Implementation of high-speed compressor using field programmable gate array

Konstantinos Papadopoulos

Microprocessor & Hardware Laboratory

Implementation of high-speed compressor using field programmable gate array (FPGA)

Konstantinos Papadopoulos

Supervisor: Assistant Professor Ioannis Papaefstathiou

Committee:

- Assistant Professor Ioannis Papaefstathiou
- Associate Professor Dionyssios Pneymatikatos
- Professor Apostolos Dollas

Technical University of Crete, Greece

January 2008

Abstract

Although current networks protocols can potentially support many data types (text, graphics, audio, and video), multimedia computing's future is tied to network bandwidth and quality-of-service issues. For the foreseeable future, network connections will range from low bandwidth (infrared, cellular modems, and Integrated Services Digital Network, or ISDN) to high bandwidth (such as gigabit networks). Advances in data compression will alleviate bandwidth problems in low-bandwidth networks by packing more data into fewer packets, effectively increasing bandwidth. Moreover, because network traffic grows ever busier, creating an insatiable demand for raw bandwidth, compression techniques are also important for high-bandwidth networks.

Nowadays, using contemporary low-cost reprogrammable field-programmable gate array (FPGA) technology enables us to implement compressors/decompressors which are capable of switching packets at speeds up to 10 Gb/sec. The processing in these switches comprises mainly of table lookups. The actual implementation of the majority of data compression algorithms consists mainly of table lookups, as well. Thus, the implementation complexity of a network compressor/decompressor is very similar to that of a network switch/router. As a result, it is claimed that devices that can compress network data at speeds up to a few Gb/sec can probably be implemented. These chips can be connected to the routers/switches and considerably reduce the bandwidth consumed. They can be applied whenever the bandwidth needed is more than the bandwidth provided, or whenever the network user is charged according to the network bandwidth used.

The purpose of this thesis is to implement the Titan-R, a single-chip IPcomp device for multi-gigabit networks, using field-programmable gate array (FPGA) technology. This chip compresses streams at speeds of a few Gbps, while introducing very low compression ratio. Moreover, the Titan-R operates transparently that is, after compression, the transmitted network packets have the same format as the unencrypted packets), making it easy to integrate in existing network infrastructures. Using a sophisticated compression algorithm, the Titan-R increased the effective bandwidth of tested IP networks from 48% to 250%. The Titan-R supports up to 32,000 different dictionaries, a feature that significantly increases the compression gain achieved when this device is connected to real networks. In terms of bandwidth supported, the Titan-R architecture is more efficient than existing approaches (see the "Related work" chapter) because it's the only one that uses a deep pipeline (256 stages) along with massive parallelism at each pipeline stage, and memory repetition for higher memory throughput.

Acknowledgements

First of all, I would like to thank my supervisor, Assistant Professor Papaefstathiou, for the excellent working atmosphere and the trust and freedom he granted me for my research. Furthermore, I am grateful to Associate Professor Pneymatikatos and Professor Dollas who agreed to evaluate this thesis. I am particularly thankful to Mr. Kimionis who obtained all I needed for my work.

I would also like to mention all my colleagues in Microprocessors and Hardware Laboratory (MHL) who have greatly helped me to accomplish this work. I am eternally grateful to PhD student Dimitrios Meintanis for his great help in learning the tools needed in Linux environment and Georgios Mplemenos for his help during the working period of this work.

I would like to dedicate this work to my mother, who left us too early, and my family for their ever-lasting support of my work and ideas and for getting me to the stage where I could attempt it. Finally, I would also like to express my gratitude to Magda for putting up with me.

Contents

List of Figures

List of Tables

1

Introduction

This initial chapter provides the motivation for conducting research in high performance data compression, summarizes the scientific contribution of the work, and describes the structure of this thesis.

1.1 Motivation

Network data compression has been used since the first congestion problems arose and high bandwidth applications were developed. In particular, the shorter representations of some data patterns are already an integral part of digital communications; everything, from the telephone network to the modems used on the PCs, already uses compression to achieve the speeds the customers are accustomed to.

The majority of the existing general network compression schemes have some common characteristics. They can compress traffic at speeds of up to 25 Mb/sec using a combination of hardware and software, the data after compression and decompression is exactly the same as the original one (lossless compression), they use a variation of the same basic algorithm and are performed either by a specific piece of software or by a very simple hardware device. The majority of these schemes are also applied to data departing to/arriving from the network gateway, and to either circuit networks or networks that carry packets of a few hundred bytes length. They are also very effective when applied to text or binary data, but cannot compress real-time data like video and audio, since these are always compressed at the source by the application itself. Even though both the current state-of-the-art and the next generation networks will carry all kinds of data, the text and binary files will always comprise a large part of the overall traffic.

Nowadays, contemporary low-cost reprogrammable field-programmable gate array (FPGA) technology enables us to implement compressors/decompressors which are capable of transmitting packets at speeds up to 10 Gb/sec. The processing in these switches comprises mainly of table lookups. The actual implementation of the majority of data compression algorithms consists mainly of table lookups, as well. Thus, the implementation complexity of a network compressor/decompressor is very similar to that of a network switch/router. As a result, it is claimed that devices that can compress network data at speeds up to a few Gb/sec can probably be implemented. These chips can be connected to the routers/switches and considerably

reduce the bandwidth consumed. They can be applied whenever the bandwidth needed is more than the bandwidth provided, or whenever the network user is charged according to the network bandwidth used. The motivation of this work was based on the fact that compression – as mentioned earlier – has been proved very effective when applied to real network data and numerous devices that compress the data sent over low-bandwidth links have been designed and are widely used. Nowadays, congestion problems in high speed networks have started to arise and they will probably be enhanced in the future. Since compression is a way of reducing the bandwidth used, it would ease the stress on these congested networks. But in order for this compression to be effective, a mechanism, that would decide the right number of connections that a network using compression can carry, should also be deployed. This mechanism should be aware of the fact that the network uses a certain compression scheme, so as to increase the number of connections admitted accordingly.

1.2 Scientific Contribution

Though current network can potentially support many different types of data (i.e. text, images, audio, video and graphics), the future of multimedia computing is tied to issues of network bandwidth and Quality of Service (QoS). The outlook is that potential users of networks will, for the foreseeable future, be diverse: from users with access to low-bandwidth connections (i.e. infrared, cellular modems, ISDN[(1)]), to users with access to very high-bandwidth connections (e.g. Gigabit networks). Advances in data compression will alleviate bandwidth problems in low-bandwidth networks by effectively increasing bandwidth. Moreover, based on technology's experience with the insatiable demand for raw computing speed (since applications seem to grow at an astounding rate), compression techniques would be important for high-bandwidth networks, as well.

In particular, in the case of high bandwidth networks, the application of Moore's law to the network integrated circuits means systems on a chip with the ability to process tens of millions of packets/second, at even the highest wire speed, would be possible. Along with Dense Wavelength Division Multiplexing (DWDM), which currently doubles the amount of fiber capacity available every year, this implies that all the kinds of network data will eventually go to packets, including backbone trunks, backbone switches, local voice switches, local data switches, business access lines, residential access lines, broadcast TV and even consumer electronics. So, even though "the network bandwidth is becoming cheaper and more readily available, we seem to find new and innovative ways to chew it up". As a result, compression desirable for these next generation high bandwidth networks, as well.

The compression algorithms generally operate by identifying repeating patterns in data and then replacing repetitive sequences with a token or reference to an earlier distance of the same sequence. From a simple perspective, compression is designed to reduce data to its most basic essence for efficient transmission and raise channel entropy. A major characteristic of the state-of-the-art networks is that they carry data from applications that have widely varying traffic characteristics –from real time

video clips to batch backup files. Thus, in order for a network to accommodate these applications, it must provide the appropriate Quality of Service (QoS) to each one of them. The most common approach today to delivering QoS is simply to provide additional bandwidth, or, in other words, to increase the useful bandwidth. Therefore, a sophisticated network resource allocation software which ensures that given QoS criteria are satisfied (even when the network works at or near capacity) will ease the strain on a modern network.

The thesis of this work is that a suitable compression framework can make certain types of packet networks more effective.

This framework comprises of:

- Hardware devices, software applications or a combination of the two to perform the actual compression. These devices/applications compress the traffic in a transparent way so that the intermediate routing devices can still be regular ones. This can be done in two ways: Either (a) with a compression/decompression device at each end of the link which would make the data appearing on the attached routing devices to be the original one or (b) the data would be compressed and then carried over ordinary packets which are routed/switched by regular devices all the way to the decompression device. These devices/applications are also capable of compressing the network data in an efficient way and at the speed of the network links.

- Interface modules for configuring these devices/applications according to the network specific characteristics. These are powerful, yet user friendly and fast for the network manager to configure the compression scheme, based on the rapidly changing features of the network data.

- A network admission control strategy for deciding how the network resources are shared between the network users. If this strategy is effective, the network can carry more data while maintaining the Quality of Service (QoS) promised to the users. If this control scheme does not take into account the reduction in the network data caused by the compression devices, the increase in the amount of traffic carried over the network will be minimal.

This thesis describes a compression framework argued to considerably increase the useful bandwidth and the number of connections admitted to a high-speed packet network. Its effectiveness is evaluated by implementation and by detailed simulation using data taken from real networks. In addition, this dissertation presents a low-complexity hardware compression device that can be used whenever the cost of the compression is a critical issue.

(1) ISDN: Integrated Services Digital Network

1.3 Structure of the Thesis

The remainder of this thesis is organized as follows:

- Chapter 2 presents the related work in lossless hardware-based data compression and, more especially, analyzes the X-MatchPRO compression/decompression architecture.
- Chapter 3 describes the main architecture of the system which leads us to high performance data compression.
- Chapter 4 points out its hardware implementation using low-cost reprogrammable field-programmable gate array (FPGA).
- Finally, chapter 5 provides results derived from the measurements, conclusions and future work on data compression.

2

Related work

This chapter presents the past work in high performance data compression. Especially, it presents the X-MatchPRO high-speed lossless(2) data compression algorithm and its hardware implementation, which enables data independent throughputs of 1.6 Gbit/s compression and decompression using contemporary low-cost reprogrammable field-programmable gate array technology. A full-duplex implementation is presented that allows a combined compression and decompression performance of 3.2 Gbit/s. The features of the compression algorithm and architecture that have enabled the throughputs are described in detail. X-MatchPRO is a fully synchronous design proven in silicon specially targeted to improve the performance of Gbit/s storage and communication applications.

(2) The term 'lossless' means that the original data can be exactly recreated after a decompression operation, and should not be confused with audio and video compression systems (such as JPEG and MPEG) which are lossy and hence only recreate an approximation of the original data.

2.1 Introduction

Lossless data compression, where the original data is reconstructed exactly after decompression, has been accepted as a tool that can bring important benefits to a computing system. The most obvious benefit of data compression is a reduction in the volume of data which must be stored. This is important where the storage media itself is costly (such as memory) or other parameters, such as power consumption, weight or physical volume, are critical to product feasibility. Using data compression reduces the total storage requirement, thus effecting a cost saving.

There are also two other positive effects that data compression brings. The first of these is a reduction in the bandwidth required to transmit a given amount of data. Less data must be transmitted in compressed form, and hence less band width is required. This can affect a cost saving in cabling operations, where a lower bandwidth link will be sufficient to meet demand. The second effect is that given a fixed bandwidth, the total time required to transmit compressed data is less than for uncompressed one. This can lead to a performance benefit, as the bandwidth of a link appears greater when transmitting compressed data and hence more data can be transmitted in a given amount of time.

Data compression applications have been increasing over the past years according to a combination of pressure for more bandwidth allied and to the need to improve storage

capacity. Lossless data compression has been successfully applied to storage systems (tapes, hard disk drives, solid state storage, file servers) and communication networks (LAN, WAN, wireless). One of the common factors for successful integration of lossless data compression in these applications is a high throughput so the compression/decompression processes do not slow the original system down. High performance lossless data compression has been researched as the means to achieve the high throughput target.
Data compression is not being used to its full advantage in systems that operate at bandwidths of over 1 Gbit/s due to performance limitations encountered in the data compression hardware. This chapter describes the X-MatchPRO method and architecture that uses a CAM-based dictionary where multiple symbols are processed per cycle to deliver the required performance so as to avoid becoming a bottleneck in a system operating at a gigabit per second bandwidth.
The remainder of this chapter is organized as follows:

- Sector 2.2 presents a review of the area of lossless hardware-based data compression.
- Sector 2.3 describes the characteristics of the X-MatchPRO algorithm.
- Sector 2.4 analyzes the system compression/decompression architecture.
- Finally, sector 2.5 points out the hardware implementation.

2.2 Background

A useful classification of lossless data compression systems identifies two main components: a model and a coder. The purpose of the model is to identify where the redundancy is located in the input data and signal repetitive data sequences to the coder. The coder uses the information obtained from the model to reduce the input data for shorter codewords and to produce a compressed output. Compression is obtained whenever the ratio of output to input bits is less than 1. Although some coding methods map better than others depending on the chosen model, many different combinations between model and coder are possible.
Modeling can be done mainly in two different ways: statistical or dictionary. Both methods have found their way to hardware and software implementations of lossless data compression systems.

Statistical Methods: Statistical methods show a cleaner separation between model and coder than dictionary methods. Statistical modeling is based on assigning values to events depending on their probability. The higher the value, the higher the probability. The accuracy with which this frequency distribution reflects reality determines the efficiency of the model. The best lossless compression figures reported in the literature correspond to software based statistical methods, like Prediction by Partial Matching (PPM) and Dynamic Markov Compression (DMC). These methods are based on variable order Markov modeling, where predictions are done based on the symbols that precede the current symbol. Statistical methods in hardware are restricted to simple higher order modeling using binary alphabets that limits speed, or simple multisymbol alphabets using zeroth-order models that limits compression. Binary alphabets limit speed because only a few bits (typically a single bit) are processed in each cycle while zeroth-order models limit compression because they

can only provide an inexact representation of the statistical properties of the data source. Coding is typically performed with methods like Huffman and arithmetic coding, the latter being preferred because its efficiency can be made arbitrarily close to the entropy or information content of the model by controlling its precision and therefore is optimal for any model. A few statistical data compressors have been reported in the literature. A zeroth-order model associated with an arithmetic coding is described by Boo et al. for coding of multilevel images. The probabilities in the model are stored in cumulative format using reference probabilities to simplify the update process. The arithmetic coding process has been simplified by truncating the multiplier. An implementation of a parallel binary arithmetic coder is done by Jiang using an IBM Q-coder as the building block. The Q-coder is a seventh-order binary Markov model associated with a corresponding binary arithmetic coder. The parallel implementation processes 4 bits in parallel and since there are only 16 possible input combinations. Parallel decoding is also possible. The same technique is used to obtain a parallel implementation of a multi-alphabet arithmetic coder associated with a byte-based zeroth-order model. The system processes 8 B at a time, but parallel decoding is in this case unfeasible because the number of possible input combinations is 256^8, hence, the complexity of the hardware is too high. The work presented by the same author in "Novel design of arithmetic coding for data compression" is the implementation of a byte-based zeroth-order model associated with a multi-alphabet arithmetic coder. Kuang et al. present another high-order binary model that describes a tenth-order Markov model with associated binary arithmetic coder. In this case, such the IBM Q-coder, the high-order binary Markov modeling uses fixed-order models and not variable-order ones such as PPM, because it is always possible to predict both symbols in a binary alphabet. The chip has been implemented in a 0.8 um and clocks at 25 MHz. The compression ratio is in the order of 0.5, while speed is data dependent but typically around 3 Mbit/s. Hsieh and Wei describe a byte-based zeroth-order model associated to a multi-alphabet arithmetic coder for video compression. A similar technique to "A VLSI architecture for arithmetic coding of multilevel images" is used to store the frequency model using some frequency counts as base and others as offsets from the base. This technique simplifies model adaptation. The chip described by Mukherjee et al. does not use arithmetic coding but three-based code, but others exist. The code is static and it does not adapt with changes in the incoming data source. Since it is not hardwired but mapped to a memory device, it can be changed to suit the application. A compression ratio of 0.5 processing 8-bit symbols results in each symbol to be processed in approximately two memory cycles. They report a compression performance of 95.2 Mb/s for compression and 60.6 Mb/s for decompression in a 2-um SCMOS technology with a clocking frequency of 83.3 MHz. An adaptive Huffman code implementation in hardware is presented in "Design and hardware architectures for dynamic Huffman coding". This design is based on content addressable memory (CAM) modules to speed up the tree adaptation process and achieves a throughput of almost 1 bit/cycle. The model is again a zeroth-order one but no details are available for the hardware implementation.

Dictionary methods: Dictionary methods try to replace a symbol or group of symbols by a dictionary location code. The modeling stage is given extra importance while coding is simplified. Some dictionary-based techniques use simple uniform binary codes to process the information supplied by the modeler. Both software and hardware based dictionary models are very popular, achieving good throughput and competitive compression. Utilities like Pkzip and ARJ in software, or hardware

algorithms like ALDC developed by IBM, by STAC/Hifn, illustrate this situation. These four examples are Lempel-Ziv-1 (LZ1) derivatives. The ALDC chip is implemented in a 0.8-um CMOS technology and clocks at 40 MHz to obtain a throughput of 320 Mb/s. The AHA implementation achieves 320 Mb/s at a 40 MHz operation and it is implemented in a 0.5-um CMOS technology. The STAC/Hifn device has been implemented in a 0.35-um CMOS technology. It clocks at 80 MHz with a throughput of 640 Mb/s. The Hifn device is also a full-duplex architecture meaning that it can compress and decompress simultaneously. Both of these chips use CAM memory to store the dictionary and enable parallel searching and adaptation. Surk presents a processing element with (PE)-based architecture for the LZ1 algorithm. Each PE compares the incoming input symbol with the symbol it stores in one cycle and shifts the symbol to its neighbor. The data input rate is constant. Post-layout simulation indicates a performance of 700 Mb/s in a 0.5-um CMOS technology. The basic symbol is 7-bits wide so the compressor is only suitable for the compression of ASCII coded model. Jung and Burleson describe another LZ1 implementation for the optimization of wireless local area networks. The architecture includes multichannel support being able to switch between different dictionaries depending on the communication channel being compressed. This improves compression since each channel has its own dictionary but there is an overhead associated with the multiplexing. A throughput of 50 Mb/s is reported based on 1.2-um CMOS technology using a clock frequency of 100 MHz. Nusinov and Pasco also present an LZ1 derivative for multichannel compression. The different dictionaries are stored in RAM memory externally and the appropriate one is uploaded in internal CAM. The chip clocks at 20 MHz and has a throughput of 80 Mb/s.
Lempel-Liv-2 (LZ2) algorithms have not become as widely used as LZ1 ones. The UNIX utility ‘compress’ uses LZ2 and the data compression Lempel-Ziv (DCLZ) family of compressors initially invented by Hewlett-Packard and currently being developed by AHA, also use LZ2 derivatives. The DCLZ family of devices clock at 40 MHz for a throughput of 160 Mb/s based on a 0.5-um CMOS technology. Bunton and Borriello present another LZ2 implementation that improves the DCLZ. This new algorithm uses a similar dictionary structure to DCLZ but it offers a more advanced dictionary maintenance mechanism where a tag is attached to each dictionary location to identify which node should be eliminated once the dictionary becomes full. The design has been implemented in a 2-um CMOS technology with a throughput of 160 Mb/s.

Other work that cannot be classified in the range of statistical or dictionary coding includes the genetic algorithms (GA) developed by the DCP Research Corp. in the DCP816 chip. This chip is implemented in a 1-um CMOS technology and has a throughput of 1.68 Mb/s clocking at 40 MHz. It supports multiple channels of compression/decompression and uses 512 kB of external RAM per channel. Sakanashi et al. present a device for printer image compression also based on a genetic algorithm that is able to select the best group of pixels to be used as context to predict the next input pixel depending on the characteristics of the image being compressed. The compressing method is lossless. It is associated to reconfigurable hardware such as a field programmable gate array (FPGA) plus a standard IBM QM-coder, a derivative from the Q-coder, to perform the compression itself. The X-MatchPRO family of devices belongs to the category of dictionary-based compressors but they are not LZ derivatives. X-MatchPRO originates from previous research and advances in FPGA technology. The flexibility provided by using this technology is of

great interest since the chip can be adapted to the requirements of a particular application easily. The objective is to use programmable hardware able to obtain good compression ratios and still maintain a high throughput so that the compression/decompression processes do not slow the original system down.

2.3 Algorithm Description

The X-MatchPRO algorithm uses a fixed-width dictionary of previously seen data and attempts to match or partially match the current data element with an entry in the dictionary. Each entry is 4 B (tuple) wide and several types of matches are possible where all or some of the bytes at different positions within the tuple match. Those bytes that do not match are transmitted as literals. This partial match concept gives the name to the procedure –the X referring to "don't care". At least 2 B have to match and when no valid match is generated a miss is codified adding a single bit to the four-byte tuple. The dictionary is maintained using a move to front (MTF) strategy whereby a new tuple is placed at the front of the dictionary while the rest move down one position. When the dictionary becomes full the tuple placed in the last position is discarded leaving space for a new one. X-MatchPRO reserves one location in the dictionary to code internal runs of full matches at location zero. Since the MTF strategy forces anything that repeats to be stored at location zero (top of dictionary), this run-length-internal (RLI) technique is used to efficiently code any 32-bit repeating pattern.

The coding function for a match is required to code several fields as follows.
A zero followed by:
If normal code:

1. Match location: It uses the binary code associated to the matching location.
2. Match type: That indicates which bytes of the incoming tuple have matched. This is codified using a static Huffman code based on the statistics obtained through extensive simulation.
3. Any extra characters that did not match transmitted in literal form.

If RLI code:

1. RLI location: The last address in the dictionary is reserved to code RLI events.
2. Run length: 8 bits are used to indicate how many 32-bit repeating patterns have been observed. The maximum run length that it is possible to process in a single code is therefore 225.

The coding function for a miss has two fields as follows:
A one followed by:

1. The 4 B in literal form.

A data tuple (4 B) is added to the front of the dictionary while the rest move one position down if a full match has not occurred. The MTF technique is only applied when dealing with full matches. In this case, the tuples from the first location until the location previous to the matching tuple move down one location, while the matching tuple is placed at the front of the dictionary.

