$ $s_5s_6s_7$
$h_6h_7s_0$ $s_0s_1s_2$	$h_7s_0s_1$ $s_1s_2s_3$	$s_0s_1s_2$ $s_2s_3s_4$	$s_1s_2s_3$ $s_3s_4s_5$	$s_2s_3s_4$ $s_4s_5s_6$	$s_3s_4s_5$ $s_5s_6s_7$
$h_7s_0s_1$ $s_0s_1s_2$	$s_0s_1s_2$ $s_1s_2s_3$	$s_1s_2s_3$ $s_2s_3s_4$	$s_2s_3s_4$ $s_3s_4s_5$	$s_3s_4s_5$ $s_4s_5s_6$	$s_4s_5s_6$ $s_5s_6s_7$

Table 5.1 - Multiple Iterations of Algorithm with Parallel Comparisons

The first column is same as explained in Chapter 4; it contains all parallel comparisons of history and search buffers as in a simple unfolded architecture. Now depending on the number of characters matched, the window will slide forward and search will be carried out in either of column 2 through 4. When there is no match of search buffer with any character in history buffer in the first iteration of the algorithm, symbol s_0 is shifted in the

history buffer, by just being appended to the codeword. The second iteration of the algorithm is executed in column 2 of Table 5.2, with all parallel matches shown with the same scheme.

If the first column search results in matching of s_0 only, the next symbol s_1 is appended, and subsequent iteration of the algorithm is executed as shown in Table 3. Similarly, if s_0-s_1 are matched in first iteration of the algorithm, the next symbol s_2 is appended, and second iteration of the algorithm is executed as shown in Column 3 of Table 5.1. And for a best possible scenario, where all the characters of search buffer s_0-s_2 are matched in first iteration of the algorithm, s_3 is appended to the codeword and subsequent iteration is executed as shown in Column 4 of table 5.1.

Having employed the same scheme, future iterations are also carried out in the current clock cycle, achieving encoding of more than one word in the same clock cycle. Thus, future strings are encoded by unfolding the algorithm, and number of strings encoded is directly dependent on the level of unfolding used.

5.3 Optimized Super Unfolded Architecture

Having discussed the details of unfolded design, a close observation of Table 5.1 reveals that one-third of matches in subsequent columns are identical to matches in the preceding columns. The results of these matches are already available to us at the end of the previous iteration, and hence, need not to be re-processed again. The matches in each preceding column which are redundant for next subsequent columns are colored blue for easy identification and shown in Table 5.2. Thus the results of previous iteration are simply carried forward to the next subsequent iteration for use in the ***Length and pointer calculation*** block.