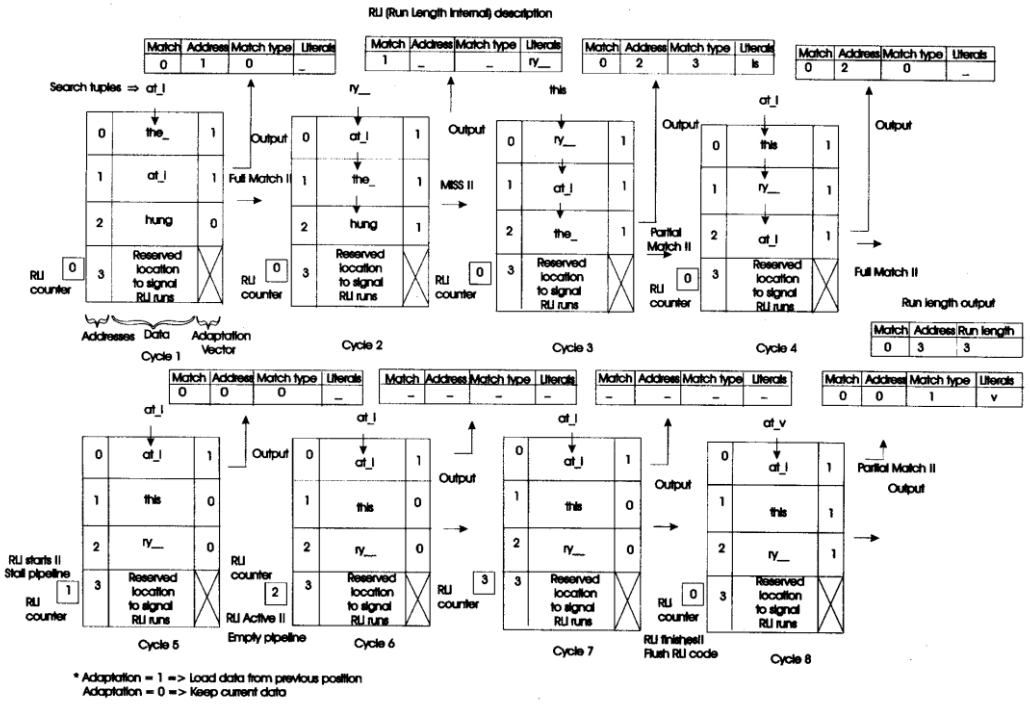

Figure 2.1. X-MatchPRO example

The algorithm is illustrated with an example in Figure 2.1. The example is based on a small dictionary of only four locations, one of which is reserved to code RLI events. Each dictionary location has a different address. The adaptation vector column defines how the dictionary adapts for the next cycle so that a 1 means load data from north neighboring location, while a 0 means keep current data. Each cycle in the figure corresponds to a different clock cycle. The search data in cycle 1 of Figure 2.1 generates a full match at location 1 and the corresponding output is generated together with a new adaptation vector that will shift the dictionary for cycle 2. The search data in cycle 2 cannot be found in the dictionary so a miss is generated with the four missing bytes being added to the output in literal form. The cycle 3 search generates a partial match where the two first bytes of the search tuple are found in location two. The match type 3 signals this matching condition and the two missing bytes are added to the output in literal form. Cycle 4 generates a new full match this time at location 2. An RLI coding event is inactive in cycles 5, 6 and 7. The RLI output is generated at cycle 8 when the run stops with a length of 3. The RLI counter only increments when the search data is present at location 0. The code generated at cycle 5 is removed from the output when the RLI counter exceeds 1 because cycle 5 would be coded as part of the run length. This output code would have been needed if the RLI counter had remained with a count of 1 indicating a single full match at location 0 and not a valid run length.

2.4 System Architecture

X-MatchPRO uses a simple coprocessor style interface to communicate with the rest of the system. Compression and decompression commands are issued through a common 16-bit control data port. A 3-bit address is used to access the internal

registers that store the commands plus information related to compressed and uncompressed block sizes for reading or writing. A total of six registers form the register bank. Three registers are used to control the compression channel and the other three for the decompression one. The first bit. In the address line indicates if the read/write operation accesses compression or decompression registers. The chip is designed to compress any block size ranging from 8 B to 32 kB. A decompression operation can be requested in the middle of a compression operation and vice versa.

2.4.1 Compression Architecture

The compression architecture is based around a block of CAM to realize the dictionary. This is necessary since the search operation must be done in parallel in all the entries in the dictionary to allow high- and data-independent throughput. The length of the CAM varies with three possible values of 16, 32, or 64 tuples trading complexity for compression. Dictionary size is variable so as to adapt algorithm complexity to the resources available in the selected FPGA. The number of tuples present in the dictionary has an important effect on compression. In principle, the larger the dictionary the higher the probability of having a match and improving compression. On the other hand, a bigger dictionary uses more bits to code its locations degrading compression when processing small data blocks that only use a fraction of the dictionary length available. The width of the CAM is fixed with 4 B/word and its columns can be configured as selectable shift-registers to implement the move to front adaptation policy.
Figure 2.2 shows the compression architecture. There are three major components in the compression architecture corresponding to compression model, coder and packer. It also shows the location of the pipeline registers used to reduce the clock period of the design. There are a total of five levels of registers from input to output and the design supports incremental transmission, which means that transmission of compressed data present in the output buffers can start before the whole data block is compressed. These two features help to maintain the latency of the design to a minimum.

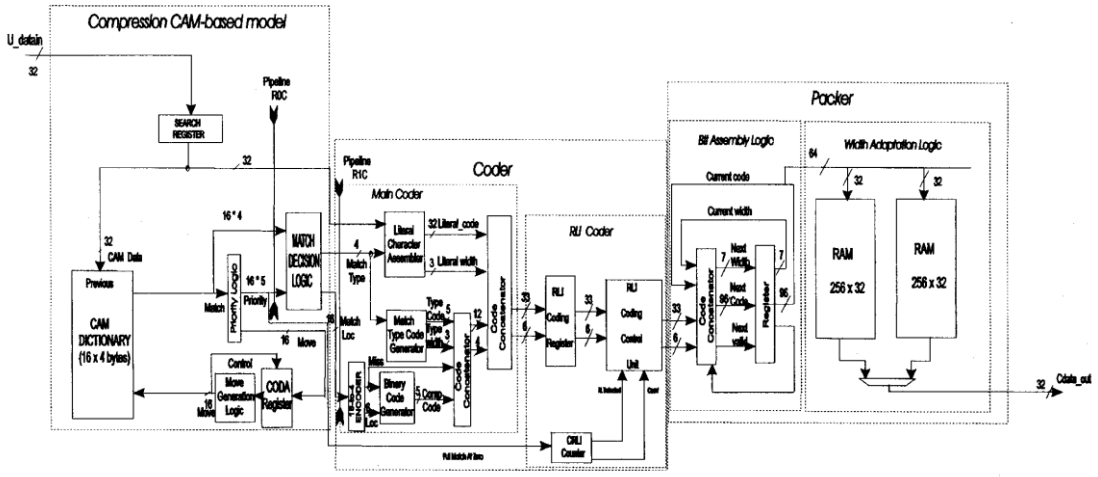

Figure 2.2. Compression architecture

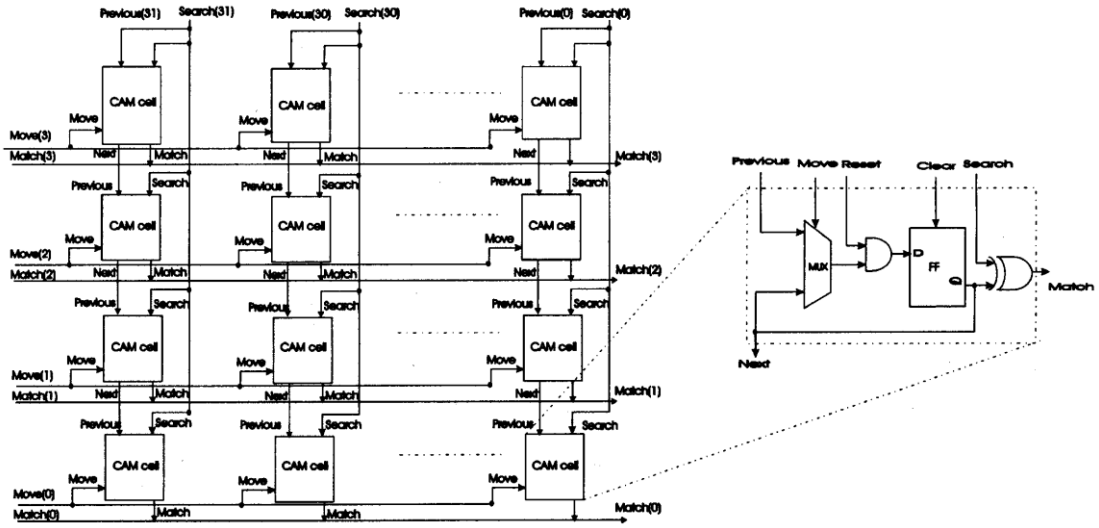

Figure 2.3. CAM-Based dictionary architecture

1. The Compression Model Comprises:
    a. Dictionary: CAM-based dictionary with 16, 32 or 64 tuples. The n-tuple dictionary is formed by a total of n x 32 CAM cells. Each cell stores one bit of a data tuple and it can maintain its current data, or load the data present in the dictionary using one XOR gate to do the comparison of each input bit plus ($\log_2$(dictionary width)) 2-input AND gates tree to obtain a single comparison bit per dictionary position. The delay of the search operation, although in principle is independent of dictionary length, in practice the high fanouts and long wires of large dictionaries degrade its speed considerably. An adaptation vector named ‘move’ in Figure 2.3 and whose length equals the dictionary length defines which cells keep its current data and which cells load data from its north-neighboring cell.
    b. Move generation logic: The adaptation vector ‘move’ is generated by the movement generation logic using the results of the search operation present in the ‘match’ vector of Figure 2.3. The movement generation logic function is to propagate up a match position so all the dictionary cells located over the match position and the match position itself load the data of their north neighboring cells, while all the dictionary cells located down the match position keep the current data. New data is always inserted at the top of the dictionary so when a data element is found in the dictionary it is promoted from its current position to the top of the dictionary in a single cycle. The propagation delay of the movement generation logic is O($\log_2$(dictionary length)) 2-input OR gates. Data flows toward the bottom of the dictionary as it grows older. The oldest data element is always located at the bottom of the dictionary and this is the one evicted from the list when room is required for a data element new to a full dictionary.
    c. Out of Date Adaptation (ODA) logic: ODA logic forces the dictionary to adapt with previous match information and breaks the critical path in compression improving speed. In principle, the adaptation vector ‘move’ must be generated using the results of the current search operation available in the ‘match’ vector before the next cycle can

start. This search and adaptation operation forms a critical feedback loop specially with large dictionaries because it depends on dictionary size with $O(1 + \log_2(\text{dictionary width} + \log_2(\text{dictionary length})))$ levels of logic. The search operation becomes critical since the fanout of the search register is directly proportional to dictionary length. It is not possible to add a pipeline register in the feedback loop without affecting algorithm functionality so to further increase the speed of the circuit the algorithm is modified introducing the ODA mechanism. ODA implies that adaptation at time t + 2 takes place using the patch results generated by the previously process data at time t and not the one at time t + 1. This technique breaks the fundamental feedback loop by adding a register between the search and adaptation circuitry. The danger is that dictionary efficiency could be lost if the ODA technique duplicates the same data in different positions in the dictionary. Prior to adding a register between the search and adaptation operations, the adaptation vector at time t provides information to reorder the dictionary at time t + 1 and makes sure that data words are unique in the dictionary. In ODA the adaptation vector at time t is not effective until time t + 2 so adaptation at time t + 1 could insert a data element at the top of the dictionary that already exists in some other dictionary location. After a few cycles the same data could be stored in multiple dictionary positions and dictionary efficiency would be lost degrading compression. The way to avoid this is by forcing the current adaptation vector to adapt not only the dictionary as before but also the next adaptation vector.

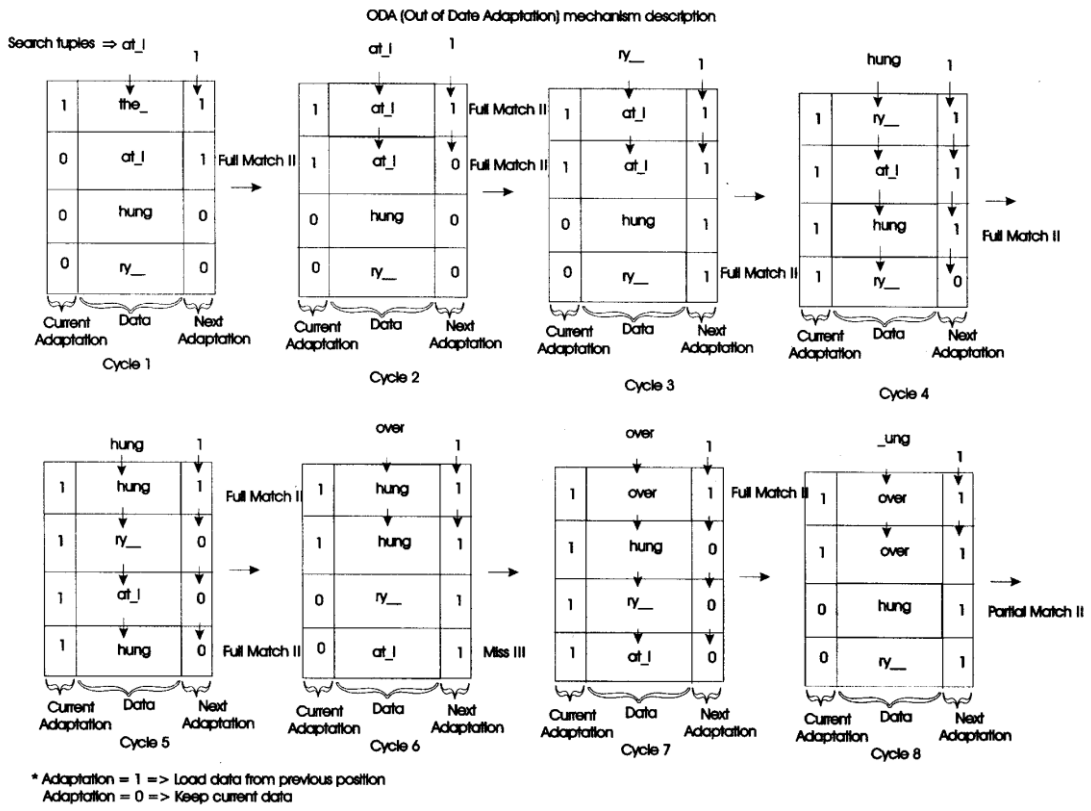

Figure 2.4. Out of date adaptation (ODA) example

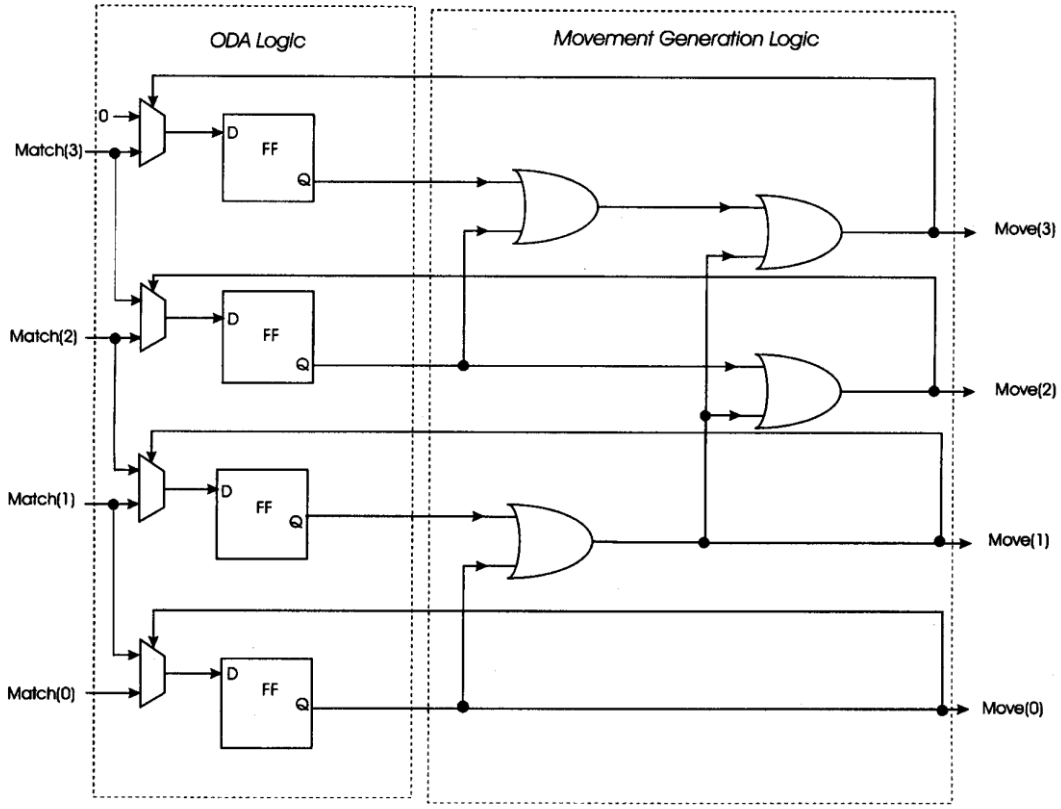

Figure 2.5. Adaptation logic architecture

Figure 2.4 illustrates this process using a small dictionary of only four positions in length and 4 B (tuple) in width. Every cycle of Figure 2.4 corresponds to a different cycle. The multiple full-match events in cycles 2 and 5 show how the search data could be found simultaneously at position 0 and at position higher than zero, but in this case the match at position 0 is selected as valid. The next adaptation vector depicted at the right of the dictionary depends exclusively on this match information. Figure 2.4 shows how ODA adapts the dictionary at time t + 2 using a modified adaptation vector originally generated at time t and how data duplication is restricted to position 0 maintaining dictionary efficiency. For example, the current adaptation vector depicted at the left of the dictionary for cycle 3 is generated shifting down the next adaptation vector of cycle 2, as indicated by the current adaptation vector of cycle 2. The current adaptation vector at cycle 3 adapts the dictionary of cycle 4. By using this simple technique, the effect of ODA in dictionary efficiency is negligible because in the worst case only one dictionary position contains repeated information and in the best case all the dictionary positions contain different data. The logic cost of ODA is very small since the basic ODA cell only contains a flip-flop and a multiplexor. Figure 2.5 shows the ODA logic plus the movement generation logic for a dictionary of four positions.

d. Priority logic: This logic assigns a different priority to each of the possible matches. A full match has the highest priority while partial matches are assigned priorities according to the number of matching bytes. The higher the number, the higher the priority.
e. Best match decision logic: Logic that selects one of the matches as the best for compression using the priority information.

2. The Coder Comprises of:
    f. Main coder: Main X-MatchPRO coder whose function is as follows: when a match is detected, it assigns a uniform binary code of size $\log_2$(dictionary size) to the match location preceded by a single bit set to 0, a static Huffman code to the match type and concatenates any necessary bytes that were not found part of a match in literal form. There are 11 possible different match type combinations of 2, 3 or 4 B matching in the tuple. The Huffman tree, obtained after extensive simulation, has only different code lengths 2, 3, 4 and 5 bits. The full match is the most probable match type and its Huffman code is only 2-bits long. Matches of three nonconsecutive bytes are the least probable and they are assigned 5-bit long Huffman codes. If, instead of a match, a miss is detected, the first single bit is set to 1 and the 4 B in literal form follow.
    g. RLI coder: The RLI coder detects the existence of multiple full matches at location zero, using a counter. If the counter exceeds the count of 1, then a RLI event becomes active, the pipeline is empty from the previous code and the output of the chip is frozen while the run length is taking place. A maximum of 255 full matches at location 0 can be coded in a single RLI codeword. The code corresponding to the last location in the dictionary is reserved to signal RLI events.

3. The Packer Comprises of:
    h. Bit assembly logic: Logic that assembles the variable-length codewords produced by the coder into 64-bit fixed length codes which are then output to the width adaptation logic.
    i. Width adaptation logic: This logic reads in 64-bit compressed words from the bit assemply logic and writes out 32-bit compressed words to the compressed output bus. It performs a buffering function smoothing the data flow out of the chip to the compressed port and it also transforms the data width from 64-bit to a more manageable 32-bit. It contains a total of 2 kB of fully synchronous dual-port RAM organized in two blocks of 256 x 32 bits to buffer compressed data before it is output to the compressed data out bus.

2.4.2 Decompression Architecture

Figure 2.6 shows the decompressor architecture. The decompressor channel is also formed by three major components: the decompression model, decoder and unpacker. The number of registers in Figure 2.6 from input to output is again five, so the latency of the compressor and decompressor channels is comparable. The design supports incremental reception so decompression of the compress block can start before the whole data block has been received.

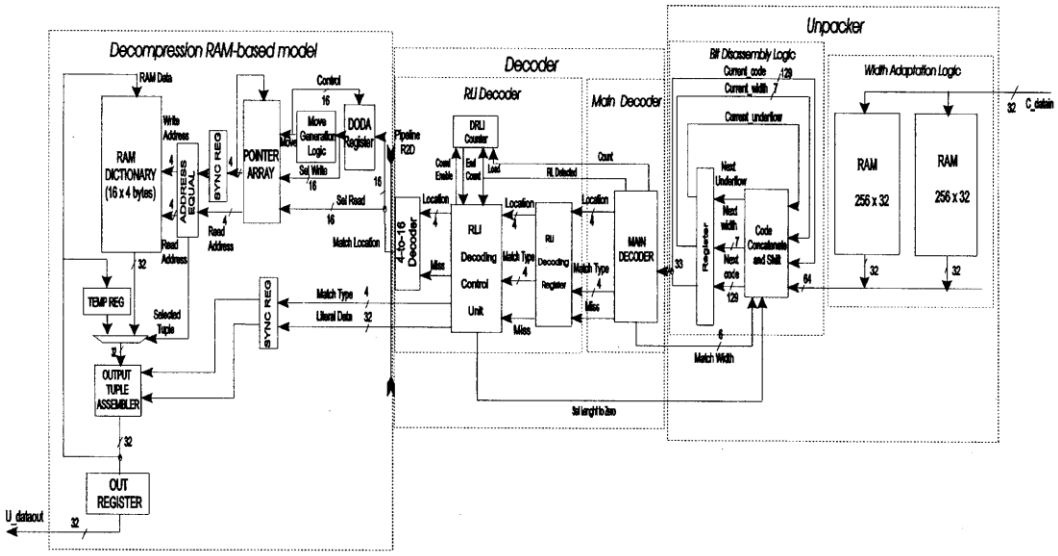

Figure 2.6. Decompression architecture

1. The Decompression Model Comprises of:
    j. Dictionary: Fully synchronous RAM-based dictionary that stores the history data during a decompression operation. The contents of the RAM dictionary during decompression must be the same as the contents of the CAM dictionary during compression in each cycle. Adaptation must take place in exactly the same way to enable correct decompression of the compressed block. The initialization of the compression CAM sets all words to zero. This means that a possible input word formed by zeros will generate multiple full matches in different locations. The algorithm simply selects the full match closest to the top. This operational mode, in effect, initializes the dictionary to a state where all the words with location address higher than zero are declared invalid without the need for extra logic. The reason is that location x – 1 is different from 0 because locations closer to the top have higher priority generating matches. The MTF adaptation mechanism shifts down the dictionary when full matches are not detected and, therefore, ensures that the last word from this initial state to be deleted from the dictionary is always the word located at location 0 at time 0. This operational mode in compression enables the decompression RAM dictionary to have only location 0 loaded with value 0 during the initialization phase because references to RAM locations higher than zero are not possible before their contents are updated. This technique avoids having a long overhead equal to dictionary size cycles to initialize each position in the RAM to a predefined value before each decompression operation. The read and write addresses are also monitored for possible collisions. If both addresses are the same, the algorithm needs to read the data that is going to be written in that common address. This data is not present in the memory yet, but it is present in the RAM data in bus. The RAM data is written in the memory normally but it is also latched temporarily in a register. Multiplexing logic selects the output coming from this register instead of the output coming from the memory when

the same address is being read and written. The read address is also modified to an unused address to make it different from the write one and avoid corrupting the RAM contents.

k. Pointer array: The pointer array logic performs an indirection function over the read and write addresses that accessed the RAM dictionary. It models the MTF maintenance policy of the CAM dictionary moving pointers instead of data. The pointer array enables mapping the CAM dictionary to RAM for decompression. Since the pointer array is much smaller than the CAM dictionary the savings in complexity allow having the full-duplex architecture in a single device. This is true because the basic pointer word width is 4, 5 or 6 bits depending on the length of the dictionary. On the other hand, the basic data word width is 32 bit. Each position in the pointer array is reset to a value the as its physical location in the array before each decompression operation.
l. Move generation logic: This logic generates the adaptation vector depending on the match type and match location. The adaptation vector moves the CAM dictionary in compression and the pointer array in decompression.
m. ODA logic: This component forces the pointer array to adapt with previous match information. The ODA logic in decompression is used to replicate the adaptation process in the compression dictionary. They have exactly the same functionality so both dictionaries are maintained in synchrony, although its use to improve the timing characteristics of the design is restricted to the compression channel.
n. Output tuple assembler: Module that assembles a decompressed tuple using dictionary information and any literal characters present in the code.

2. The Decoder Comprises of:
    o. Main decoder: The main decoder obtains a match type and a match location from the codeword supply by the bit unpacker. The first bit defines if a match or a miss follows. If a match is detected the next $\log_2$(dictionary size) following bits in the codeword define the match location. The Huffman code for the match type follows the match location code. If the match is partial the missing bytes follow the match type. If instead of a full or a partial match a miss is detected the next 32 bits following the first bit correspond to the four missing bytes.
    p. RLI decoder: RLI decoder that when the match location in the code word corresponds to the last position of the dictionary output match location 0 and match type 0 as many times as the number of the repetitions indicated in the next 8 bits that defined the run length. A counter is loaded with the run length and then it counts up until this value is reached.

3. The Unpacker Comprises of:
    q. Bit disassembly logic: This logic unpacks 64 bits of compressed data read from the internal buffers into variable-length codewords. To be able to shift out old data and concatenated new data, the codeword length must be supplied by the decoder logic. This feedback loop between the decoder logic and the unpacker one is illustrated in Figure

2.6 with the signal “match width” extending from the main decoder module to the code concatenate and shift module. The architecture of this module has been parallelized so concatenation of new data is done in parallel to the decoding operation and only the shifting of old data out must wait for the decoding operation to complete. This feedback loop remains, though, as the critical path of the design and limits the maximum clock frequency.

r. Width adaptation logic: The logic performs the equivalent but opposite function as its counterpart in the compression channel. It reads in 32-bit of compressed data from the input compressed bus and it writes out 64-bit of compressed data to the bit disassembly logic when it requires more data. It performs a buffering function smoothing the data flow in the chip from the compressed port. It contains 2 kB of fully-synchronous dual-port RAM organized in two blocks of 256 x 32 bits each as in the packer.

2.5 Hardware Implementation

The X-MatchPRO compressor/decompressor processor is a fully contained unit having a simple architecture and uncomplicated interface – Figure 2.7 shows the global architecture together with the PCI interface.

The X-MatchPRO design is a dictionary style compressor based around a dictionary implemented in the form of a content addressable memory (CAM). The length of the CAM varies with values ranging from 16 to 1024 tuples (4-byte locations) trading complexity for compression. Typically, the device complexity increases by a factor of 1.5 each time the dictionary doubles. Dictionary size is variable to be able to adapt algorithm complexity to the resources available in the selected FPGA. Each dictionary entry contains exactly 4 bytes. The dictionary adaptively stores the most recent phrases that have occurred in the data stream. Compression is achieved by replacing repeated phrases with references to the dictionary (these are codewords which are shorter than the phrase itself).

The coding section is active during compression. This generates the required codewords and forms successive codewords into fixed 32-bit width words for writing to external medium. The decoding section is responsible for the reverse process – data is read from the external medium and generates the required dictionary references to allow the decompressed data to be recreated.

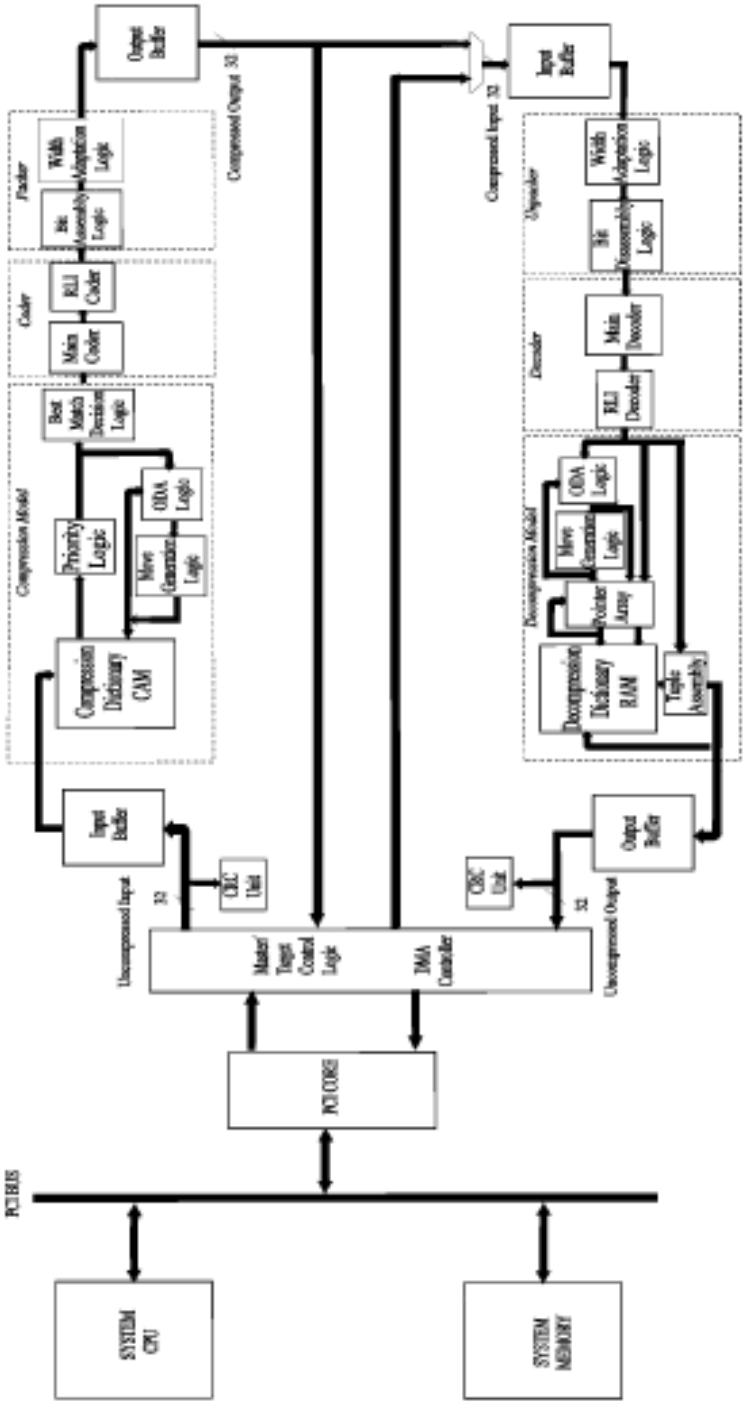

Figure 2.7. X-MatchPRO plus PCI interface architecture

2.5.1 Register bank description

A total of 10 registers form the register bank that controls the compression/decompression engines and coding/decoding buffers. These registers are accessed by using the address bus and the control bus and can be read or written. Figure 2.8 shows the format of these registers.

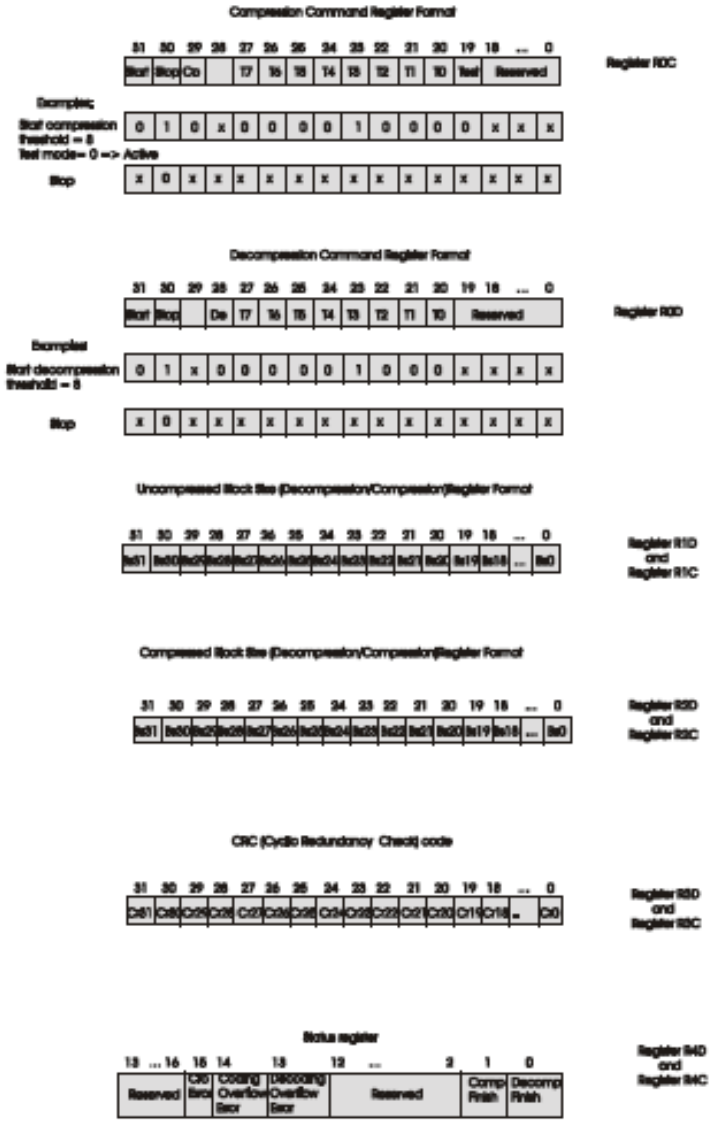

Figure 2.8. Register format

Address	Channel	Register	Function
1000	Decompression	R0D Command register	Activates or stops the decompression channel
1001	Decompression	R1D Uncompressed block size register	Sets the number of bytes of the uncompressed block before decompression
1010	Decompression	R2D Compressed block size register	Reserved
1011	Decompression	R3D Decompression CRC	CRC code is stored here after completion of a decompression operation
0001	Decompression	R4D Decompression Status	Status information of the decompression channel
1100	Compression	R0C Command register	Activates or stops the compression channel
1101	Compression	R1C Uncompressed block size register	Sets the number of bytes of the uncompressed block before compression
1110	Compression	R2C Compressed block size register	Sets the number of bytes of the compressed block before compression
1111	Compression	R3C Compression CRC	CRC code is stored here after completion of a compression operation
0000	Compression	R4C Compression Status	Status information of the compression channel

Table 2.1. Register access description

2.5.2 X-MatchPRO threshold value

The threshold value is input with the command and written in the command register. It defines a programmable latency. A small value means a low latency but it is more probable that underflows in the output buffers will take place. A bigger value introduces more latency but these conditions are not so frequent. After an underflow in the output buffers the threshold value also defines the distance between write and read addresses before more compressed or uncompressed data is output or requested respectively.

Underflow conditions are not error conditions but they will generate bubbles where valid data is not present in the compressed or uncompressed data out streams during compression or decompression respectively.

The threshold can have any value between 1 and 128. A threshold of 1 implies minimum latency => 1*64 bits of data are written in the buffer before the bus is requested during compression to output compressed data or 1*32 bits of data are written in the output buffers before the bus is requested during decompression.

A threshold of 128 implies maximum latency or blocked operational mode => 128 * 64 bits of data are written in the buffer before the bus is requested during compression to output compressed data or 128*32 bits of data are written in the output buffer before the bus is requested during decompression.

2.5.3 X-MatchPRO latency

In compression latency is defined as the number of cycles found between the moment the compression engine stops inputting data and the output buffers finish emptying the buffers (=> chip ready to start a new operation). The compression latency has two components one fix and one variable. The fixed component of 4 cycles is defined by the levels of registers located between the input search register and the output buffers (5 levels). The variable component is defined by how much data is in the buffers when the compression engine finishes its operation (flushing operation). The probability of having a long flushing operation is small when the threshold value setting is small. This variable component depends, however, in the input data. If the data expands, the latency will grow because more data will be left in the buffers to be output during the flushing operation.
In decompression, latency is also controlled by the threshold value. Latency can be defined as the number of cycles that elapse between the first tuple of compressed data enters the chip and the first tuple of uncompressed data leaves the chip. There are again two components. The levels of registers (5 levels) between the decoding buffers and the output register in the device introduced a fixed component of 4 cycles. The output buffer introduces the other component and it depends on the threshold value. A threshold value of 8 introduces a latency of 8 because 8 32-bit tuples must be written in the buffer before the number of 32-bit words exceeds the threshold value and the bus is requested to output uncompressed data.

2.5.4 X-MatchPRO operational modes

The device organizes the data block to be processed during compression and decompression operations in records of 512 bytes. This means that it will request the input data bus until one record has accessed the input buffers and then it will release the data bus and rearbitrate for new data if required until the whole block has accessed the input buffers. The compression and decompression engines are engaged shortly after the first input record has started accessing the bus and data will be available in the output buffers after a short latency. It is the responsibility of the system to service the requests originating in the output buffers to avoid having overflow errors in these buffers.

2.5.4.1 Compression mode

To start a compression operation the CPU must write two registers: The uncompressed block size register (UBSR) must be written first and the command register (CR) must be written second. The UBSR tells the compression engine when it must stop after processing all the bytes of data present in the block. The UBSR specifies the number of bytes present in the block and can be any value between 8 and 65536. The CR puts the device in compression mode and it also contains the threshold value to control the output buffer. It also sets the test bit that sets the device to self-checking test mode when 0 or to full-duplex mode when 1. The device requests the uncompressed data in bus after the command register has been set using the signal *bus request cu*. The system will grant the bus using *bus acknowledge cu* when data is ready for compression.
Data must be available in the uncompressed data in bus one cycle after the bus has been granted. If data is not ready for the device the *wait cu* signal in the

uncompressed data in bus can be asserted. The chip requests the compressed bus when the number of 64-bit words available in the output buffer is bigger than the threshold value using the *bus request cc* signal and waits for the acknowledgement *bus acknowledge cc*.
If data cannot be collected from the compressed data out bus the corresponding *wait cc* signal can be used to hold the outputting of data by the device. When the device produces compressed data in the compressed bus it asserts the *compressed data valid* signal active. The engine is known to be active because the *compressing* signal is active. The chip stops processing data when the value stored in UBSR is reached.
Then a *flushing c* signal is activated to indicate that any remaining compressed data in the output buffers is being flushed out. When the buffers are emptied of their contents the device asserts the signal *finished c* active for one cycle and the *interrupt request* signal. The system can read the compressed block size register (CBSR) at the end of a compression operation to obtain the resulting compressed block size in bytes. This value could be compared with the original uncompressed block size to evaluate the compression efficiency. The system can also read the status register to monitor that an abnormal termination did not take place. After this cycle the device is ready to start a new compression operation. Figure 2.9 corresponds to a typical compression operation.

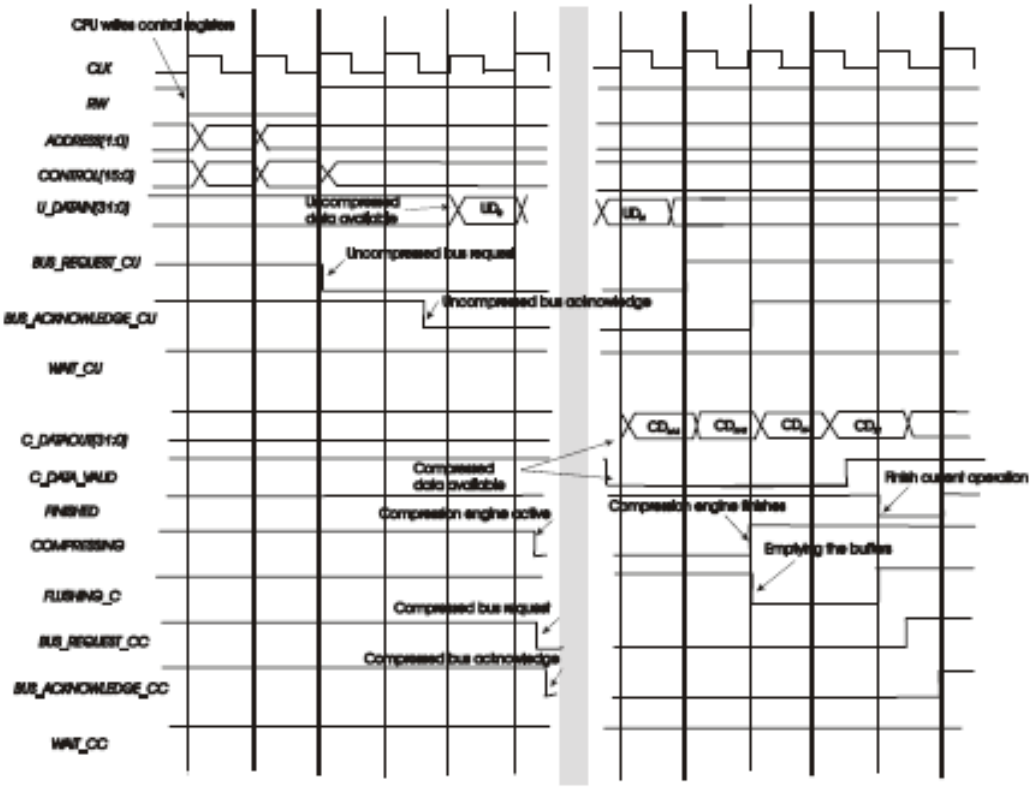

Figure 2.9. Compression operation

2.5.4.2 Decompression mode

To start a decompression operation the system must write 2 registers. The UBSR and the CR have the same function as in compression. The UBSR is used to indicate the device how much data must be decompressed before finishing the decompressed operation. Then, the system requests the compressed data in bus with the *bus request dc* signal and the bus is granted with the *bus acknowledge dc* signal. The *bus request dc* during decompression is equivalent to a compressed data request. Once the bus is granted the system is responsible to make available 32 bits of compressed data per cycle as long as the bus request signal is maintained active. The system can use the *wait dc* signal to insert wait cycles in the bus. The engine writes uncompressed data in the output buffers. Once the amount of data is larger than the threshold value the device asserts the bus *request du* signal requesting the uncompressed data out bus. The bus is granted with the *bus acknowledge du* signal. . When the device produces uncompressed data in the uncompressed data out bus it asserts the *uncompressed data valid* signal active. The engine is known to be active because the *decompressing* signal is active. When the output buffers are emptied of their contents the device asserts the signal *finished d* active for one cycle and the *interrupt request* signal. . The system can read the status register to monitor that an abnormal termination did not take place. After this cycle the device is ready to start a new decompression operation. Figure 2.10 shows a typical decompression cycle.

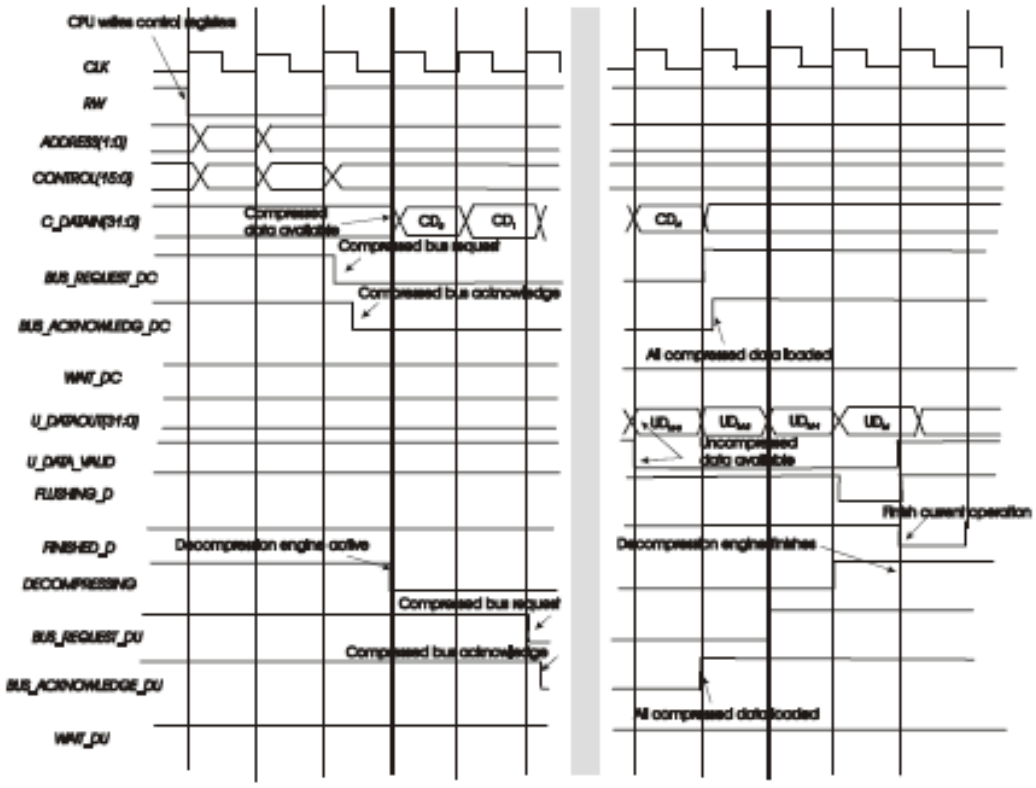

Figure 2.10. Decompression operation

2.5.5 X-MatchPRO Error conditions

2.5.5.1 Output Buffer Coding Overflow and Output Buffer Decoding Overflow

Overflow errors should never be encountered under normal operation conditions. To avoid overflow errors the output bus that holds compressed data during compression and uncompressed data during decompression should be granted if it is being requested when the inputting of one data record has finished and before the inputting of a new data record starts.

2.5.5.2 CRC Error

A CRC error should never be encountered under normal operation conditions. The CRC error signal is used during compression in test mode. Both channels are active and a CRC code is calculated using all the data input to the compression channel and output by the decompression channel. A CRC error indicates a hardware failure because either the compression or the decompression channels failed to successfully perform its operation and there has been a mismatch in the calculated CRC's by each channel.

3

System Architecture

This chapter describes the device which implements the LZ77 algorithm since it is the optimal for this end-to-end compression scheme. In the next sections the hardware architecture and the implementation of this device are outlined. But before moving to the actual implementation, it is essential to point out more details about the specific algorithm.

3.1 Compression Algorithm

There are mainly four different compression classes. Each one is suitable for particular applications. The problem with the network streams is that they consist of different kinds of data and so a flexible algorithm should be used.

The basic idea behind a substitutional compressor is to replace an occurrence of a particular phrase or group of bytes in a piece of data, with a reference to a previous occurrence of that phrase (dictionary-based compression algorithm). There are two main classes of schemes, named after Jakob Ziv and Abraham Lempel, who first proposed them in 1977 and 1978: the LZ77 and the LZ78.

It is widely supported that the LZ78 based algorithms produce better results by increasing the complexity of the matching circuit and thus making the compression procedure either slower or more expensive, in terms of hardware resources needed. However, this is not the case for the very irregular real network traces.

The main reasons the LZ77 based algorithms achieve a higher compression gain than the LZ78-based ones are as follows:

- The LZ78 ones "adapt slowly to their input data" and so in the case of the "non-stationary" network traffic, they cannot achieve the expected high compression gain.
- The major deficiency of the LZ77 is the limited size of the look-ahead buffer since it can go up to 32 bytes, in order to be effective. In the case where a 48-byte network packet is compressed at each run of the algorithm, the size of this buffer should, by nature, be much less than 48 bytes and so this deficiency does not affect the compression ratio.
- Another deficiency of the LZ77 is that the size of the search buffer is also limited. Since the larger the buffer, the better the compression, large buffers are preferred. However, this deficiency does not affect the compression ratio a network compression device can achieve since the traffic is "non-stationary" and thus a buffer even in the order of tens of kilobytes cannot achieve better results than a 4 KB one.