Col 1	Col 2	Col 3	Col 4	Col 5	Col 6
$h_0\,h_1h_2$ $s_0\,s_1s_2$	$h_1h_2h_3$ $s_1s_2s_3$	$h_2h_3h_4$ $s_2s_3s_4$	$h_3h_4h_5$ $s_3s_4s_5$	$h_4h_5h_6$ $s_4s_5s_6$	$h_5h_6h_7$ $s_5s_6s_7$
$h_1\,h_2h_3$ $s_0\,s_1s_2$	$h_2h_3h_4$ $s_1s_2s_3$	$h_3h_4h_5$ $s_2s_3s_4$	$h_4h_5h_6$ $s_3s_4s_5$	$h_5h_6h_7$ $s_4s_5s_6$	$h_6h_7s_0$ $s_5s_6s_7$
$h_2\,h_3h_4$ $s_0\,s_1s_2$	$h_3h_4h_5$ $s_1s_2s_3$	$h_4h_5h_6$ $s_2s_3s_4$	$h_5h_6h_7$ $s_3s_4s_5$	$h_6h_7s_0$ $s_4s_5s_6$	$h_7s_0s_1$ $s_5s_6s_7$
$h_3\,h_4h_5$ $s_0\,s_1s_2$	$h_4h_5h_6$ $s_1s_2s_3$	$h_5h_6h_7$ $s_2s_3s_4$	$h_6h_7s_0$ $s_3s_4s_5$	$h_7s_0s_1$ $s_4s_5s_6$	$s_0s_1s_2$ $s_5s_6s_7$
$h_4\,h_5h_6$ $s_0\,s_1s_2$	$h_5h_6h_7$ $s_1s_2s_3$	$h_6h_7s_0$ $s_2s_3s_4$	$h_7s_0s_1$ $s_3s_4s_5$	$s_0s_1s_2$ $s_4s_5s_6$	$s_1s_2s_3$ $s_5s_6s_7$
$h_5\,h_6h_7$ $s_0\,s_1s_2$	$h_6h_7s_0$ $s_1s_2s_3$	$h_7s_0s_1$ $s_2s_3s_4$	$s_0s_1s_2$ $s_3s_4s_5$	$s_1s_2s_3$ $s_4s_5s_6$	$s_2s_3s_4$ $s_5s_6s_7$
$h_6\,h_7s_0$ $s_0\,s_1s_2$	$h_7s_0s_1$ $s_1s_2s_3$	$s_0s_1s_2$ $s_2s_3s_4$	$s_1s_2s_3$ $s_3s_4s_5$	$s_2s_3s_4$ $s_4s_5s_6$	$s_3s_4s_5$ $s_5s_6s_7$
$h_7\,s_0s_1$ $s_0\,s_1s_2$	$s_0s_1s_2$ $s_1s_2s_3$	$s_1s_2s_3$ $s_2s_3s_4$	$s_2s_3s_4$ $s_3s_4s_5$	$s_3s_4s_5$ $s_4s_5s_6$	$s_4s_5s_6$ $s_5s_6s_7$

Table 5.2: Identification of Redundant Comparisons

These redundant matches are not redundant for only the next subsequent column, but also for also for further subsequent columns. Removing the matches already carried out previously in future iterations helps reduce the number of matches for the unfolded design considerably. In fact, for the given window, with W=11, h=8 and s=3, the number of matches are reduced from 4x8x6 =192(literal) to 72 (practical/optimized).

The optimized architecture in our design utilizes this advantage and redundant matches are discarded. This analogy helps reduce area of optimized design as compared to a simpler form of repeated blocks. The details of the optimized architecture are shown in Figure 5.1. The Length and Pointer Calculation blocks are interconnected to carry forward

the results of previous matches to next subsequent blocks. A simple logic may then be used to select the required pointers and lengths.