In addition to the better compression ratio achieved by the LZ77-based algorithms, there are some other reasons that make it more attractive for a network compression scheme:

- The decoding process is much faster, due to the intrinsic latency characteristics of the algorithm.
- It has a pre-defined longest delay in making the best possible matches and this delay is in the order of a few tens of character cycles.
- It is much easier to implement in hardware.
- There are no patents for the basic algorithm.

This thesis concentrates on the LZ77 Coding, which is implemented by this device.

3.1.1 LZ77 Coding

The main idea of LZ77 Coding is to use part of the previously seen input stream as a dictionary. Whenever an LZ-77 compressor processes a phrase that has already been seen, it outputs a pair of values corresponding to the position of the phrase in the previously-seen buffer of data and the length of the phrase. In particular, a compressor maintains a window to the input stream and shifts the input in that window from right to left, as strings and symbols are being compressed. This window is divided into two parts: the dictionary, which includes the symbols that have been input and compressed, and the lookahead buffer, containing the text it tries to find a match in the dictionary. In practical implementations the dictionary is some thousands of byte long, while the lookahead buffer is only tens of bytes long. In Figure 3.1 is the pseudo-code for the compression.

```
while( lookAheadBuffer not empty )
    {
    get a pointer ( position, match ) to the longest match in the window
        for the lookahead buffer;

    if( length > MINIMUM_MATCH_LENGTH )
        {
        output a ( position, length ) pair;
        shift the window length characters along;
        }
    else
        {
        output the first character in the lookahead buffer;
        shift the window 1 character along;
        }
    }
```

Figure 3.1. Pseudo-code for LZ77

Decompression is simple and fast: Whenever a (position, length) pair is encountered, go to that (position) in the window and copy (length) bytes to the output.

In Figure 3.2 a run of the compressor is shown and in Figure 3.3, a run of the decompressor.

Sliding-window-based schemes can be simplified by numbering the input text characters mod N, in effect creating a circular buffer. The sliding window approach automatically creates the LRU effect which must be done explicitly in LZ78 schemes. Variants of this method, like the gzip and the winzip software tools, apply additional compression to the output of the LZ77 compressor, such as dynamic Huffman coding and Arithmetic Coding, that result in a degree of improvement over the basic scheme. This increase is achieved whenever the data is rather random and the LZ77 compressor has little effect.

```
Output           History Lookahead

                           SIDVICIIISIDIDVI
S                        S IDVICIIISIDIDVI
I                       SI DVICIIISIDIDVI
D                      SID VICIIISIDIDVI
V                     SIDV ICIIISIDIDVI
I                    SIDVI CIIISIDIDVI
C                   SIDVIC IIISIDIDVI
I                  SIDVICI IISIDIDVI
I                 SIDVICII ISIDIDVI
I                SIDVICIII SIDIDVI  length: 3, offset: 9
(9, 3)       SIDVICIIISID IDVI     length: 2, offset: 2
(2, 2)     SIDVICIIISIDID VI    length: 2, offset: 11
(11, 2) SIDVICIIISIDIDVI
```

Figure 3.2. A typical run of the LZ77 compressor

```
History                 Input

                        S
S                       I
SI                      D
SID                     V
SIDV                    I
SIDVI                   C
SIDVIC                  I
SIDVICI                 I
SIDVICII                I
SIDVICIII               (9,3)   -> SID
SIDVICIIISID            (2,2)   -> ID
SIDVICIIISIDID          (11,2)  -> VI
SIDVICIIISIDIDVI
```

Figure 3.3. A typical run of the LZ77 decompressor

3.2 Implementation Details

The device was designed using the VHSIC(3) hardware description language (VHDL). The results of the initial high-level design were compared with the theoretical results so as to ensure the functionality of these devices is correct. Then, the VHDL code was synthesized, was placed and routed using the Xilinx ISE 9.1i and simulated using Modelsim 6.0a.

(3) VHSIC: Very-High-Speed Integrated Circuits

3.3 Format of Compressed Data

According to the end-to-end scheme, the network traffic is compressed at a source site, encapsulated over ordinary network packets and then decompressed at the destination site. In particular in the source node all the cells that have the same header are collected, all the headers removed and thus a long data-stream formed. Then and before the transmission if this datastream, the compression algorithm is applied to it and the compressed stream is encapsulated into ATM(4) cells. So, each compressed packet consists of a 48-byte compressed stream as the payload and the original header. Therefore, all the data coming on a certain VC(5)/VP(6) into the compression, leave the device on the same VC/VP, after being processed. In other words, the flows are not altered in any way except of the number of cells they consist of.

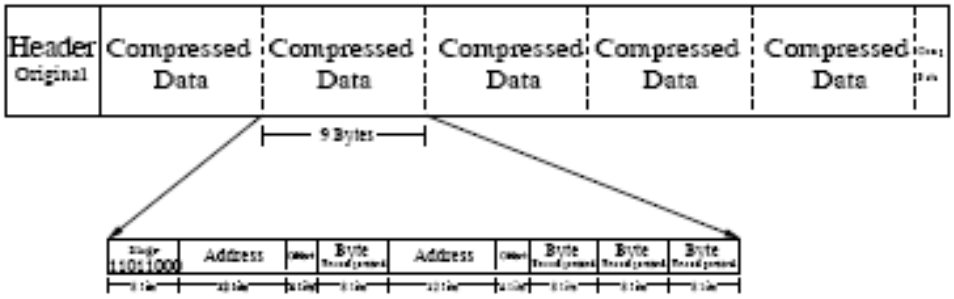

Figure 3.4. Compressed Cell Format

As it has already been described, the compressed streams comprise of 2-byte tokens and 1 byte uncompressed bytes, together with some bit flags for distinguishing the two. These flags should also be transmitted for the decompressor to make the same distinction. These bits cannot be efficiently sent together with their corresponding items. Instead eight outputs items (bytes or tokens) are collected together and, then, one byte consisting of the 8 flags is transmitted followed by the eight items. Figure 3.4 shows the format of a compressed cell. Recall that the last three bytes of the shown cell carry a portion of a 9-byte compressed quantity. This is not a problem since the decompressor can form data-streams comprising of more than one cell and then perform the actual decompression.

(4) ATM: Asynchronous Transfer Mode
(5) VC: Virtual Circuit/Virtual Channel
(6) VP: Virtual Path

3.4 System Interconnections

The System Chips are interconnected to each other and to the existing network terminals as shown in Figure 3.5. In this figure it is clear that both the compressed and the uncompressed data is carried by ATM cells.

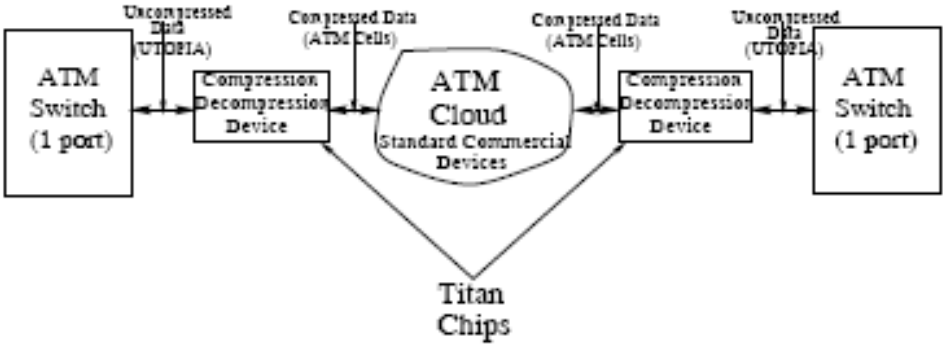

Figure 3.5. System Interconnections

3.5 Core Hardware Architecture

In Figure 3.6, the block diagram of the system chip is shown.

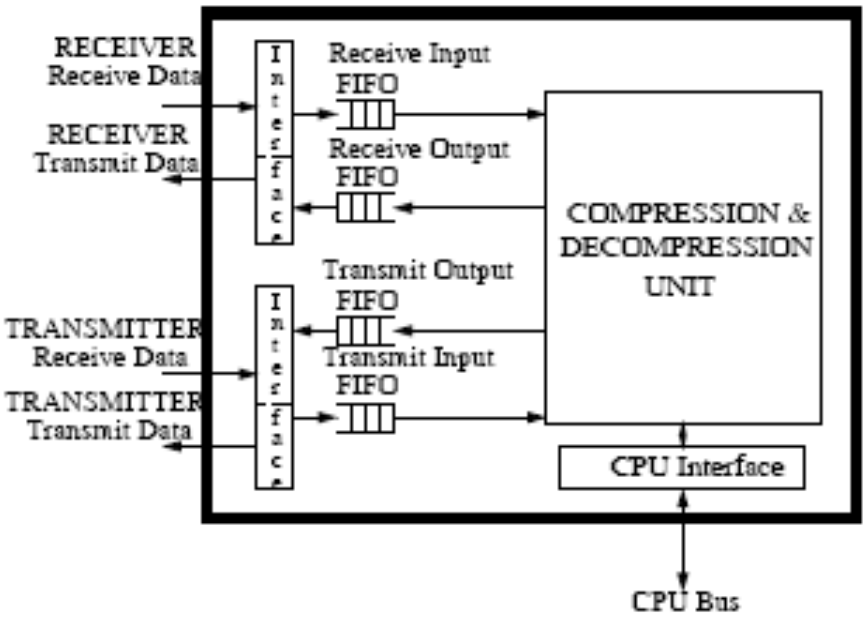

Figure 3.6. Block Diagram of the system

As in every dictionary based compression device, the core comprises of the dictionary and the comparison circuits around it. Therefore, the speed of the device depends heavily on the memory throughput and the comparisons' latency. In the architecture proposed in this chapter the speedup implementation techniques of pipelining, parallelism and repetition of information have all been used in order to accelerate this core. In particular, the main characteristics of the architecture are:

- 256-stage pipeline.
- 16 comparisons in parallel at each pipeline stage.
- Memory repetition (100% more memory used) for higher memory throughput.

This architecture was implemented using the VHDL and the Synthesiser previously mentioned.

Using these speedup techniques and after some optimisations of the device, the network data can be compressed at speeds up to 2.5 Gb/sec and the latency introduced is within the acceptable limits for network traffic (5 cell times). The latency can be further reduced if greater hardware resources are to be used, as it will be described in the next section.

3.5.1 Compression Unit

The compression unit implements the LZ77 algorithm which is applied only to the payloads of the ATM cells. Its block diagram is shown in Figure 3.7. In general, the compressibility table determines if a flow should be compressed or if it should bypass circuits ensure the main unit and the header memory, and the merge and bypass circuits ensure the cells will be formed and sent over the transmission link correctly.

Moving to the exact functionality of the device, at first the header bits described by a certain register are used as an index to the compressibility table so as to determine whether this particular cell should be compressed or not. If the cell should not be compressed it is sent through the bypass path to the merge circuit. If it is a compressible cell the compressibility table points to the dictionary to be used for compressing the cell. In each entry of the table the 15 last bytes of the payload of the last cell on this flow are also stored. These bytes together with the first byte of the new cell form a 16-byte look-ahead buffer which is sent to the compression unit. In the next clock cycle, the second byte of the cell will arrive. These two first bytes of the cell together with the 14 last bytes of the previous cell will form the new look-ahead buffer and so on. In this manner the 48 bytes of the payload are processed by the compression unit in 48 byte-clock cycles.

As stated above, the core circuit is organised in a 256-stage pipeline. In Figure 3.8 a pipeline stage is demonstrated. It consists of a memory bank of 31 bytes for each dictionary[7], a "crossbar" so as to route each 16-byte quantity to a specific comparator and 64 comparators that are used 4 times each at every clock cycle. The inputs of each stage are:

a) the 16-byte long look-ahead buffer,
b) the 15-bit address of the dictionary that should be used,
c) register LONMA which specifies what is the longest match up to the last point of the pipeline and what is the dictionary address of the first byte of this match and
d) the four PRENA1-PRENA4 registers which specify the 4 longest matches found in the last pipeline stage and the addresses of these matches.

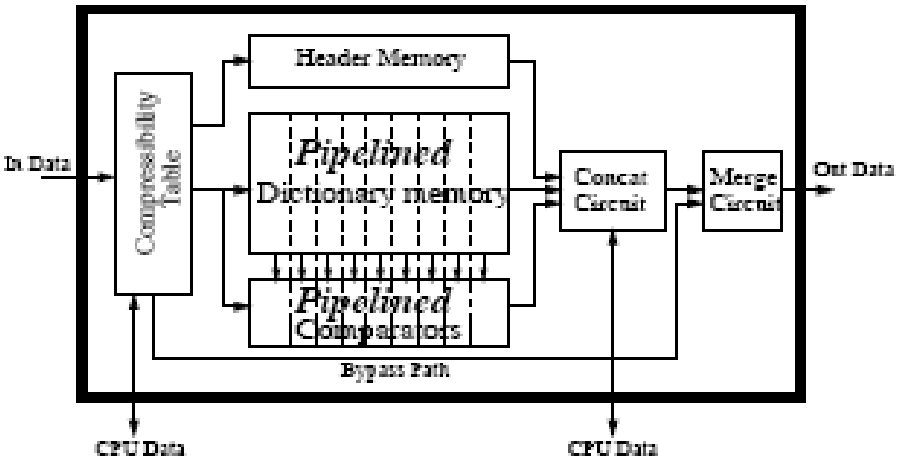

Figure 3.7. Compression Unit Block Diagram

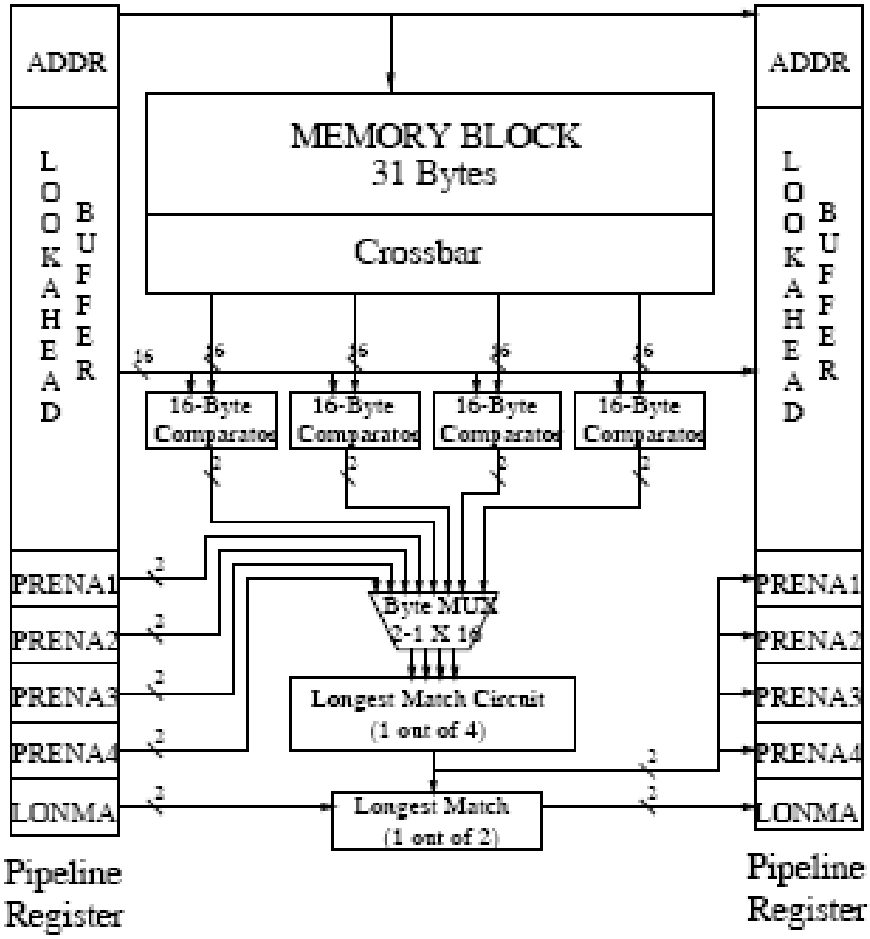

Figure 3.8. Block diagram of a Pipeline stage. Wire widths are in Bytes.

The outputs of each stage are:

a) the unchanged 16-byte look-ahead buffer,
b) the also uncharged 15-bit address fields,
c) the possibly altered LONMA register and
d) the new PRENA registers which specify the 4 longest matches found on this stage.

The reasoning behind the size of the memory is as follows: The algorithm implemented has a longest possible match of 16. Thus, taking 16 subsequent bytes, all their possible matches are included in these 16 bytes and the next 15 subsequent ones. So, it is guaranteed that all the matches of the first 16 bytes are included in the 31 bytes stored in the memory. This concept is illustrated in Figure 3.9. Note that the last 15 bytes should also be included in another memory bank together with the 16 bytes next to them in the incoming stream, for all the possible matches to be examined. Thus, this technique requires 93.75% of memory overhead.

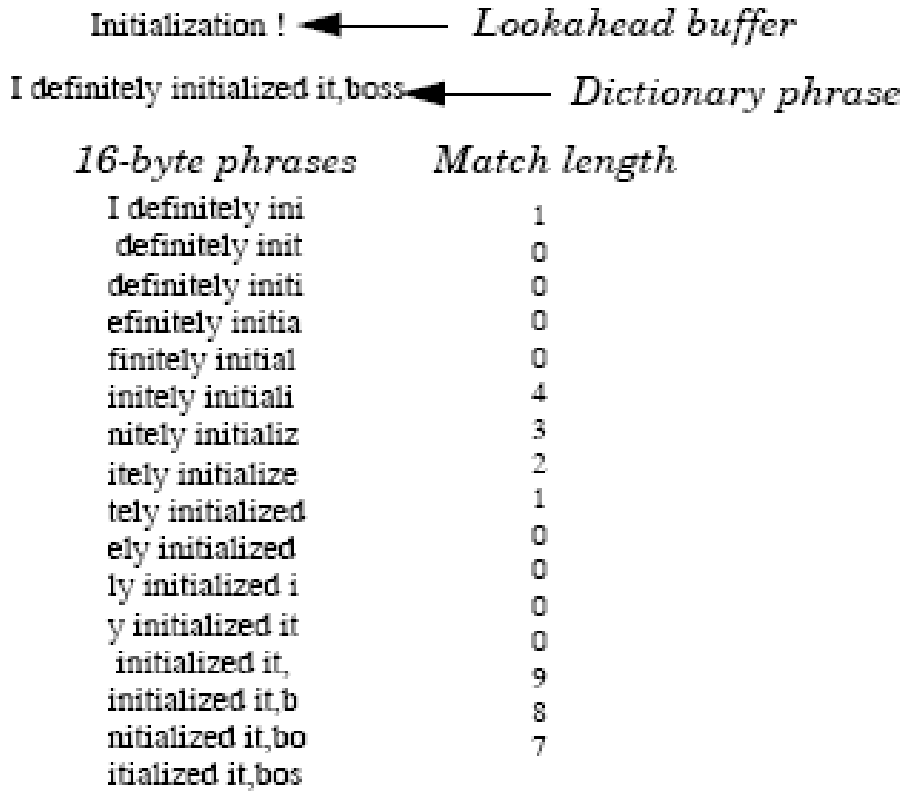

Figure 3.9. Overview of the compression task

Since the main objective has been to minimize the time for the comparisons of the 16 byte look-ahead buffer with every single byte in the dictionary, parallelism is also used. In every pipeline stage, there are 64 byte-comparators each used 4 times in each major cycle. Therefore, 256 comparisons are done in each major cycle. Since the 16, 16-byte long strings should be compared with the 16-byte long look-ahead buffer 16 * 16 = 256 comparisons are needed. As a result, in each clock cycle, all the possible matches of the look-ahead buffer with a particular 16-byte stream are identified.

The exact timing of a pipeline stage is shown in Figure 3.10. The memory is accessed and at the same time the register LONMA is compared with the four PRENA registers. The longest of these 5 matches is stored in register LONMA, together with the corresponding address in the dictionary. After the memory is read the first 4 16-byte comparisons are executed and their results stored in the corresponding registers. After the results are stored, the second set of comparisons starts and at the same time the 4 comparison results are compared with one another and the longest match is stored in register PRENA1. Similarly, all the four PRENA registers are loaded with the 4 longest matches produced by the 4 sets of comparisons.

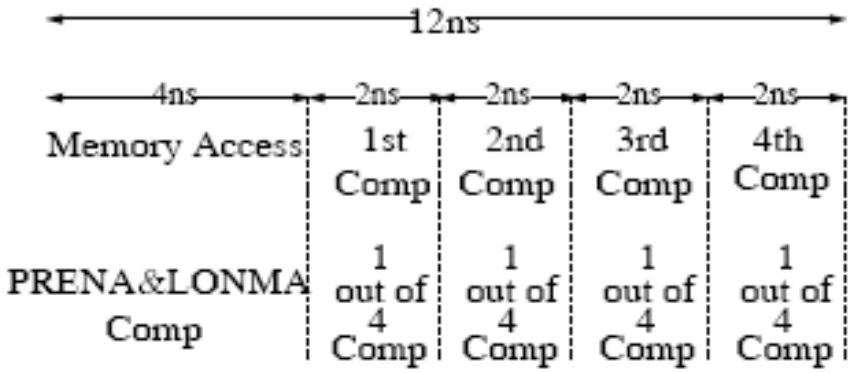

Figure 3.10. Timing diagram of a pipeline stage

As shown in the circuit diagram of Figure 3.8, in each one of the pipeline stages, there is a 31 byte per dictionary SRAM memory bank, 64 byte-comparators, 4 longest match circuits and a 15-byte pipeline register. Since there are 256 pipeline stages, the total hardware-cell count is 15872 8 bit-SRAM cells –all registers are also included-, 16K 8 bit-comparators and 1K longest match circuits, per dictionary.

By using all the above speedup factors, the compressor can process data at a constant speed of 622 Mb/sec introducing a latency of 256 clock cycles or 5 cell times. This is a significant improvement over the current network compressors since, as it is described in Chapter 2, the processing speed of the fastest such compressor is 100 Mb/sec and its latency is up to 2 cell times.

In order for this design to be used in an even faster network (e.g. a 1044 Mb/sec one), the only alternation needed is the following: Instead of having 64 comparators in each pipeline stage, 256 are needed, so as all the necessary comparisons can be done at the same time. By following the same calculations as in the last paragraphs, the latency of each pipeline stage will be 5 ns and, thus, a clock rate of 200 MHz can be used. As a result, the compressor would be able to process data at a rate up to 1.4 Gb/sec.

(7) Since the address is 15-bit wide up to 32K dictionaries can be supported. However the number of the actual dictionaries will depend on the cost requirements the device should satisfy.

3.5.2 Decompression Unit

The decompression unit is much simpler than the compression one, since there is no need for comparisons between the input data and the one stored in the dictionary. It just maintains the compressibility table and a 4 KB dictionary for each compressible flow. Its block diagram is shown in Figure 3.11. Using the compressibility table it first determines if a cell comprises of compressed or not. If the header corresponds to an uncompressible flow, the cell is sent over the bypass path. If it is a compressed cell, the compressibility table entry points to the dictionary that should be used for the decompression of it. Then, for each byte the decompressor determines, if it is a part of an (address, length) token or an uncompressed byte. In the latter case, it sends the byte to the merge circuit and stores it in the next free entry of the corresponding dictionary. In the former one, the memory item, at the corresponding address, is fetched and the first length bytes of it are written to the output buffer. The new string is written in the next free position of the dictionary.

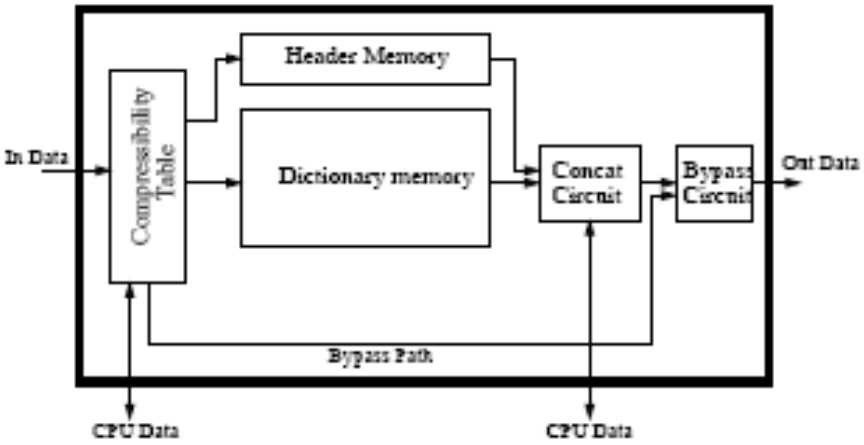

Figure 3.11. Decompression Unit Block Diagram

Since the decompression unit needs at most two memory accesses per input byte and assuming the same delay parameters as in the compression unit, its latency is 7 ns if a standard non-pipelined architecture is used. If the two accesses are performed into two different pipeline stages, in which case a dual port SRAM is also needed, the latency of the device will be 4 ns. Therefore, even when a non-pipelined architecture is used the unit can decompress data at speeds up to 1066 Mb/sec.

4

Hardware Implementation

This chapter describes the way implemented each component of the whole system. The following sectors present the implementation of every component in details and shows figures and tables about the architecture and the pinout used. The last ones provide the FSM(8) used for the successful operation of each pipeline stage and some optimizations made for improving the performance which is our main scope.

(8) FSM: Finite State Machine

4.1 Memory bank

As required, instead of LZ77 algorithm, a memory is used for saving dictionaries plus data duplicated for the reasons described in the section 3.5.1. This module is a ROM(9), generated by Xilinx CORE Generator, which contains a dictionary in every odd address and duplicated data in every even one. Its size is chosen to be 8 Kbytes (64 Kbits), 4 KB for the dictionary and 4 KB for the memory overhead. Previous research has showed that the optimal width for this memory is 16 Bytes (128 bits), so a memory bank, the size of which is 31 Bytes, will be resulted from reading two addresses. Subsequently, the depth of the memory is 512 words so as the address length is 9 bits.
Figure 4.1 shows the interface of this component while Table 4.1 describes the functionality of signals.

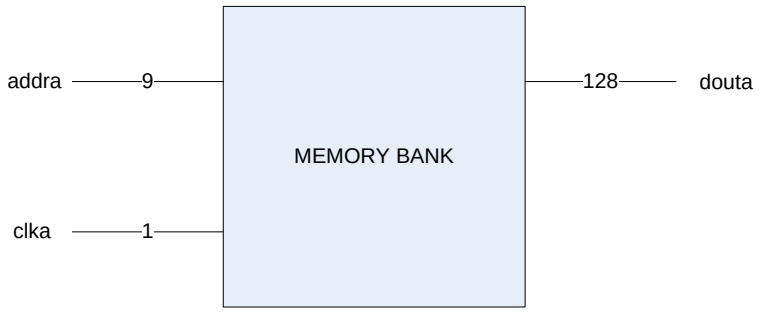

Figure 4.1. Memory Bank Interface

Signal	Width (bits)	Type	Description
clka	1	Input	Clock signal
addra	9	Input	Read address
douta	128	Output	Output Data

Table 4.1. Memory Bank pinout

The use of ROM proves that dictionary must not be written from the system; it is only initialized externally; it is initialized by a .COE(10) file which is loaded from the user to specify the values of the memory.

(9) ROM: Read Only Memory
(10) COE: COEfficient

4.2 Crossbar

"Crossbar" is not used with the strict sense of the term. It is a component that gets a 16-byte long input (output data from memory), creates the 31-byte long memory bank, saves it in a latch and uses it for further processing.
Crossbar Interface is showed in Figure 4.2 and more details about signal functionality are referred in Table 4.2.

Signal	Width (bits)	Type	Description
input	128	Input	Input data
wrbuffer	1	Input	Buffer write signal (if '1' write)
dsit	1	Input	Situation of written data (in buffer)
compNo	2	Input	No of comparison
output1	128	Output	1st output
output2	128	Output	2nd output
output3	128	Output	3rd output
output4	128	Output	4th output

Table 4.2. Crossbar pinout

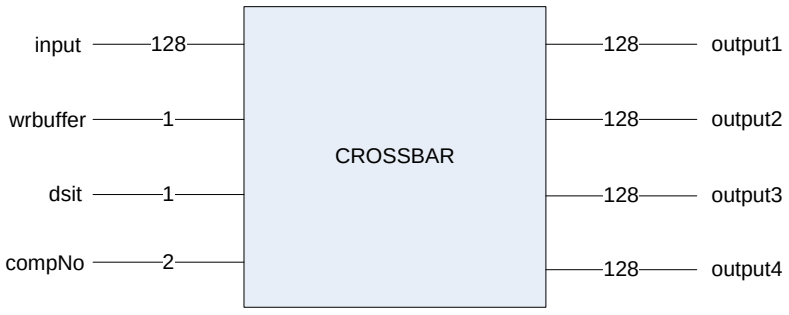

Figure 4.2. Crossbar Interface

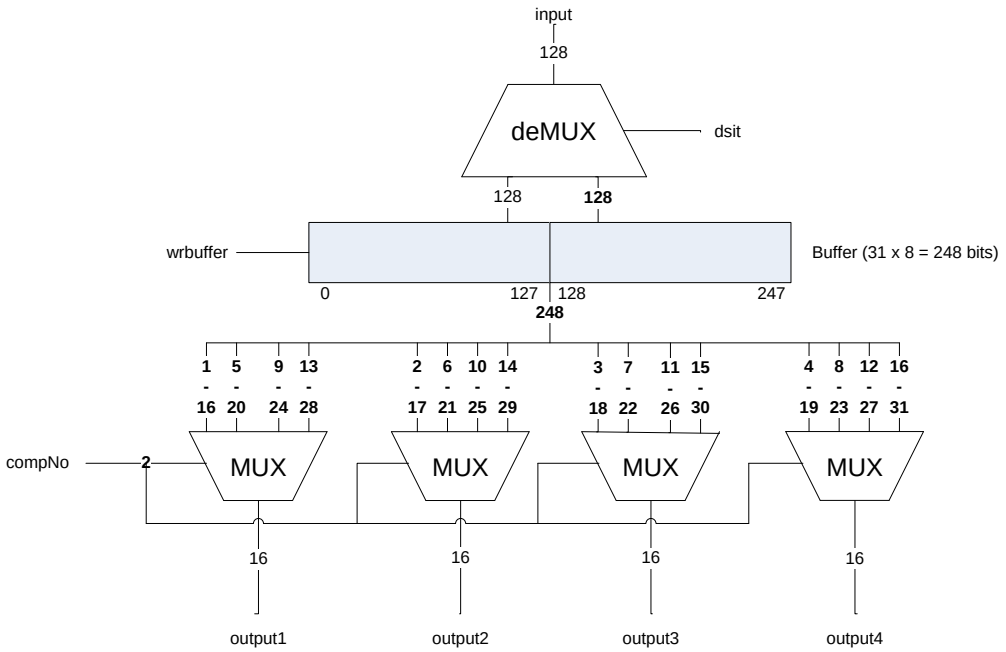

Figure 4.3. Crossbar Architecture

This component is implemented by using a 31-Byte (248-bits) long latch –as stated before- for saving the memory bank, a demultiplexer 1-2 (16 byte input – outputs), which decides where the input data are going to be saved in the latch and four multiplexers 4-1 (16 byte inputs – output) that decide which 16 bytes are going to be sent to the four outputs for further processing. The architecture of this module is showed in Figure 4.3.

As presented in figure above, according to the "dsit" signal, which controls the demultiplexer, the input data are saved either in bytes 1-16 (if '0') or in bytes 17-31 (if '1') of the latch. Furthermore, according to the "compNo" signal, multiplexers choose the desirable bytes to be sent to the outputs. All the possible combinations are presented in Table 4.3.

compNo	**outputs**	**Output1 (bytes)**	**Output2 (bytes)**	**Output3 (bytes)**	**Output4 (bytes)**
00		**1-16**	**2-17**	**3-18**	**4-19**
01		**5-20**	**6-21**	**7-22**	**8-23**
10		**9-24**	**10-25**	**11-26**	**12-27**
11		**13-28**	**14-29**	**15-30**	**16-31**

Table 4.3. Output combinations according the No of comparison

4.3 Pipeline Stage Comparator

Pipeline Stage Comparator is a circuit which compares two 16 byte (128 bit) long vectors, which are its inputs. The first one is the unchanged Look-ahead buffer and the second one is 16 bytes from the memory bank derived from the Crossbar using the method described in the previous section. Its output is the longest match (the largest number of common bytes in row) between these vectors concatenated with the address of the first byte of the second vector in memory bank.
The interface of this component is showed in Figure 4.4 and described in Table 4.4, which is presented above.

Signal	Width (bits)	Type	Description
input1	128	Input	1st input of comparator (Look-ahead buffer)
input2	128	Input	2nd input of comparator (output of Crossbar)
memoryaddress	9	Input	Memory address of the dictionary in BRAM
firstbyte	4	Input	Address of the first byte in memory bank
longestMatch	16	Output	Output of comparator (longest match between input1 & input2)

Table 4.4. Pipeline Stage Comparator pinout

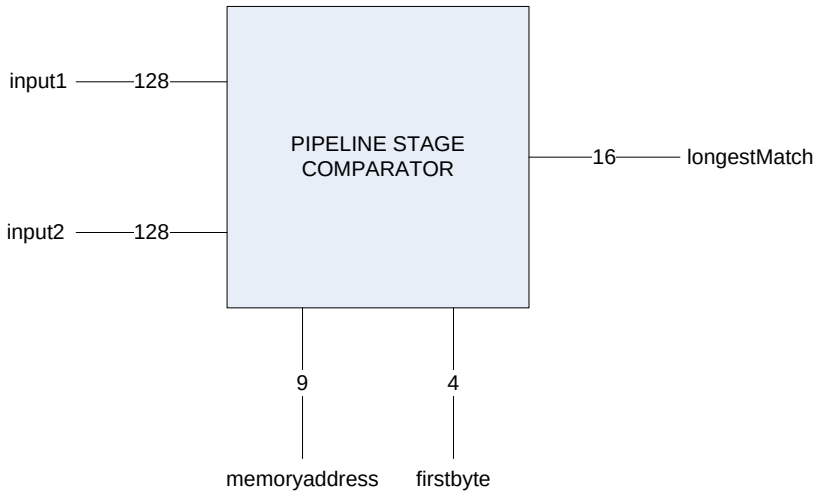

Figure 4.4. Pipeline Stage Comparator Interface

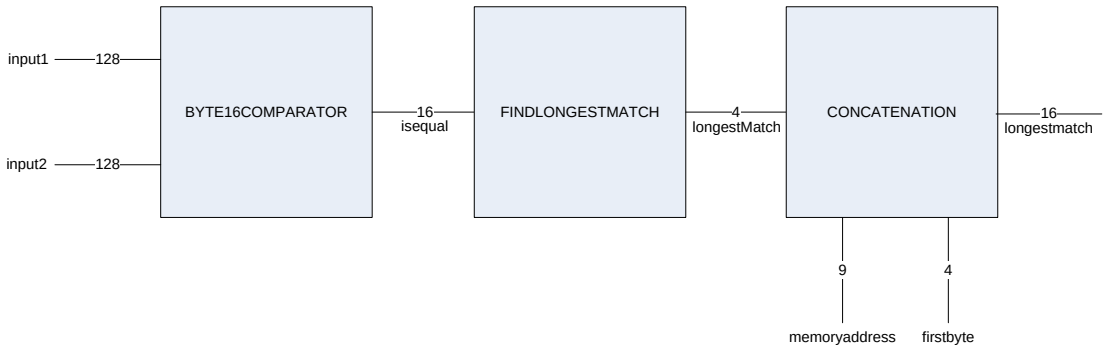

Figure 4.5. Pipeline Stage Comparator Architecture

The procedure, which is followed for leading us to the result, is divided in three stages. At the first stage, the two vectors are compared and a new vector (16 bit long) is created, which shows which bytes are the same in the leading vectors. At the second stage, the longest match is counted using the new 16 bit long vector and, finally, in the third stage, the longest match created is concatenated with the address of the first byte as described before.
Each one stage is processed by a specific subcircuit, so the three subcircuits used are a 16-Byte Comparator, a Find Longest Match Circuit and a Concatenation Circuit. The connection between these components, which leads us to the architecture of Pipeline Stage Comparator, is showed in Figure 4.5.
The implementation of these three components, presented in Figure 4.5, is described in details at the following subsectors.

4.3.1 16-Byte Comparator

The first part of a Pipeline Stage Comparator compares the two inputs (16 bytes long) and exports, as a result, a 16 bit long vector, in which the '1' shows that the specific bytes are equal and the '0' shows that are not. The interface of 16-Byte Comparator, as this structure is called, is showed in Figure 4.6 and more information is given in Table 4.5.

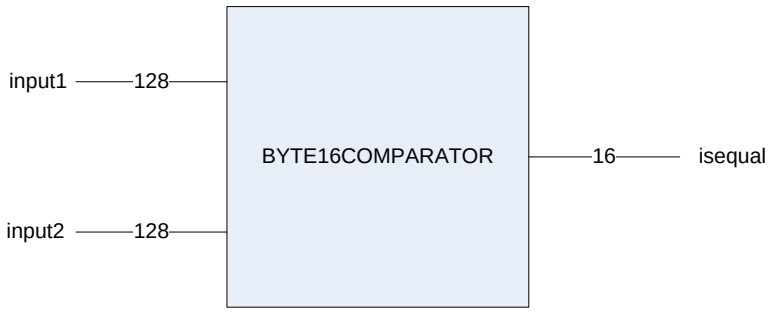

Figure 4.6. 16-Byte Comparator Interface

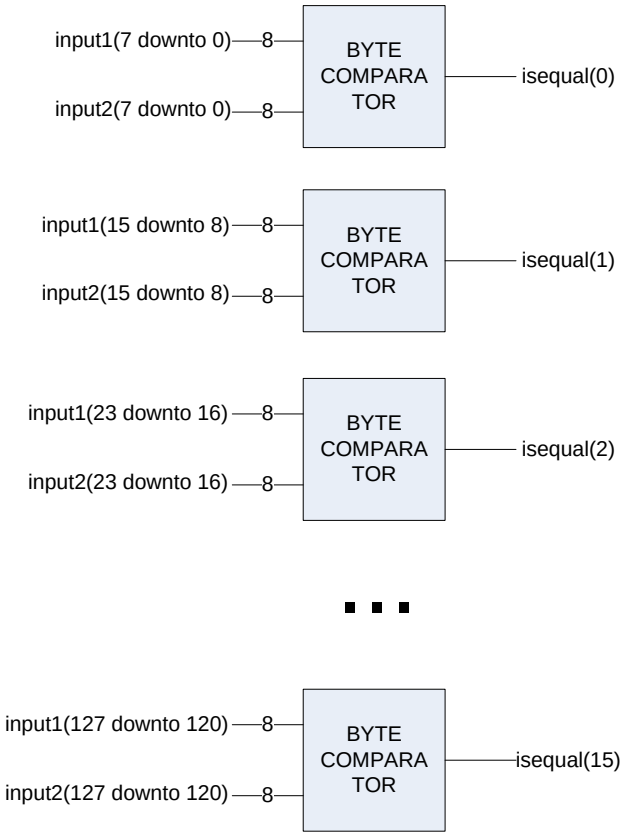

Figure 4.7. 16-Byte Comparator Architecture

Signal	Width (bits)	Type	Description
input1	128	Input	1st input of comparator
input2	128	Input	2nd input of comparator
isequal	16	Output	Output of comparator (shows which bytes between input1 & input2 are equal)

Table 4.5. 16-Byte Comparator pinout

The 16-Byte Comparator unit is implemented using 16 Byte comparators. The architecture used is presented in Figure 4.7.
The structure of Byte Comparator is described in details above.

4.3.1.1 Byte Comparator

Each Byte Comparator gets, as input, two vectors (one byte long each one), compares them and exports, as output, a bit, which shows if the two bytes (vectors) are equal (‘1’) or not (‘0’). Figure 4.8 shows the interface of this structure and Table 4.6 gives more information about the functionality of every signal.

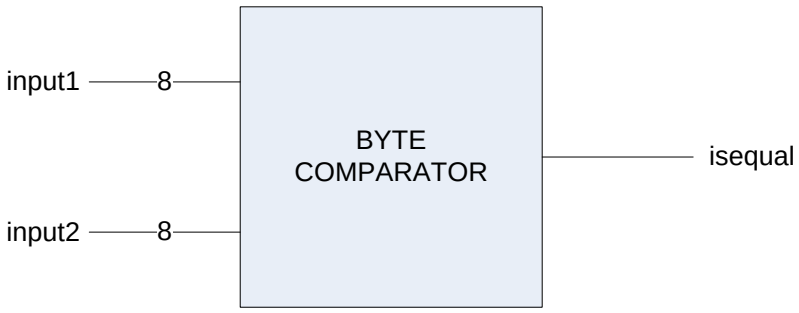

Figure 4.8. Byte Comparator Interface

Signal	Width (bits)	Type	Description
input1	8	Input	1st input of comparator
input2	8	Input	2nd input of comparator
Isequal	1	Output	Output of comparator (shows if bytes -input1 & input2- are equal)

Table 4.6. Byte Comparator pinout

This structure is implemented using eight XNOR gates, each of which checks if two bits –each one from the specific input byte– are equal. The outputs of these gates are sent to an AND gate which "decides" if all bits are equal. In other words, it examines whether the input bytes are equal or not. So, an AND gate of eight inputs has to be used, which is not so common because of the fanin constraints. As a result, the implementation uses two AND gates of four inputs and one AND gate of two inputs for the results which are the first ones.
The architecture, which is described before, is showed in Figure 4.9.

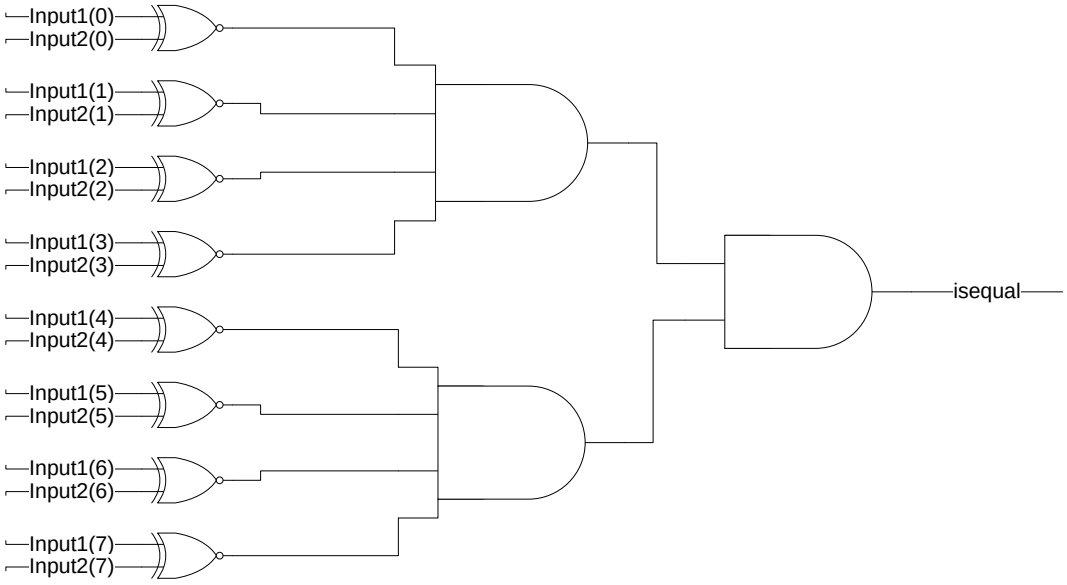

Figure 4.9. Byte Comparator Architecture

4.3.2 Find Longest Match Circuit

After the comparison, previously described, a 4-bit counter is placed for counting the '1' in row in the 16 bit long vector derived from the 16-Byte Comparator. Its output is the longest match counted, the largest number of the '1' in row.
Find Longest Match Circuit Interface is showed in Figure 4.11 and the functionality of the signals is described in Table 4.7.

Signal	Width (bits)	Type	Description
Isequal	16	Input	Input (shows which bytes are equal)
longestMatch	4	Output	Output (the calculated longest match)

Table 4.7. Find Longest Match Circuit pinout

Find Longest Match Circuit is implemented in behavioral VHDL. Firstly, the counter is initialized to 0. Every time, the iterator meets a '1', the counter is increased by 1 and, when it meets a '0', the counter is initialized to 0. Each time, the circuit checks if the new value of the counter is greater than the previous maximum value, and if so, it stores the value. Finally, after the whole procedure finishes, the maximum value is sent as output.
Figure 4.10 shows the pseudo-code used for implementing this component.

```
sum := 0 ;
maxSum := 0 ;

for i :=0 to 15 then

      if isequal(i) = '1' then

            sum := sum + 1 ;

      else

            sum := 0 ;

      end if ;

      if sum > maxSum then

            maxSum := sum ;

      end if ;

end loop ;

longestMatch := maxSum ;
```

Figure 4.10. Pseudo-code for counting the longest match

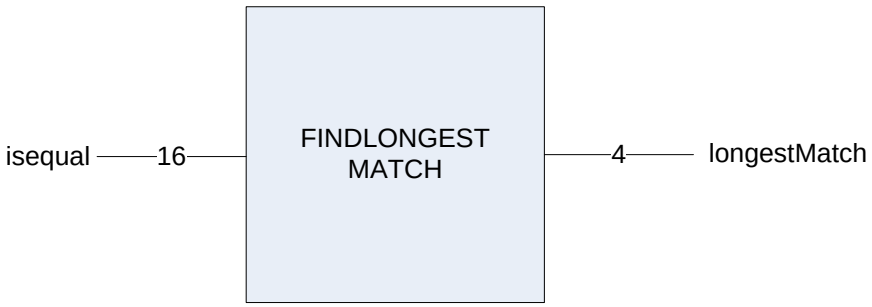

Figure 4.11. Find Longest Match Circuit Interface

4.3.3 Concatenation Circuit

Concatenation Circuit is the final part of a Pipeline Stage Comparator where the final result is created. Except for the longest match, it is important to know where the longest match is found, so an address must be stored in the final result, which is created by the concatenation of the longest match and the address of the first byte of the second input (derived from the crossbar) in BRAM. This address has two parts; the first one is the memory address of the memory bank (8 bits) and the second one is the position of the first byte in the memory bank (4 bits). If these parts are summarized with the longest match derived from the counter of Find Longest Match Circuit (4 bits), the size of the final result will be resulted, which is 16 bits long. The procedure, which is followed for the creation of the result, is simple and the structure of the vector derived is showed in Figure 4.12.