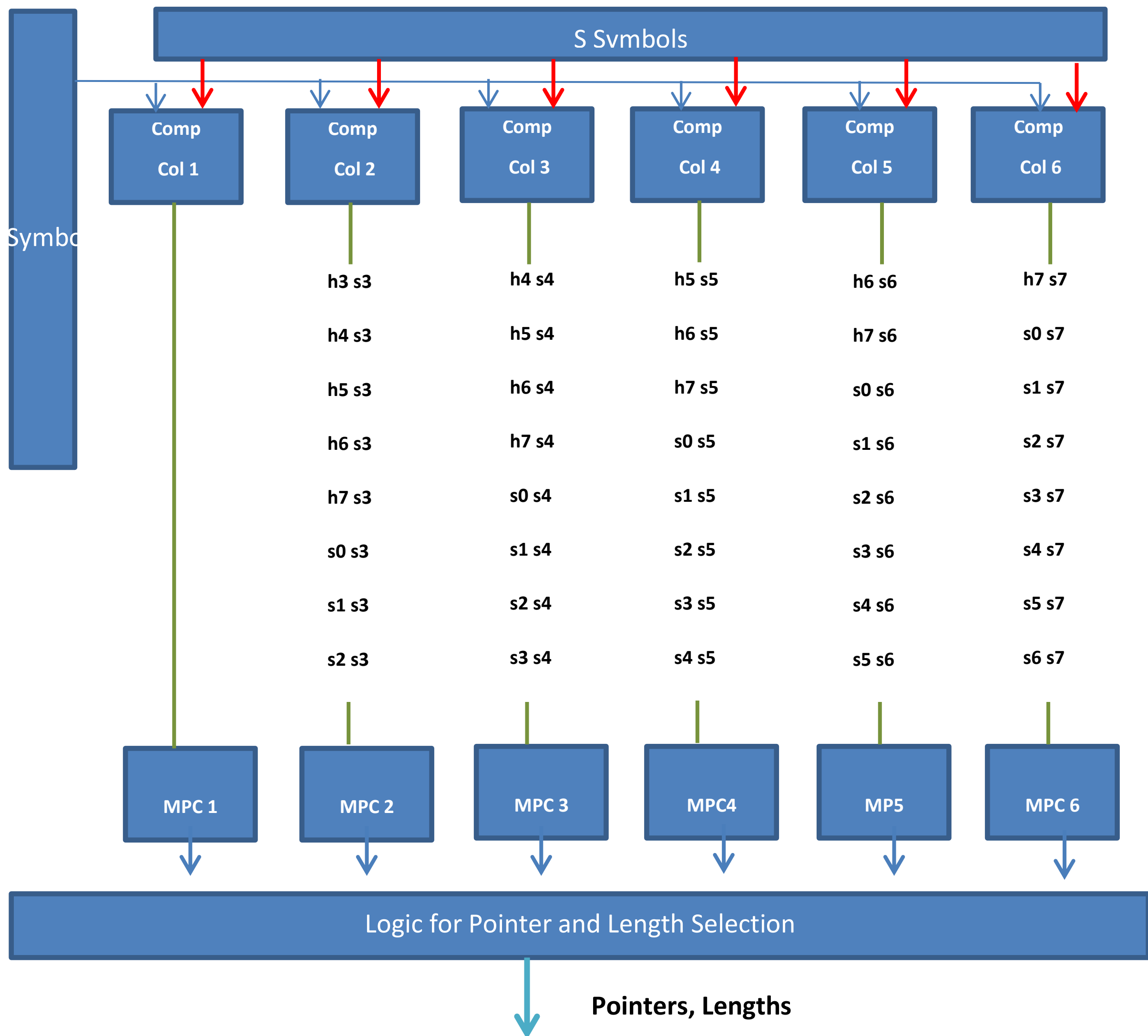

Fig 5.1 High Speed Super Unfolded Architecture

In each of the subsequent iteration, we just need to incorporate one sub-column of comparators to match the new characters, whereas the previous two sub-columns being redundant are not used, and their results from previous iterations are carried forward to this iteration for utilizing in Pointer Calculation block. Except for the first column, each subsequent new column implies just one sub-column of new characters to be matched, thus achieving a much anticipated reduction in terms of area.

The overall advantage achieved by this optimized design is not only in terms of space, but also in speed, as future strings are also encoded in the same clock cycle. Increasing the length of window, and thus lengths of history and search buffers, and unfolding the design by higher orders, the throughputs achieved are very high as compared to proposed designs in existing literature so far.

This design helps achieve boosting of throughputs and ideally suited for high speed communication applications which are bandwidth thirsty. Either devices work more efficiently in the allotted Bandwidths or they are able to work at virtually enhanced Bandwidths than existing Bandwidths provided.

5.4 Summary

In this chapter, we have further elaborated upon the proposed design and taken it two steps ahead, first by unfolding it, and later on optimizing the super-unfolded architecture. Both the designs are discussed in detail in this chapter. The redundancies introduced in unfolding of design are identified and matching is optimized to achieve reduction in not only in terms of area, but also in terms of speed. Pipelining of the design helps further reduce the critical time and achieve even higher operating clocks, contributing to higher throughputs.

CHAPTER 6

EXPERIMENTAL RESULTS AND COMPARISONS

6.1 Results

This research was aimed at proposing a novel architecture for high speed lossless data compression. Considerable improvement and enhancement of throughputs was perceived to be desired from proposed design. The architecture is a systolic array implementation and inherits the obvious advantages of simplicity, flexibility and error-resistant.