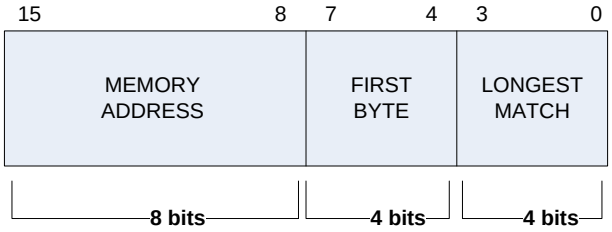

Figure 4.12. Structure of final result

4.4 Byte MUX 2-1 x 16

This module gets as inputs, four PRENA1-PRENA4, which are the results of the four comparisons of the previous pipeline stage, and four results of the Pipeline Stage Comparators of the current Pipeline Stage and chooses which of them will be further processed. The Interface of Byte MUX 2-1 x 16 is showed in Figure 4.13 and more details for the functionality of the signals are given in Table 4.8.

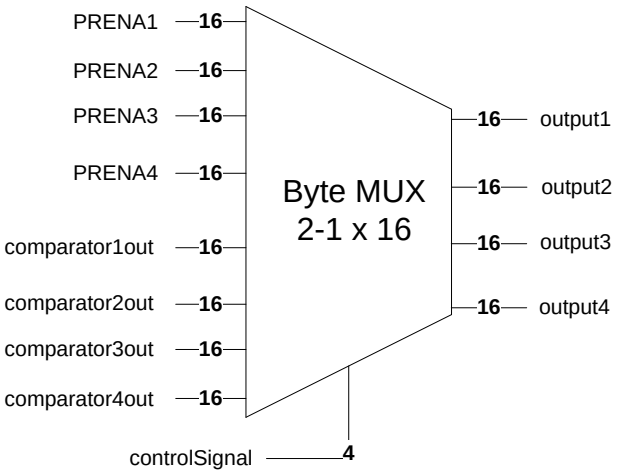

Figure 4.13. Byte MUX 2-1 x 16 Interface

Signal	Width (bits)	Type	Description
PRENA1	16	Input	Result of 1st comparison in previous pipeline stage
PRENA2	16	Input	Result of 2nd comparison in previous pipeline stage
PRENA3	16	Input	Result of 3rd comparison in previous pipeline stage
PRENA4	16	Input	Result of 4th comparison in previous pipeline stage
comparator1out	16	Input	Output of 1st comparator
comparator2out	16	Input	Output of 2nd comparator
comparator3out	16	Input	Output of 3rd comparator
comparator4out	16	Input	Output of 4th comparator
control	4	Input	Control signal
output1	16	Output	First output stream
output2	16	Output	Second output stream
output3	16	Output	Third output stream
output4	16	Output	Fourth output stream

Table 4.8. Byte MUX 2-1 x 16 pinout

Byte MUX 2-1 x 16 is implemented using four multiplexers. Every multiplexer chooses the value of output, which might be either the value of PRENAx or the value of comparisonxout. So every multiplexer is a 2-1 multiplexer with 16 bit long inputs – output. The structure of this implementation is presented in Figure 4.14.

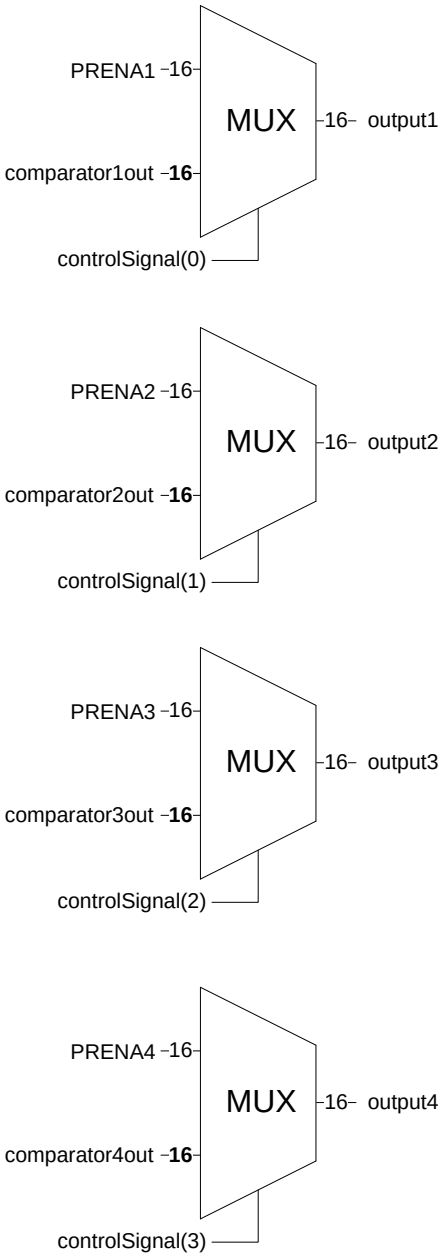

Figure 4.14. Byte MUX 2-1 x 16 Architecture

4.5 Longest Match Circuit (1 out of 4)

Longest Match Circuit is a simple structure of Comparator 4-1. It gets, as input, four 16 bits long vectors (the outputs of Byte MUX 2-1 x 16) and exports a 16 bits long vector, which is the largest one of the inputs. Figure 4.15 shows the interface of Longest Match Circuit and Table 4.9 describes the functionality of the signals.

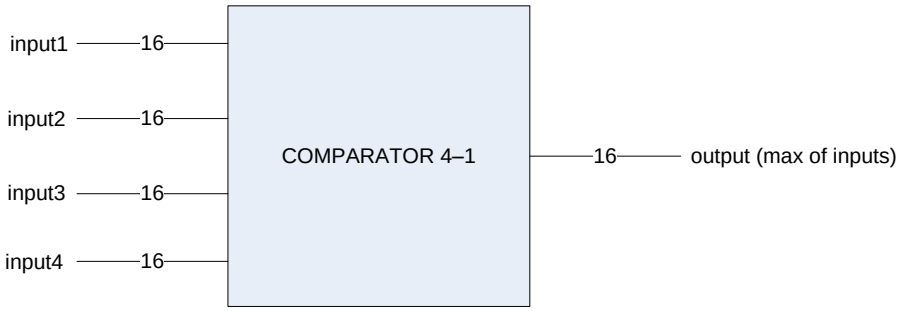

Figure 4.15. Longest Match Circuit Interface

Signal	Width (bits)	Type	Description
input1	16	Input	1st input of comparator
input2	16	Input	2nd input of comparator
input3	16	Input	3rd input of comparator
input4	16	Input	4th input of comparator
output	16	Output	Output of comparator (max of input1, input2, input3 & input4)

Table 4.9. Longest Match Circuit pinout

This module is implemented using three Comparators 2-1. The first one compares the first two inputs, the second one compares the other two and the third one compares the results of the other two comparators. It must be pointed out that every comparator compares only the 4 LSBs, which are the longest match and the remaining 12 bits are the memory address. At last, the final output is the concatenation of the largest "longest match" (4 bits) with the memory address (12 bits) chosen from the specific input. Figure 4.16 shows Longest Match Circuit Architecture.

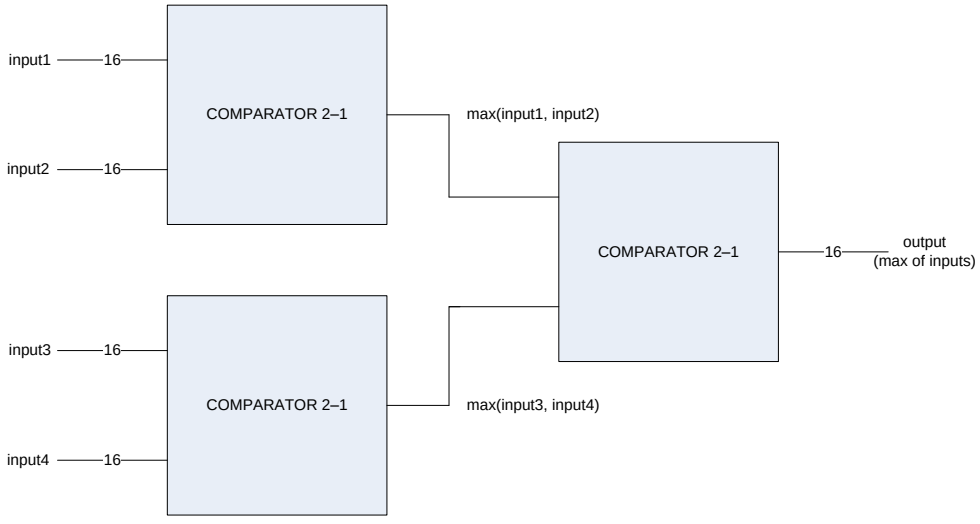

Figure 4.16. Longest Match Circuit Architecture

4.6 Longest Match (1 out of 2)

This part is the last one of the whole procedure of a pipeline stage. It is a Comparator 2-1 with 16 bits long inputs – output. It compares the two inputs and exports the greater one. As Longest Match Circuit does, it only compares the four LSBs of the input vectors and, finally, creates the final result using the procedure of concatenation, just as described in the previous section. Furthermore, the first input is the longest match calculated by the current pipeline stage and the second one is the longest match calculated by all the previous pipeline stages.
Table 4.10 shows the pinout of this comparator and Figure 4.17 presents Longest Match Interface.

Signal	Width (bits)	Type	Description
input1	16	Input	1st input of comparator
input2	16	Input	2nd input of comparator
output	16	Output	Output of comparator (max of input1 & input2)

Table 4.10. Longest Match pinout

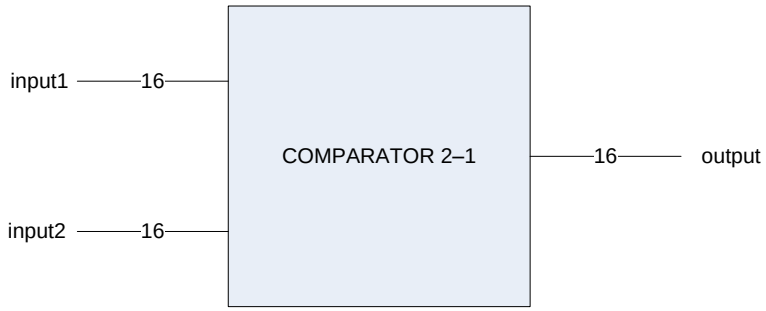

Figure 4.17. Longest Match Interface

4.7 Pipeline Registers

There are some pipeline registers used for separating each pipeline stage from the next one and storing the results which are derived from the whole procedure of the current pipeline stage. These results will be used for further processing to the next pipeline stages. A general interface of a pipeline register is presented in Figure 4.18 and Table 4.11 describes the functionality of the signals.

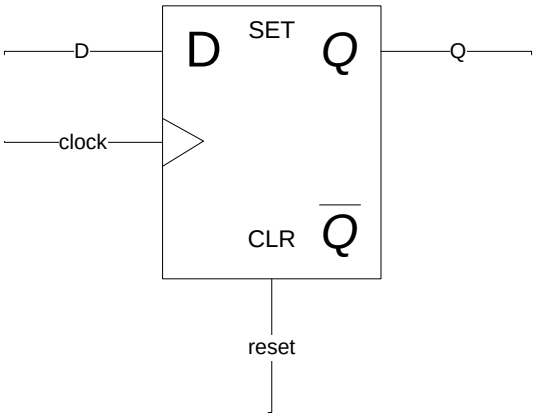

Figure 4.18. Pipeline Register Interface

Signal	Width (bits)	Type	Description
clock	1	Input	Clock signal
reset	1	Input	Register reset signal
enable	1	Input	Register enable signal
D	N	Input	Register input stream
Q	N	Output	Register output stream

Table 4.11. Pipeline Register pinout

The pipeline registers used are one for storing the unchanged memory address (N = 15), one for the unchanged look-ahead buffer (N = 128), one for the possibly altered LONMA (N = 16) and four for the new PRENAs (N = 16) which specify the four longest matches found on this stage. Table 4.12 points out all the registers used.

Pipeline Register	Input – output length (bits)	Description
ADDR	15	Memory Address
Look-ahead buffer	128	Look-ahead buffer
PRENA1	16	Result of 1st comparison
PRENA2	16	Result of 2nd comparison
PRENA3	16	Result of 3rd comparison
PRENA4	16	Result of 4th comparison
LONMA	16	Total Longest match

Table 4.12. Pipeline Registers used in each Pipeline Stage

4.8 Finite State Machine (FSM)

The use of a FSM is thought indispensable for securing the right operation of the procedure of each pipeline stage. According to the timing diagram described in chapter 3, the procedure of a pipeline stage is divided in five parts and it finishes after five cycles. First of all, during the first cycle, memory is accessed for gaining the first set of data, which is the first half of memory bank and, simultaneously, it is chosen to compare the four PRENAs (results from the previous pipeline stage). A second memory access takes place at the second cycle. The output data is the second half of memory bank and, as it is fully regained, the first comparison can take place, so it is implemented and the result is stored in PRENA1 pipeline register. Finally, the other three comparisons take place at the following three cycles and their results are stored in PRENA2 – PRENA4, similarly. Every of the cycles described above corresponds a state of the FSM created. All this information is summarized in Figure 4.19, which shows the loop of this procedure, and Figure 4.20, which shows the FSM scheme that presents the state sequence.
The first thought about the FSM needed is to create a FSM, which operates in four cycles; the first state is only used one time at system start (reset) and the fifth one does the first memory access again and goes back to the second one. This solution is chosen for improving the system performance, but it has a main drawback, because these four states are used for the four comparisons and PRENAs are compared only once, which is not desirable. So the FSM described is the only solution.
More details about the comparisons and which are the parts of memory bank compared are presented in sector 4.2, which describes the implementation of "Crossbar".

```
while
(1)   1st memory access (bytes  1 - 16) & PRENAs comparison
(2)   2nd memory access (bytes 17 - 31) & 1st comparison
(3)   2nd comparison
(4)   3rd comparison
(5)   4th comparison
back to while
```

Figure 4.19. Pipeline Stage procedure

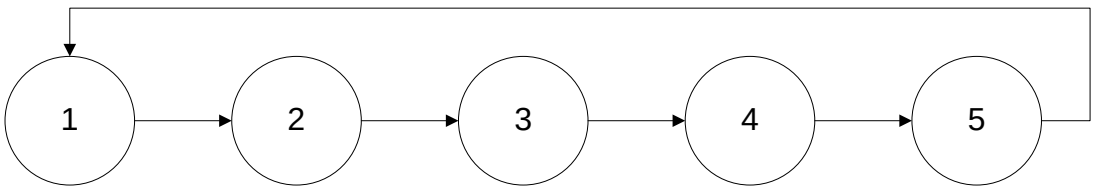

Figure 4.20. FSM scheme

The signals controlled by this FSM are summarized in Table 4.13.

Signal	Component	Description
wrbuffer	Crossbar	Write enable of latch
dsit	Crossbar	Control signal of deMUX
compNo	Crossbar	Control signal of MUXs
control	Byte MUX 2-1 x 16	Control signal of MUXs
prena1RegLdEn	PRENA1 Register	Enable of Register
prena2RegLdEn	PRENA2 Register	Enable of Register
prena3RegLdEn	PRENA3 Register	Enable of Register
prena4RegLdEn	PRENA4 Register	Enable of Register
firstbyte1	Pipeline Stage Comparator 1	Situation of first byte in memory bank (used in concatenation circuit for creating the final result)
Firstbyte2	Pipeline Stage Comparator 2	Situation of first byte in memory bank (used in concatenation circuit for creating the final result)
Firstbyte3	Pipeline Stage Comparator 3	Situation of first byte in memory bank (used in concatenation circuit for creating the final result)
Firstbyte4	Pipeline Stage Comparator 4	Situation of first byte in memory bank (used in concatenation circuit for creating the final result)

Table 4.13. Signals controlled by FSM

4.9 Pipeline Stage

Pipeline Stage is a component of a higher level than those previously described. Especially, it contains all the components described above; a memory (two BRAMs), a Crossbar, four Pipeline Stage Comparators, a Byte MUX 2-1 x 16, a Longest Match Circuit (Comparator 4-1), a Longest Match (Comparator 2-1) and some Pipeline Registers presented in sector 4.7. The connections between these components are described in the specific sectors and summarized in the Figure 4.21.
Moreover, this figure does not show the FSM which is placed for controlling the procedure of each Pipeline Stage. The FSM used is described in details in sector 4.8.

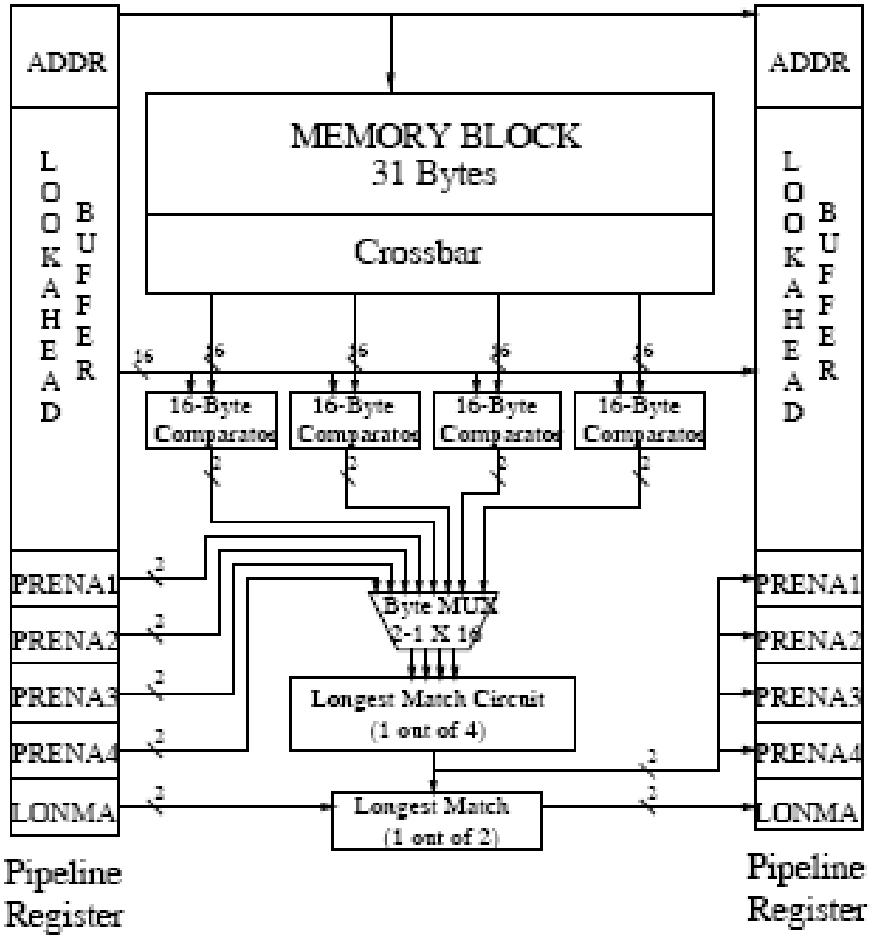

Figure 4.21. Block diagram of a Pipeline stage. Wire widths are in Bytes.

Pipeline Stage Interface is presented in Figure 4.22 and more details about signal functionality are presented in Table 4.14.

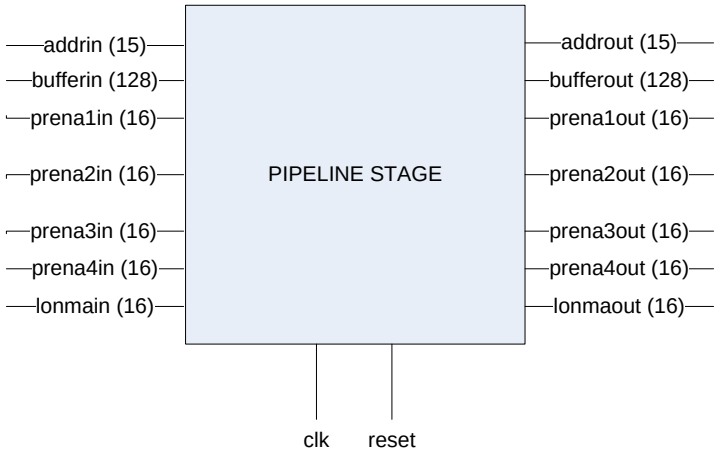

Figure 4.22. Pipeline Stage Interface

Signal	Width (bits)	Type	Description
clk	1	Input	Clock signal of system
reset	1	Input	System reset
addrin	15	Input	Memory address input
bufferin	128	Input	Look-ahead buffer input
prena1in	16	Input	PRENA1 input
prena2in	16	Input	PRENA2 input
prena3in	16	Input	PRENA3 input
prena4in	16	Input	PRENA4 input
lonmain	16	Input	LONMA input
addrout	15	Output	Memory address output
bufferout	128	Output	Look-ahead buffer output
prena1out	16	Output	PRENA1 output
prena2out	16	Output	PRENA2 output
prena3out	16	Output	PRENA3 output
prena4out	16	Output	PRENA4 output
lonmaout	16	Output	LONMA output

Table 4.14. Pipeline Stage pinout

4.10 Compressor

Compressor is the top level of this system. The main characteristics of the architecture used are:

- 256 pipeline stages placed in row (the outputs of pipeline stage n are connected to the inputs of pipeline stage n+1)
- 16 comparisons in parallel in every pipeline stage (increasing the comparisons' latency help us to improve the speed of the device).
- Memory repetition (100% memory overhead) for higher memory throughput and, consequently, higher speed.

This architecture is showed in Figure 4.23.

The interface of the device is similar to the one of the lower level (Pipeline Stage), because the inputs of the device are connected to the inputs of the first pipeline stage and the outputs of 256^{th} pipeline stage to the outputs of compressor. Figure 4.24 shows the interface described above. In addition, more details about the signals of this figure are given in Table 4.15.