For its obvious advantages of adaptive dictionary and independence of apriori knowledge of data stream, LZ 77 data compression algorithm is has been used for implementation. In our experiment, we have used a Window Size N=11 characters, with History Buffer H=8 characters and Search Buffer S=3 characters. The proposed design was conceived and developed in a step-by-step approach. Thorough study and analysis of design has led to optimization of the architecture and redundant modules have been eliminated, which results in simplifying computational complexity, saving critical area and power requirements.

The design has then been unfolded to order 2, with future searches of Search Buffer incorporated in the current clock cycle. Whereas the algorithm lays down basis for a limited search equaling the size of search buffer in a single clock cycle, the same idea has been extended by incorporating future searches within the same clock cycle, and results obtained in parallel. This unfolded design ensures encoding of 8 characters (s_0 – s_7) in a single clock, hence enhancing throughput manifolds.

The proposed design has been extensively simulated and synthesized on various families of FPGAs using Model Sim ® and Xilinx ® platforms respectively for all possible values of data including “WORST” and “BEST” case scenarios. The inputs and outputs to the architecture are shown in Figures below. There is a considerable reduction in terms of area as well as increase in terms of speed as already claimed in preceding chapters.

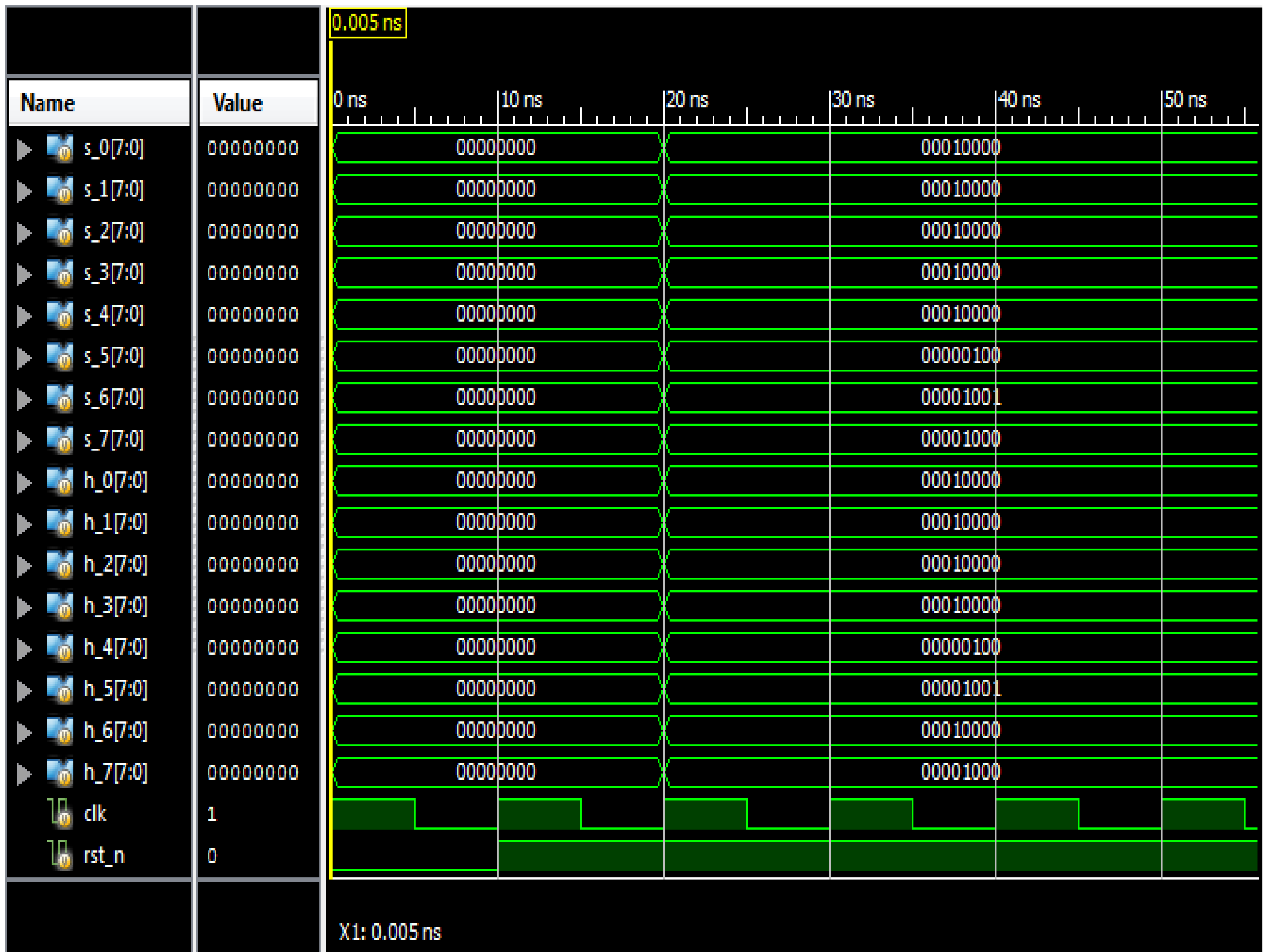

Figure 6.1 Simulated Design with Random Input Data

The inputs are passed to Registers before being input to the design. 8-bit Registers H_0 – H_7 contain the history buffer characters whereas 8-bit Registers S_0 – S_7 contain the Search Buffer characters required to be encoded. Maximum match length of 3 characters entails 6 x 2-bit Registers "length1-length6" to be used for storing results from respective iteration. The length of history buffer of 8 characters entails 6 x 3-bit Registers for storing pointers from respective iterations. The initializing followed by inputs to registers after 20 ns is shown in Figure 6.1 above. Accordingly, the outputs stored in length/pointer registers are shown in Figure 6.2 below.