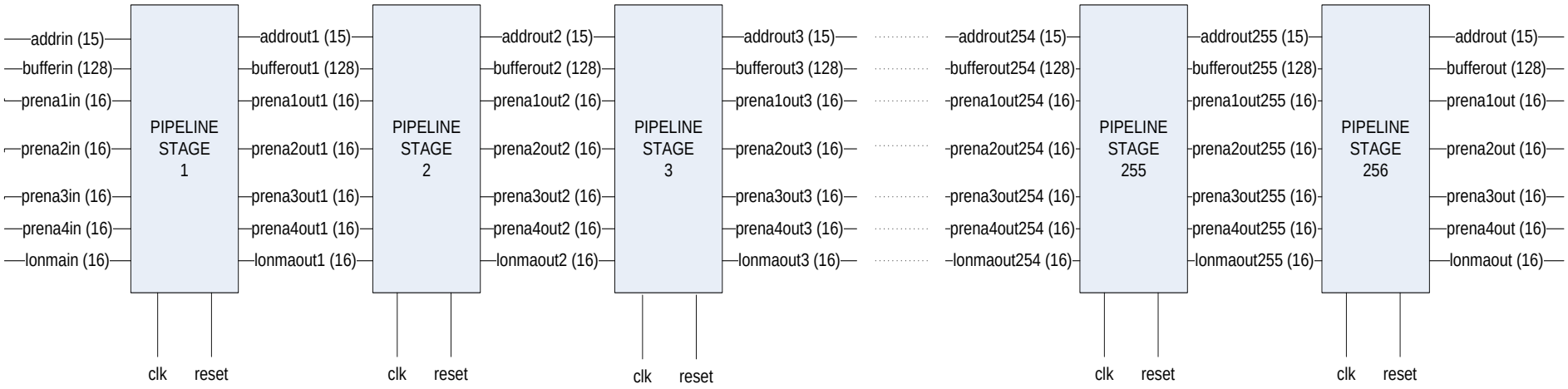

Figure 4.23. Compressor Architecture

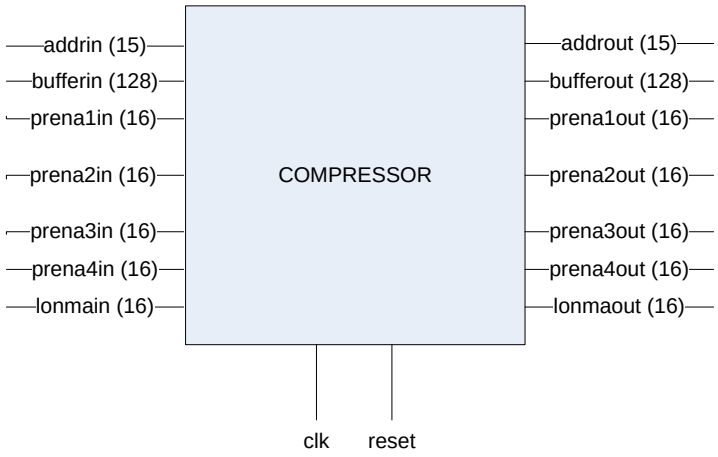

Figure 4.24. Compressor Interface

Signal	Width (bits)	Type	Description
clk	1	Input	Clock signal of system
reset	1	Input	System reset
addrin	15	Input	Memory address input
bufferin	128	Input	Look-ahead buffer input
prena1in	16	Input	PRENA1 input
prena2in	16	Input	PRENA2 input
prena3in	16	Input	PRENA3 input
prena4in	16	Input	PRENA4 input
lonmain	16	Input	LONMA input
addrout	15	Output	Memory address output
bufferout	128	Output	Look-ahead buffer output
prena1out	16	Output	PRENA1 output
prena2out	16	Output	PRENA2 output
prena3out	16	Output	PRENA3 output
prena4out	16	Output	PRENA4 output
lonmaout	16	Output	LONMA output

Table 4.15. Compressor Interface

The architecture described before and showed in Figure 4.23 is the optimal, but, in practice, it cannot be implemented because of the device utilization constraint. According to the measurements, 512 BRAMs must be used, but the largest FPGA today has only 336 and 20% more logic cells than the same FPGA has. So it is essential to find an alternative solution for implementing the compressor with 256 pipeline stages. The solution found is to implement the compressor on two FPGAs (half of the compressor on each one). The connection between them is implemented using an external interface. This architecture can be created because of the low system frequency (around 50 MHz) and it will be easy to create the external interface needed. This final architecture for the compressor is showed in Figure 4.25.

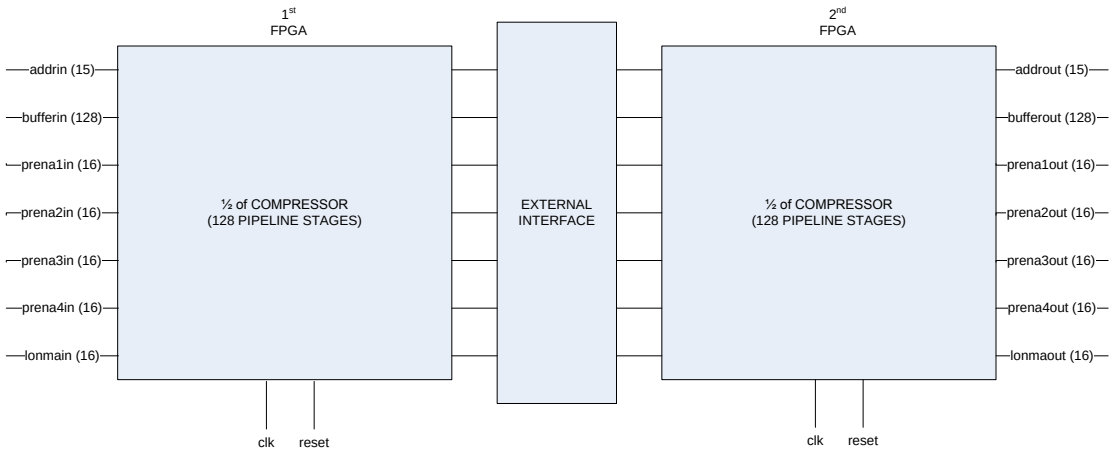

Figure 4.25. Compressor Architecture (with device utilization constraint)

4.11 Optimizations

Regardless the throughput derived from the main design described above is satisfactory, some experiments conclude that the performance can be improved. So it is essential to make some improvements – optimizations. The experiments show that the critical path of this device is the FLM(11) Circuit, the counter placed in Pipeline Stage Comparator for counting the number of equal bytes in row, which is predictable, because includes 16 steps, each one for every byte. Each step includes an adder, a comparator and some multipliers and all steps must be serial, because everyone needs the result derived from the previous one. Consequently, the work is concentrated in this component to improve the performance of the device. The remainder of this sector describes the optimizations chosen to be done in FLM Circuit for higher speed of the device.

(11) FLM: Find Longest Match

4.11.1 Optimization 1 – Pipelined FLM (16 pipeline stages)

The first thought is to make a fully pipelined FLM Circuit with 16 pipeline stages, each of which will be used for the calculation of a byte, but this is not potential, because this implementation leads us to a design, which needs 40 – 45 % more logic cells than the largest FPGA has available. This constraint leads us to characterize this attempt a failure and go on finding a different approach.

4.11.2 Optimization 2 – Pipelined FLM (8 pipeline stages)

After the previous futile attempt, it is decided to decrease the number of pipeline stages of this component. So 8-stage pipelined FLM Circuit is tested and the results show that it can be implemented using the available logic cells. Every stage must be responsible for calculating two bytes and designed for this purpose. It concludes an

adder, a comparator and two multiplexers for the calculation of every byte. The architecture used for implementing a FLM pipeline stage is showed in Figure 4.26.

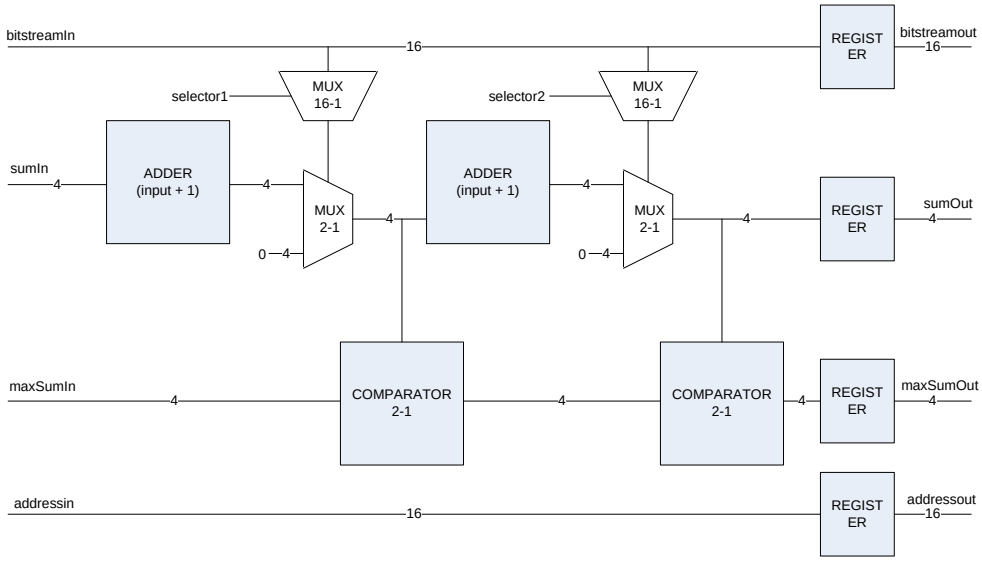

Figure 4.26. FLM Pipeline Stage Architecture

The adders increase the previously calculated match by 1 and create the new match, which is chosen if the specific bytes are equal. Otherwise, the new match is initialized to 0. This procedure is implemented with the use of two multipliers (one for each byte calculation). The first one selects the specific bit (refers to the specific bytes) from the bitstream (output of 16-Byte Comparator), which shows if the bytes are equal (‘1’) or not (‘0’). The second one gets as inputs the increased match and the constant “0000” and chooses the new match according to the result of the previous multiplier (if ‘1’, output is the increased match, else the constant “0000”). Furthermore, the new match is compared with the longest one calculated at the previous pipeline stages. Finally, bitstream and memory address go through the next pipeline stages; the bitstream for the selectors of multipliers in following pipeline stages and the memory address for the creation of final result in concatenation circuit.
Apart from the changes in the architecture, the procedure followed by each pipeline stage must be changed. So the FSM of the system is changed. The procedure still lasts five cycles, but there are 8 cycles needed for the initialization of every pipeline stage because of the 8-stage pipelined FLM circuit and, consequently, the increased latency of the device. Figure 4.27 shows the state sequence of the new FSM and Figure 4.28 presents more information about the procedure followed.

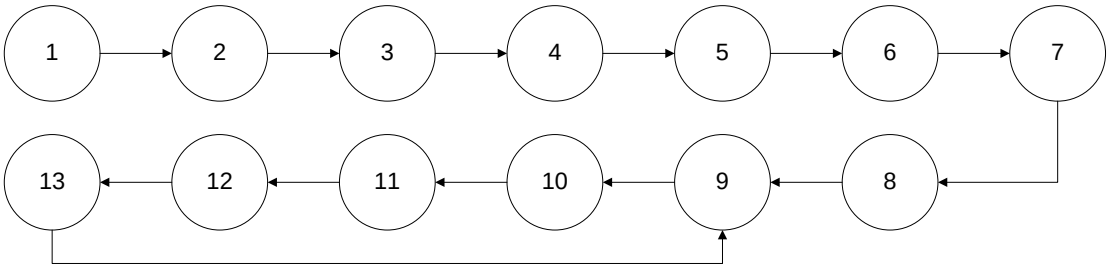

Figure 4.27. FSM scheme (optimized system)

```
(1)   1st memory access (bytes 1-16) & PRENAs Comparison (MUX)
(2)   2nd memory access (bytes 17-31), 1st comparison (Crossbar) &
      PRENAs Comparison (MUX)
(3)   2nd comparison (Crossbar) & PRENAs Comparison (MUX)
(4)   3rd comparison (Crossbar) & PRENAs Comparison (MUX)
(5)   4th comparison (Crossbar) & PRENAs Comparison (MUX)
(6)   1st memory access (bytes 1-16) & PRENAs Comparison (MUX)
(7)   2nd memory access (bytes 17-31), 1st comparison (Crossbar) &
      PRENAs Comparison (MUX)
(8)   2nd comparison (Crossbar) & PRENAs Comparison (MUX)
while
(9)   3rd comparison (Crossbar) & PRENAs Comparison (MUX)
(10)  4th comparison (Crossbar) & 1st comparison (MUX)
(11)  1st memory access (bytes 1-16) & 2nd comparison (MUX)
(12)  2nd memory access (bytes 17-31), 1st comparison (Crossbar) &
      3rd comparison (MUX)
(13)  2nd comparison (Crossbar) & 4th comparison (MUX)
back to while
```

Figure 4.28. Pipeline Stage procedure (optimized system)

This time, the procedure is divided in three parts, in contrast to the procedure previously described which is divided in the two parts. The first part is referred to the memory use, the second one to the part before the FLM circuit (as refer to the choice of crossbar which leads Pipeline Stage Comparators) and the third one to the part after it (as refer to the selection of Byte MUX 2-1 x 16).

5

Performance, Conclusions & Future Work

In this chapter, a performance comparison will be made between FPGA implementation of Titan-R and the results published by X-MatchPRO implementation in [8]. Furthermore, this chapter summarizes the work and the conclusions and suggests areas for further study.

5.1 Performance

This section is divided in two parts; the first one presents the results published by X-MatchPRO implementation and the second one provides the performance of this implementation plus the utilization of the device used.

5.1.1 X-MatchPRO Performance

As refer to X-MatchPRO implementation, two data sets have been chosen as representatives of network and computer-originated traffic: the memory data set is formed by data captured directly from main memory in a UNIX workstation used in an engineering environment. The disc data set is formed by typical data found in the hard disk of the same workstation.
Figures 5.1 and 5.2 show the compression performance comparison. It is common in networking and storage applications that data is present in small packets so the performance of these algorithms is evaluated in function of four different block sizes plus file-based compression. The "Y" axis is the compression ratio defined as the ratio output bits / input bits so the smaller the figure the better the compression. The "X" axis is the block size defined as the number of bytes in a block data to be compressed independently. This means that the dictionary is cleared each time a data block is processed. Memory data exhibits a strong 32-bit granularity because it is based on a 32-bit operating system so it suits well the X-MatchPRO algorithm. Compression improves with block size until a 4-Kbyte block size is used. This is a natural block size for memory pages and further increases in block size do not improve compression significantly.

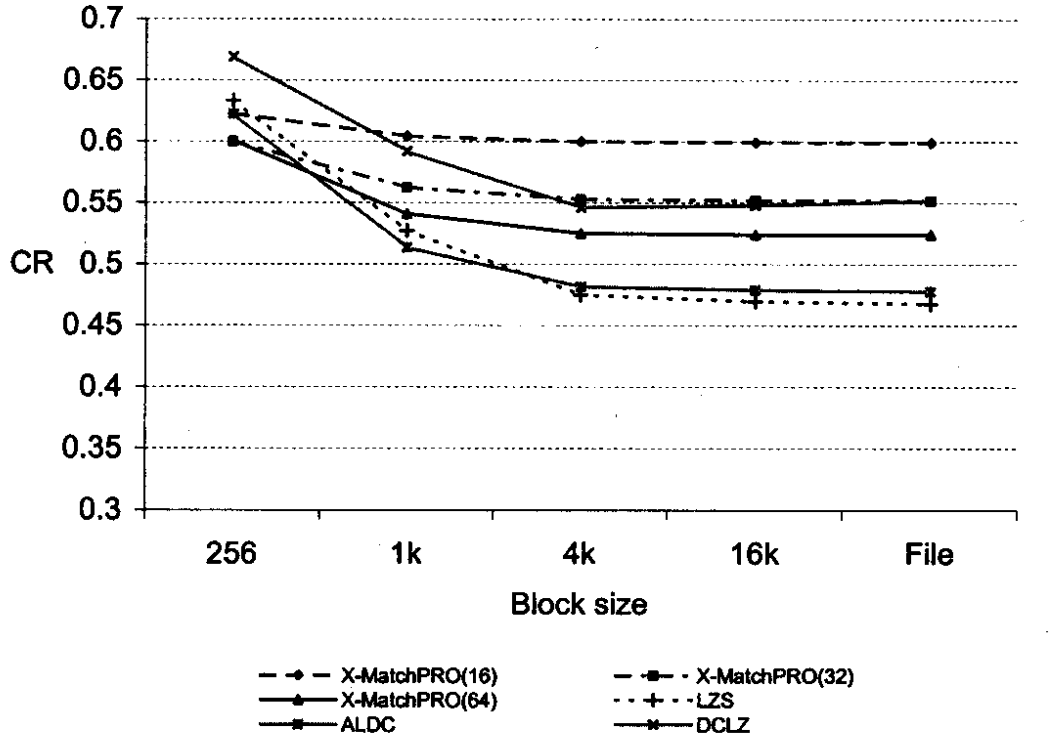

Figure 5.1. Compression performance on the memory data set

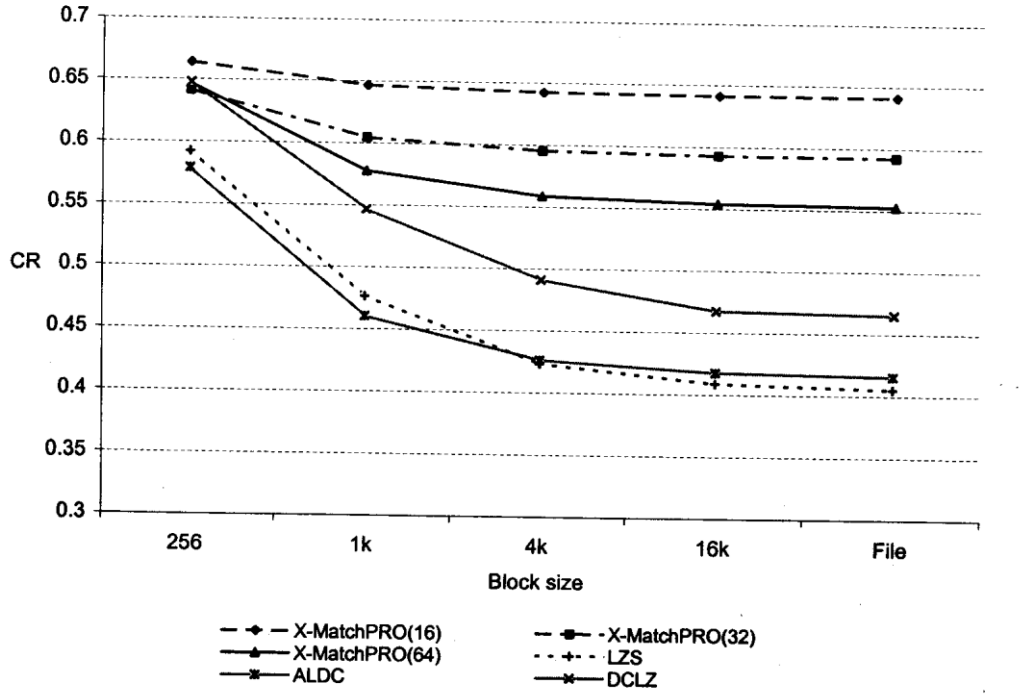

Figure 5.2. Compression performance on the disc data set

The disc data set of Figure 5.2 is more textually bias with a lot of database information so byte-oriented methods such as LZ derivatives have an advantage. It shows that compression improves with packet size until around 16 kB for the LZ-derivatives and 4 kB for X-MatchPRO. The smaller X-MatchPRO dictionaries tend to saturate earlier than their LZ equivalents.
Table 5.1 shows a summary of the features of lossless data compression devices. X-MatchPRO results are based on three different dictionary sizes: 16, 32 and 64 locations. A dictionary larger than 64 locations improves compression but the flip-flop rich architecture of the dictionary demands larger FPGAs. It is also necessary to replace the uniform binary coding of the match locations by a more complex coding technique. Otherwise, the extra number of bits required to code the match locations in a large dictionary damages the compression ratio specially when compressing small packets. These three implementations trade complexity for compression while speed remains invariant. Table 5.1 summarizes the characteristics of the X-MatchPRO algorithm as implemented in Xilinx, Altera and Actel technologies. The complexity figures correspond to the dictionary with 16 entries. Doubling the dictionary size increases chip complexity by a factor of 1.5 approximately.

DEVELOPERS	System Design Group Loughborough University		
CHIP	X-MatchPROv4 (16-word dictionary)		
PROCESS	0.18 micron SRAM-CMOS FPGA Xilinx VIRTEX-E	0.18 micron SRAM-CMOS FPGA Altera APEX20KE	0.25 micron FLASH-CMOS FPGA Actel A500K ProASIC
COMPLEXITY	5367 LUT's 55% of a XCV400EBG 432-8	5040 LC's 60% of a EP20K200EFC 484-1	9039 TILE's 70% of a A500K130-BG456
CLOCK SPEED	50 MHz	50 MHz	25 MHz
THROUGHPUT	200 Mbytes/s	200 Mbytes/s	100 Mbytes/s
FULL-DUPLEX PERFORMANCE	400 Mbytes/s	400 Mbytes/s	200 Mbytes/s
ALGORITHM	X-MatchPRO	X-MatchPRO	X-MatchPRO
EXTERNAL RAM REQUIRED	NO	NO	NO
COMPRESSION RATION	0.58 16 word 0.53 32 word 0.51 64 word	0.58 16 word 0.53 32 word 0.51 64 word	0.58 16 word 0.53 32 word 0.51 64 word

Table 5.1. X-MatchPROv4 Performance Summary

The X-MatchPRO chips use a lower-clock frequency than the ASIC implementations, but it can achieve higher throughput thanks to its internal parallel architecture able to process 4 B of input information in a single cycle while all the other solutions only process a single byte. All these chips use CAM circuits to implement the dictionaries and in the case of the fastest X-MatchPRO chips, each input symbol can be processed in a single cycle. Adaptation in the fast LZ1 implementations is based on keeping a window with the most recently seen symbols in the dictionary. Symbols enter and leave the dictionary in a FIFO(12) style, so model adaptation is simplified if compared with X-MatchPRO, where the best match must be resolved before the model is ready for a new cycle. X-MatchPRO solves the adaptation feedback loop that exists in its model with the use of the out-of-date adaptation mechanism that delays the arrival of match information to the dictionary by one cycle without affecting its efficiency. The packing and unpacking of compressed data is also simple in ASIC devices because they map variable-length streams of symbols to fixed length codewords so the boundaries between codewords are easily identifiable. On the other hand, X-MatchPRO codewords are variable in length and their unpacking is a more complex process and indeed, a performance limitation factor in the chip. The complexity and performance of the Altera Apex and Xilinx Virtex chips is comparable because both use a hierarchical architecture with SRAM switches based on logic cells (Altera) or logic elements (Virtex) with similar complexity and identical feature size. Actel ProASIC devices, on the other hand, use a flat architecture with fine-grained logic cells that increases routing complexity and negatively affects performance. ProASIC feature size is also larger than the Xilinx and Altera feature size and this is also a reason for lower performance.
To sum up, X-MatchPRO offers an unprecedented level of compression/decompression throughput in a FPGA implementation of a lossless data-compression algorithm for general applications. The hardware architectures have been verified in three different FPGA technologies. The fine granularity of the Actel ProASIC devices has proven very efficient to implement the flip-flop rich X-MatchPRO architecture. The higher granularity of the Altera and Xilinx technologies combined with a more advanced process has enabled throughputs well over the Gbit/s mark. The full-duplex implementation effectively uses the memory resources available in these FPGAs to simultaneously handle a compressed and uncompressed data stream. The architecture is easily scalable so it can be adapted to newer FPGAs with higher gate counts with little effort. The aim is to improve compression for the disk data set by increasing dictionary length and introducing more efficient coding techniques than simple uniform binary coding for the match locations. It is also expected that an ASIC implementation of this algorithm will be able to improve throughput by a typical factor of 3, if compared with a similar feature size FPGA.

(12) FIFO: first-in/ first-out

5.1.2 Titan II Performance

This section presents Titan II, which is the second version of an ASIC implementation of a high-speed compressor. Titan II developed by Y. Papaefstathiou and has the same architecture with Titan-R described in previous chapters. This implementation performance is only presented for reference, because any comparison between ASIC and FPGA implementation has no sense.

5.1.2.1 Performance versus hardware cost

To calculate the silicon area and the timing characteristics, the design was synthesized (using Synopsys Design Compiler V, http://www.synopsys.com), and placed and routed (using Silicon Ensemble-PKS, http://www.cadence.com) for UMC's 0.18-µm-CMOS technology, with a worst-case 2-input NAND gate delay of 0.25 ns, and a worst-case memory latency of 2.45 ns. The Titan II was initially designed for 1-Gbps networks. By using all the speedup factors described earlier, the compressor can indeed process data at a constant speed of more than 1 Gbps (1.17 Gbps), introducing a latency of 256 clock cycles or a few mean packet times. For this design to be usable in multigigabit networks (for example, 2.5 Gbps to 10 Gbps), the only alteration needed is to use 256 comparators in each pipeline stage instead of 64, letting the device perform all the necessary comparisons simultaneously. When using 256 comparators in each pipeline stage, the latency is slightly less than 3 ns, allowing Titan II to work with a clock rate of 333 MHz. As a result, every compressor can process data at a rate of more than 2.5 Gbps (2.63 Gbps). By plugging four of these compressors in parallel (and using the compressibility table to load-balance flows to compression units), the system developer can achieve a 10-Gbps total throughput. Obviously, in this case, the system administrator should take care to initialize the compressibility table such that it optimizes the load balancing. The far simpler decompression unit can process data at 2.7 Gbps without any alterations to its architecture. Consequently, to process data at a full-duplex rate of 10 Gbps, a compression or decompression system requires four pairs of compression and decompression units.