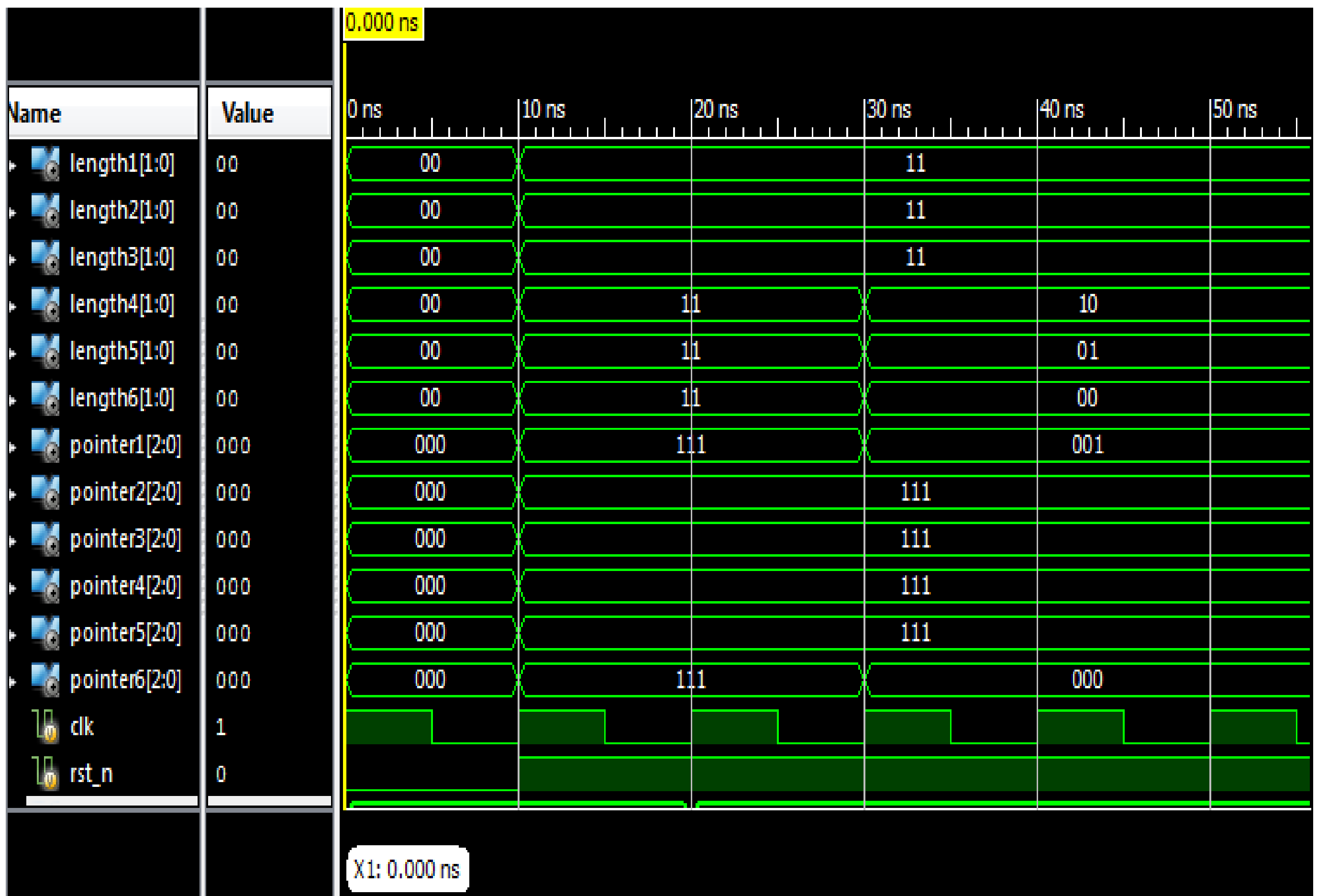

Figure 6.2 Simulated Design Outputs from Random Input Data

As seen from the simulation results, the outputs for 8 characters to be encoded are available within one clock cycle. The pointers and lengths are stored in their respective registers which may then be used using a simple logic circuit that picks up the necessary length/pointer pairs, append respective characters and form the desired code-words for further processing.

The Super unfolded architecture has been synthesized on various families of Virtes ® FPGA. When synthesized on Virtex-6 family of FPGA, the design can work on a maximum operating clock of 270 MHz, promising throughputs of 17.3 GigaBits / sec. Considerable slices are still available using the buffer specifications used, implying that larger buffers may be used, and increased characters may be encoded, enhancing the throughput many times. Detailed results are tabulated in Table 6.1.

Architectural Information	Virtex 4	Virtex 5	Virtex 6
No of Slices	335 / 6144	158 / 19200	158 / 93120
No of 4 input LUTs	632 / 12288	431 / 19200	397 / 46560
Max Operating Frequency	204 MHz	209 MHz	270 MHz
Throughput	**13 GBit/sec**	**13.4 Gbits/sec**	**17.3 Gbits/sec**

Table 6.1 : Experimental Results of Proposed Design

6.2 Comparisons

The proposed idea of carrying out future searches within the same clock cycle has already been introduced in [32]. However, the underlying design of modified column logic block, responsible for calculating lengths and pointers has been the main motivation for this proposed design. The block processes data in parallel and a considerable improvement from [32] has been obtained. Due to the nature of the design, our proposed design takes two stages of pipelining for fully parallel processing, whereas the design in [32] requires seven stages of pipelining for fully pipelined. Either speed or area has to be compromised in the previous design, and own proposed design overcomes this dis-advantage.

6.3 Summary

In this chapter, we have discussed in detail the experimental results of the proposed architecture. The design is being systolic array based is modular, fully parallel highly error-resisting. The nature of design makes it easier to extend the size of buffers so that larger number of future searches may be incorporated within the current clock cycle. The results obtained are far better than proposed designs in literature.

CHAPTER 7

CONCLUSION AND FUTURE RECOMMENDATIONS

7.1 Conclusion

Considerable improvements have been incorporated in design. By incorporating future searches in current clock cycle and elimination of redundant matches, high speed compression is accomplished and highest throughputs have been achieved.

The proposed architecture finds wide-spread applications in the communication industry. The design may be implemented on an FPGA and used in layered design, with a multiple data communication hubs, having various ports for communication applications depending on bandwidth allocation. Each layer may then be interfaced with this design through its corresponding port and all layers work in parallel to achieve high throughputs for the entire communication bandwidth. The optimized architecture saves space available on FPGA board for incorporating a further encryption scheme to help improve communication standard.

With the throughputs obtained from the design, the device may be easily connected on 100GigaBit interfaces, and incoming data streams may be distributed using MUXES at headers. The results also show that existing Bandwidths may be effectively and efficiently utilized, and even enable communication devices to communicate at virtually enhanced Bandwidths than allocated.

7.2 Future Recommendations

The novelty of design and its flexible nature promises a lot of work that can take on from the research conducted so far. The design has been developed using LZ 77 Algorithm. However, using the same scheme, the architecture can be modified for use with other data compression algorithms that may prove to be optimal for a given application. LZ 77 is, in fact, the first and the simplest of dictionary based schemes. Considerable improvements to

the LZ 1 or LZ 77 Algorithms have been documented and the same scheme may be used for implementation using this design.

The proposed design may also be extended for larger window sizes, including larger history and search buffers, and limitations to the size of window and history and search buffers may be identified.

For larger window sizes, the searches for matches may be done in parallel to optimize the design for higher throughputs. The design shall have to be broken up into multiple matching groups, for example, for window size W=40 with history buffer h=32 and search buffer s=8, we have 256 parallel matches to be processed. These matches may be pipelined into two groups of 128 matches each, four groups of 64 matches each or even eight groups of 32 matches each. The tradeoff would exist between area and speed, again depending on the nature of application for which it is being used.

Moreover, critical time path in column logic circuit may be reduced by pipelining the design, which may further enhance the compression speed and increase throughputs further. Moreover, unfolding design to higher orders may introduce latency in the architecture that may be again overcome through pipelining of the blocks required to work in parallel and independent.

These are few of the leads that are perceived to be carried on from the research concluded in this dissertation and may be exploited in future work on the subject.

References

[1] Fundamental Data Compression by Ida Mengyi Pu, Butterworth_Heinemann Publishers, 2006.

[2] B. Jung, W. P. Burleson, "Real-time VLSI compression for High-Speed Wireless Local Area Networks, Proceedings of Data Compression Conference, 1995.

[3] B. Jung, W. P. Burleson, "Efficient VLSI for Lempel_Ziv Compression in Wireless Data Compression Networks", Proceedings of International Symposium on Circuits and Systems, London June 1994.

[4] B. Jung, W. P. Burleson, "A VLSI Systolic Array Architectures for Lempel_Ziv Based Data Compression", Proceedings of IEEE Symposium on Circuits and Systems, 1994.

[5] Mark Milward, Jose Luis Nunez-Yanez and David Mulvaney, "Lossless Parallel Compression Systems", Electronic Systems and Control Division Research 2003, Department of Electronic and Electrical Engineering, Loughborough University, LE11 3TU, UK.

[6] Konstantinos Papadopoulos, Ioannis Papaefstathiou, "Titan-R: A Reconfigurable hardware implementation of a high-speed compressor", 16th International Symposium on Field-Programmable Custom Computing Machines, 2008 IEEE.

[7] Ioannis Papaefstathiou, "An IP Comp Processor for 10-GPBS Networks", IEEE test and Design of Computers", November-December 2004 .

[8] Suzanne Rigler, William Bishop, Andrew Kennings, "FPGA-Based Lossless Data Compression using Huffman and LZ77 Algorithms" IEEE 2007.

[9] R. B. Tremaine, P. A. Franaszek,J. T. Robinson,C. O. Schulz,T. B. Smith,M.,E. Wazlowski,P. M. Bland, "IBM Memory Expansion Technology", IBM J. RES. & DEV. Vol. 45 No. 2 March 2001.

[10] Kurtis T. Johnson and A. R. Hurson, "General Purpose Systolic Arrays", Pennsylvania State University, IEEE transactions , 1993.

[11] H.T. Kung. "Why Systolic Architectures?" Cotnpurer, Vol. 15. No. 1. Jan. 1982, pp. 37-46.

[12] http://en.wikipedia.org/wiki/Information_theory

[13] D. Huffman, "A method for the construction of minimum redundancy codes," Proc. IRE, 1958, Vol 40,pp. 1098-1101, Sep 1952.

[14] C. E. Shannon, "A Mathematical Theory of Communication", The Bell System Technical Journal, Vol. 27, pp. 379–423, 623–656, July, October, 1948.

[15] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, vol. IT-23 No. 2, May 1977.

[16] Ziv, J., And Lempel, A. Compression Of Individual Sequences via Variable-Rate Coding, IEEE Transactions On Information Theory 24 (1978), 530–536.

[17] Christina Zeeh, "The Lempel Ziv Algorithm", Seminar "Famous Algorithms" January 16, 2003.

[18] J.-J. Fan and K.-Y. Su, "An efficient algorithm for matching multiple patterns," IEEE Trans. Knowledge Data Eng., vol. 5, pp. 339–351, Apr. 1993.

[19] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Commun. ACM, vol. 20, no. 10, pp. 762–772, Oct. 1977.

[20] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," SIAM J. Computing, vol. 6, no. 2, pp. 323–350, June 1977.

[21] Mohamed A. Abd El Ghany, Aly E. Salama and Ahmed H. Khalil, "Design and Implementation of FPGA- based Systolic Array for LZ Data Compression", IEEE 2007.

[22] M. Kato, "Chipmakers push Lempel–Ziv LSIs for motherboard use," in Nikkei Electron. Asia, Apr. 1994, pp. 80–83.

[23] J. Chang, H. J. Jih, and J.W. Liu, "A Lossless Data compression Processor", in Proc. Of 4th VLSI/CAD Workshop, pp 134-137, Nan-Tou, Taiwan, August, 1993.

[24] Ranganathan N., and Henriques S., "High Speed VLSI designs for LZ-based data compression", IEEE Transactions, Circuits and Systems II, Analog Digital Signal Processing, 1993, 40 (2), pp 96-106.

[25] Ming-Bo Lin , "A Lossless Data Compression and Decompression Algorithm and Its Hardware Architecture" published in IEEE Transactions on VLSI Systems in 2006.

[26] M.-B. Lin, "A Parallel VLSI architecture for the LZW data compression algorithm", in Proc. Int. Symp. VLSI Technol., Syst., Appl., 1997, pp 98-101.

[27] C.Y. Lee and R.Y. Yang, "High throughput Data Compressor Design using Content Addressable Memory", Proc. Pt. G., vol.142, Feb 1995.

[28] Konstantinos Papadopoulos, Ioannis Papaefstathiou, "Titan-R: A Reconfigurable hardware implementation of a high-speed compressor", 16th International Symposium on Field-Programmable Custom Computing Machines, 2008 IEEE

[29] Shih-Arn Hwang and Cheng-Wen Wu, “Unified VLSI Systolic Array for LZ Data Compression”, IEEE Transactions on Very Large Scale Integration, Vol. 9, No. 4, August 2001.

[30] Storer, J.A.: 'Image and text compression' (Kluwer Academic Publishers, Norwell, Massachusetts, 1992)

[31] J. A. Storer, Data Compression: Methods and Theory. Rockville, MD: Computer Science Press, 1988.

[32] Rizwana Mehboob, Shoab A. Khan, Zaheer Ahmed, “High Speed Lossless Data Compression Architecture”, 10th IEEE Multi-Topic Conference INMIC 2006, Islamabad.

[33] T. C. Bell, J. G. Cleary, and I. H. Witten. Text Compression. Prentice Hall, Englewood Cliffs, NJ, 1990.

[34] Bayoumi, Magdy. Ling, Nam. Specification and Verification of Systolic Arrays. World Scientific Publishing Co. Pte. Ltd. Singapore. 1999.

[35] B.W.Y. Wei R. Tarver, J.-S. Kim, and K. Ng, “A Single Chip Lempel-Ziv Data Compressor”, in Proc. IEEE Int. Symposium. On Circuits and Systems (ISCAS), Chicago, May 1993.

[36] Brown, Andrew. VLSI Circuits and Systems in Silicon. McGraw-Hill Book Company. London. 1991.