5.1.2.2 Silicon area versus compression gain

Using the network data presented earlier and altering the Titan II's hardware modules produced the results in Figure 5.3. A full-featured, single-dictionary device that can simultaneously compress data at 2.5 Gbps and decompress packets at 2.7 Gbps requires 230-Kbyte gates and 96-Kbyte, 1-bit SRAM cells. Consequently, using the 0.18-µm technology, this device is about 5.5 mm2. Moreover, a 10-Gbps device (with four cores operating in parallel) can easily fit in a 25-mm2 die (5 mm X 5 mm). If less die area is available, the chip's dimensions can be reduced, by reducing memory repetition (so that not all possible matches will be examined) and the sizes of the supported dictionaries. Reducing the die area results in lower compression gains, as Figure 9 clearly demonstrates. In particular, reducing the device's dimensions by a factor of four reduces the compression gain by about 30%; thus, if 2 mm2 are available, the worst HTTP traffic compression gain decreases from 48% to 35%.

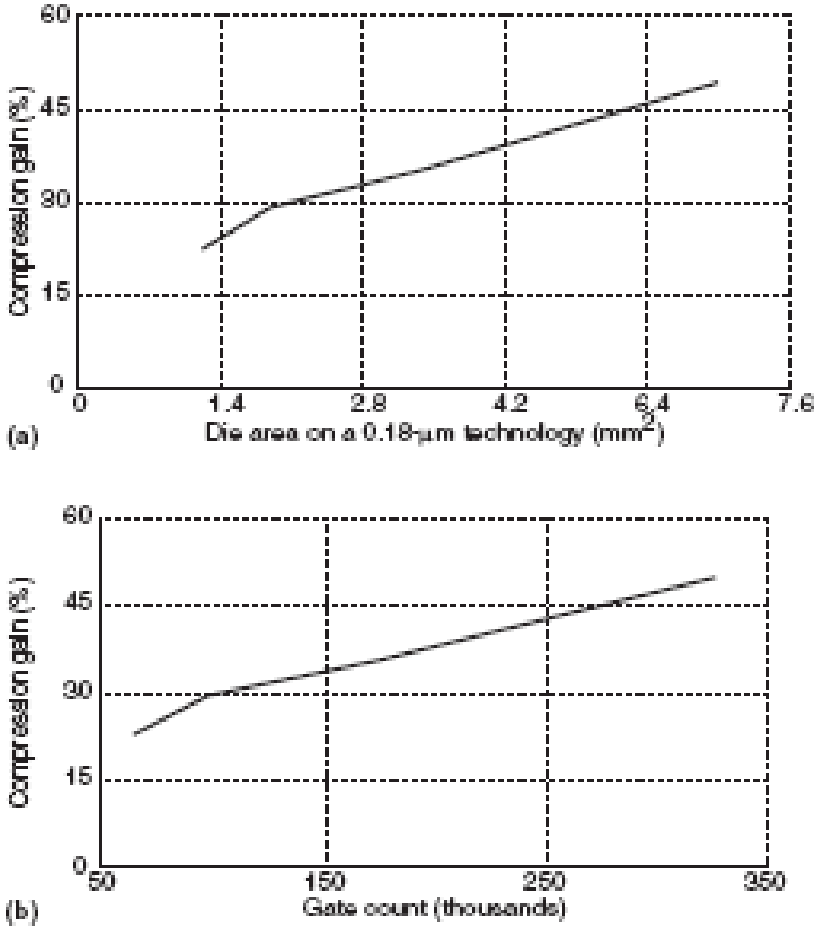

Figure 5.3. Results of using a 0.18−µm technology against the worst-case compression gain: die area in mm2 (a) and gate count (b).

5.1.2.3 Silicon area versus throughput

If the device is to be used on a lower-bandwidth IP network, the required silicon area would be reduced proportionally to the requested throughput. Because this increases the number of clock cycles for each major cycle, the device should perform fewer comparisons in parallel, and therefore requires fewer comparators. In fact, the number of comparators decreases proportionally to the decrease in bandwidth. Figure 10 shows this trade-off. Therefore, on a 2.5-Gbps network, the device requires 230-Kbyte gates and 96-Kbyte 1-bit SRAM cells. If the bandwidth serviced is 1.25 Gbps, the device needs 180-Kbyte gates and 102-Kbyte 1-bit SRAM cells. In general, reducing the gate count by 25% halves the device's bandwidth. For lower than 250-Mbps speeds, a slightly different architecture is used, and thus its silicon area estimations are not included in Figure 5.4.

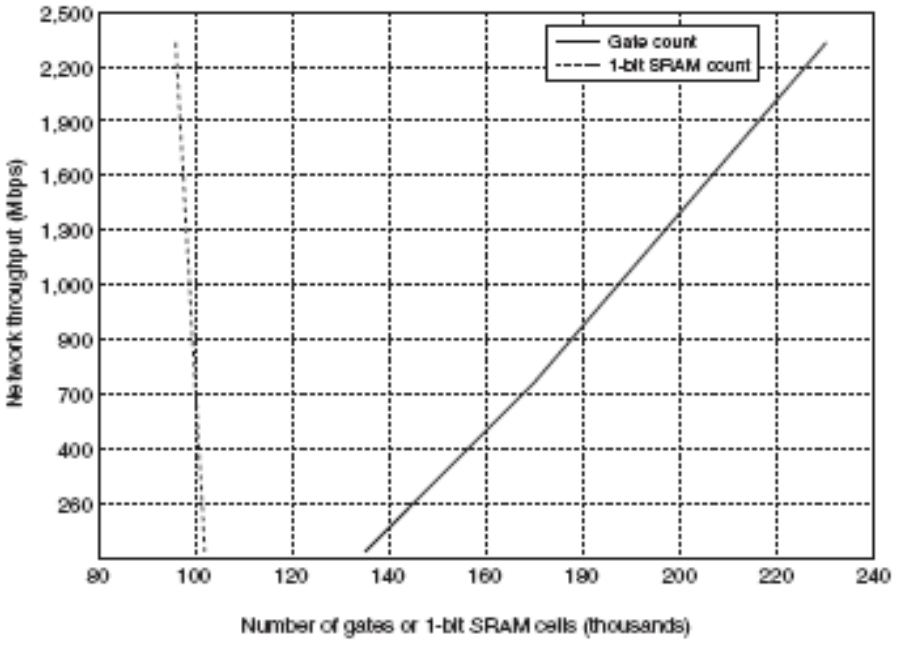

Figure 5.4. Gate and 1-bit SRAM count against device throughput.

Because the proposed compression system is highly flexible, it can also be used in lower-end networking systems, such as office gateways, to reduce contention on the link connected to the third-party network provider, thus making those gateway systems more effective. I am currently extending the proposed design approach to lower-speed access networks, especially wireless, where I expect to see similar benefits. When creating an efficient compression device in a wireless environment, special care should be taken to reduce the device's power consumption. I am therefore fine-tuning the overall architecture as well as the micro-organization of the various submodules to consume the least power possible.

5.1.3 Titan-R Performance

In previous chapters, the Titan-R architecture is presented. We present the idea on which the architecture is based and then we describe the design process and the implementation of it. Also, we propose an idea in order to improve the throughput of it. Now, we have to evaluate it and present the area cost as well as the performance and the throughput. These values will give us a clear view of the quality of our design. Furthermore, we will compare our design with X-MatchPRO architecture.
First of all, Xilinx ISE 9.1i was used in order to develop, synthesize and implement our design and ModelSim 6.0a in order to verify its correct functionality. The device family used is Virtex5, the device is "XC5VLX330T" and the device speed is -2. After completing the implementation we measure the area cost and the performance using the synthesis and place and route tools of ISE.
Before starting the evaluation of our design and presenting the results, we will give some information to the readers about the metrics we use in order to evaluate our system. The metric we used to measure the area cost is the number of Slice LUT's and Logic Cells (LC's). The number of Slice LUTs is the number of the Reported Slices multiplied by a factor of 4. Moreover, the number of BRAMs used is another metric measured for evaluating the memory cost.

Furthermore, some metrics are used in order to measure the speed of our design. Using Xilinx ISE synthesis tools, the Performance (Operating Frequency) of the system is measured. Multiplying this Frequency with the input bits of the system per cycle, the Throughput is calculated. Throughput is used widely by most researchers in order to evaluate their research. Another metric which shows the compression quality is the compression ratio which is defined as the ratio output bits / input bits, so the smaller the figure the better the compression.
This section contains, at first, area and memory evaluation and utilization. Then we evaluate the Performance of our system. At paragraph 5.1.1, X-MatchPRO Architecture is evaluated by combining Area, Memory and Performance results. Finally, our implementation is compared with related work.

First of all, the area cost for every structure of the Titan-R Architecture is shown in Tables 5.2 – 5.7.

Structure	**No of Logic Cells**	
	Default design	**Optimized design**
1-bit adder carry out:	512	-
2-bit adder:	512	-
2-bit adder carry out:	512	-
3-bit adder:	1536	-
3-bit adder carry out:	512	-
4-bit adder:	3584	8192
4-bit adder carry out:	512	-
Total (Adders/Subtractors):	**7680**	**8192**

Table 5.2. Adders/Subtractors Logic Cells Utilization

Structure	**No of Logic Cells**	
	Default design	**Optimized design**
128-bit register:	128	128
15-bit register:	128	128
16-bit register:	640	4224
4-bit register:	-	7680
8-bit register:	-	4096
Total (# Registers):	**896**	**16256**

Table 5.3. Registers Logic Cells Utilization

Structure	**No of Logic Cells**	
	Default design	**Optimized design**
1-bit latch	31744	31744
Total (# Latches):	**31744**	**31744**

Table 5.4. Latches Logic Cells Utilization

Structure	**No of Logic Cells**	
	Default design	**Optimized design**
2-bit comparator greater	1024	-
3-bit comparator greater	2048	-
4-bit comparator greater	4608	8704
5-bit comparator greater	512	-
Total (# Comparators):	**8192**	**8704**

Table 5.5. Comparators Logic Cells Utilization

Structure	**No of Logic Cells**	
	Default design	**Optimized design**
1-bit 16-to-1 multiplexer	-	8192
Total (# Multiplexers):	**-**	**8192**

Table 5.6. Multiplexers Logic Cells Utilization

Structure	**No of Logic Cells**	
	Default design	**Optimized design**
1-bit xor2	65536	65536
Total (# Xors):	**65536**	**65536**

Table 5.7. Xors Logic Cells Utilization

After "Synthesize XST" and "Place & Route" processes, some measurements for the device utilization of Titan-R implementations are derived. The results of these measurements of both implementations are presented in Tables 5.8 – 5.11. Especially, Tables 5.8, 5.9, 5.10 and 5.11 show Slice Logic Utilization, Slice Logic Distribution, I/O Utilization and Specific Feature Utilization of the device, respectively.

	Default design	**Optimized design**
Number of Slice Registers (out of 207360)	60672 (28%)	124304 (56%)
Number of Slice LUTs (out of 207360)	126938 (60%)	116440 (56%)
Number used as Logic (out of 207360)	126938 (60%)	116440 (56%)

Table 5.8. Slice Logic Utilization

	Default design	Optimized design
Number of Bit Slices used	181546	199632
Number with an unused Flip Flop	120874 (66%)	75328 (37%)
Number with an unused LUT	54608 (30%)	83192 (41%)
Number of fully used Bit Slices	6064 (3%)	41112 (20%)

Table 5.9. Slice Logic Distribution

	Default design	Optimized design
Number of IOs	448	448
Number of bonded IOBs (out of 960)	448 (46%)	448 (46%)
IOB Flip Flops/Latches	-	-

Table 5.10. I/O Utilization

	Default design	Optimized design
Number of Block RAM/FIFO (out of 324)	256 (79%)	256 (79%)
Number of BUFG/BUFGCTRLs (out of 32)	16 (50%)	3 (9%)

Table 5.11. Specific Feature Utilization

Except for the device utilization measurements, some measurements referred to the performance of the device are carried out. Table 5.12 shows the timing summary derived from these measurements.

	Default design	Optimized design
Minimum period (ns)	20.274	5.976
Maximum Frequency (MHz)	49.323	167.343
Minimum input arrival time before clock (ns)	18.643	6.274
Maximum output required time after clock (ns)	2.799	2.799
Maximum combinational path delay (ns)	0.962	0.957

Table 5.12. Timing Summary (Speed Grade: -2)

Regardless of the low clock frequency of this device for both implementations (around 49 MHz for the default implementation and 167 MHz for the optimized one), a great throughput value is achieved thanks to the architecture used which achieves a high level of parallelism. The throughput achieved is 2.525 Gb/s for the default design and 8.566 Gb/s for the optimized one.
Furthermore, the compression ratio is calculated by the division of the compressed size and the raw size. In this device, the values of compression ratio are derived from the Titan II hardware simulator according to the trace size.
Finally, the performance summary of Titan-R implementations (including the values of compression ratio) is presented in Table 5.13.

DEVELOPERS	Microprocessor & Hardware Laboratory Technical University of Crete	
CHIP	Titan-R	
	Default design	Optimized design
PROCESS	FPGA Xilinx VIRTEX5	FPGA Xilinx VIRTEX5
COMPLEXITY	126938 LUT's 60% of a XC5VLX330T	116440 LUT's 56% of a XC5VLX330T
CLOCK SPEED	49.323 MHz	167.343 MHz
THROUGHPUT	2.525 Gb/s	8.566 Gb/s
ALGORITHM	LZ77	LZ77
EXTERNAL RAM REQUIRED	NO	NO
COMPRESSION RATION	0.3253 (Trace size: 2.965) 0.4377 (Trace size: 1.714) 0.4398 (Trace size: 3.339) 0.3452 (Trace size: 1.846) 0.4629 (Trace size: 1.582)	0.3253 (Trace size: 2.965) 0.4377 (Trace size: 1.714) 0.4398 (Trace size: 3.339) 0.3452 (Trace size: 1.846) 0.4629 (Trace size: 1.582)

Table 5.13. Titan-R Performance Summary

Except for the measurements carried out on the whole system, some other measurements were done using devices with less pipeline stages for computing the performance achieved. The main advantage of these devices is that they need decreased area cost and can be implemented in smaller (and cheaper) FPGA devices. These implementations have much lower throughputs, but they may be useful according to the application. So, we chose to measure four implementations of 16, 32, 64 and 128 pipeline stages implemented in Spartan3, Virtex2P, Virtex4 and Virtex5 families, respectively. Tables 5.14 and 5.15 show the performance summary of these devices as refer to the default and optimized design, respectively.

DEVELOPERS	Microprocessor & Hardware Laboratory Technical University of Crete			
CHIP	Titan-R (default design)			
PIPELINE STAGES	16	32	64	128
PROCESS	FPGA Xilinx SPARTAN3	FPGA Xilinx VIRTEX2P	FPGA Xilinx VIRTEX4	FPGA Xilinx VIRTEX5
COMPLEXITY	28592 LUT's 69% of a XC3S2000	54759 LUT's 82% of a XC2VP70	113640 LUT's 84% of a XC4VLX160	126938 LUT's 60% of a XC5VLX330T
CLOCK SPEED	20.641 MHz (Speed grade: -5)	35.525 MHz (Speed grade: -7)	43.424 MHz (Speed grade: -12)	49.323 MHz (Speed grade: -2)
THROUGHPUT	66.051 Mb/s	227.36 Mb/s	555.827 Mb/s	1.263 Gb/s

Table 5.14. Devices with less pipeline stages Performance Summary (default design)

DEVELOPERS	Microprocessor & Hardware Laboratory Technical University of Crete			
CHIP	Titan-R (optimized design)			
PIPELINE STAGES	16	32	64	128
PROCESS	FPGA Xilinx SPARTAN3	FPGA Xilinx VIRTEX2P	FPGA Xilinx VIRTEX4	FPGA Xilinx VIRTEX5
COMPLEXITY	24359 LUT's 59% of a XC3S2000	51991 LUT's 78% of a XC2VP70	88972 LUT's 65% of a XC4VLX160	116440 LUT's 56% of a XC5VLX330T
CLOCK SPEED	70.889 MHz (Speed grade: -5)	145.427 MHz (Speed grade: -7)	150.095 MHz (Speed grade: -12)	167.343 MHz (Speed grade: -2)
THROUGHPUT	226.845 Mb/s	930.733 Mb/s	1.921 Gb/s	4.283 Gb/s

Table 5.15. Devices with less pipeline stages Performance Summary (optimized design)

The final measurements were carried out in designs with less pipeline stages, but this time, we used the same FPGA device (XC5VLX330T of Xilinx Virtex5 family). The only purpose of these measurements was to compare the performance achieved in all these designs. Table 5.16 presents the results for the clock frequency and the throughput of these systems. Moreover, Figures 5.5 and 5.6 show the graphs of these results.

Pipeline stages	Clock Frequency (MHz)		Throughput (Mbps)	
	Default design	Optimized design	Default design	Optimized design
8	52,877	175,108	84,6032	280,1728
16	49,277	171,174	157,6864	547,7568
32	47,732	167,343	305,4848	1070,9952
64	49,323	167,343	631,3344	2141,9904
96	48,51	167,343	931,392	3212,9856
128	49,323	167,343	1262,6688	4283,9808

Table 5.16. Clock Frequency and Throughput measurement results

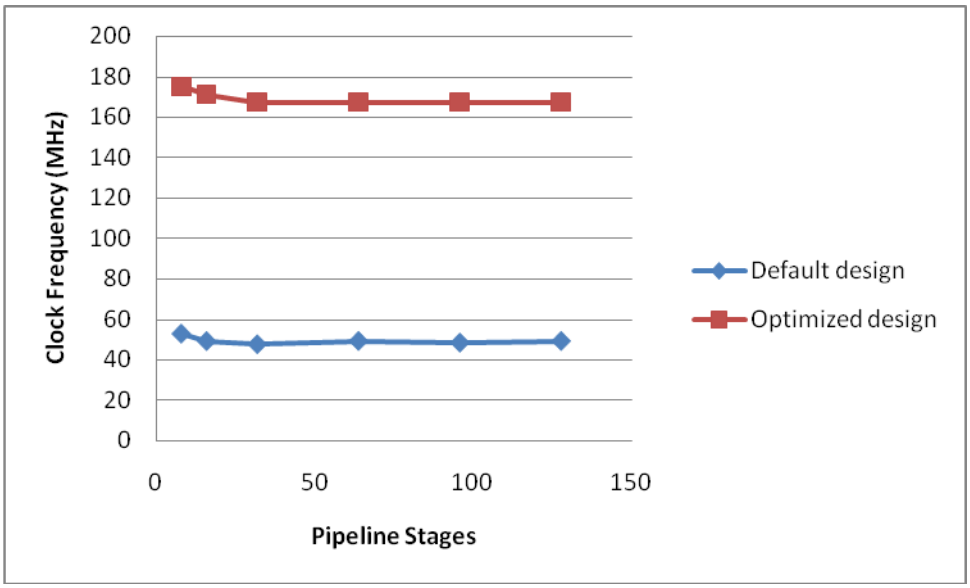

Figure 5.5. Clock Frequency Graph

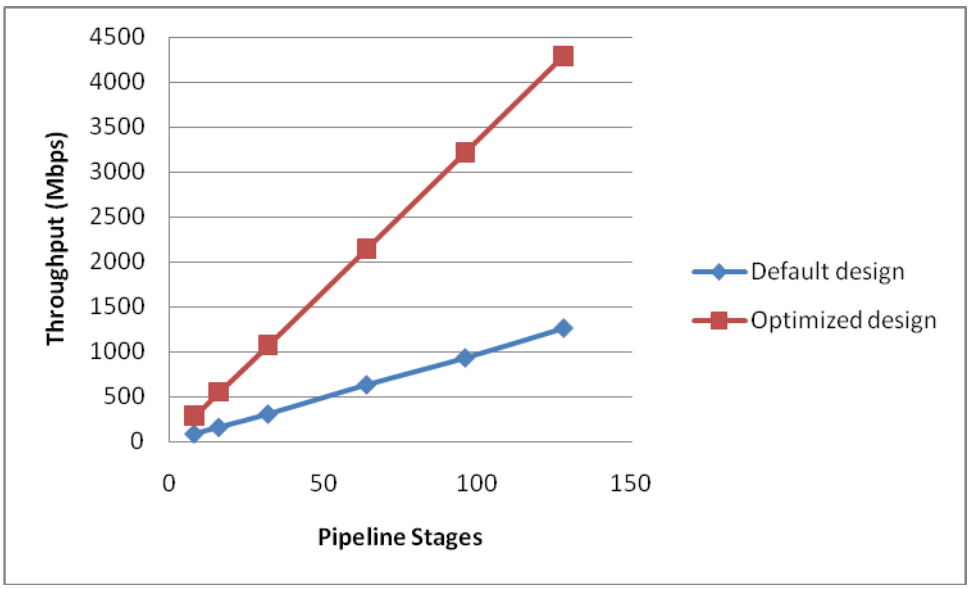

Figure 5.6. Throughput Graph

The final step of the procedure of system development was the simulation in order to verify the correct functionality of the system. The simulator used was Modelsim 6.0a. The system is simulated by using many different testbenches, which are divided in three main categories. The first category included testbenches chosen for having no match (longest match equal to zero) or low value of longest match, the second one included those with high value of longest match or full match (longest match equal to 16) and the third one included testbenches chosen in random or hierarchical method. Hierarchical testbenches was chosen for having linearly increased longest match according to the time. All the testbenches used show that the system works properly in every situation with the same good performance.

5.2 Conclusions

In this work, we developed the Titan-R Architecture which is an FPGA-based IPcomp Processor for high-speed networks. Two were the implementations proposed for creating Titan-R; the first one was the default implementation with a simple counter in Pipeline Stage Comparator which serially counts the longest match between two 16 bytes long vectors and the other one was to use a pipelined counter (with 8 pipeline stages) instead of the simple one for improving the performance of the system. The main goal of this thesis was to improve, as much as it was possible, the performance of this device and, especially, to increase the value of the throughput. That was the motivation for creating the optimized version of Titan-R.
The results obtained from the implementations of the device are encouraging, regarding they are compared to those reported by other systems as X-MatchPRO architecture. We achieved to develop a system with higher throughput and lower compression ratio (better compression) trading the area and memory cost and the latency of the system. The default implementation achieves a value of throughput around to 2.5 Gb/s and compression ratio lower than that proposed to the related work, but a large FPGA device with 60% LUT's Utilization and 79% BRAM's Utilization is used. Moreover, the latency is very high at a level of 256. Then, the optimized implementation achieves to improve the value of throughput around to 8.5 Gb/s by keeping the compression ratio and the device utilization at the same levels, but the main drawback of this implementation is that it increases the latency of the system and, especially, multiplies it by a factor of 8, due to the 8 pipeline stages used for implementing the counter (FLM circuit) in Pipeline Stage Comparator of every Pipeline Stage of the whole system. The new value of latency is 256 x 8 = 2048.

5.3 Future Work

Regardless of the encouraging results derived from the above implementations, there are some things necessary for the further development of the system.
The first idea is to implement the decompression unit by using the implementations of the compression one, which is not so complex, because the decompression unit is much simpler than the compression one and does not need to compare the input data to the dictionary data. It simply maintains the compressibility table and a 4-Kbyte dictionary for each compressible flow.
Another idea is to develop an interface with both the two units (compression and decompression one), which will dynamically change according to the operation needed. This interface is going to be developed by using dynamic reconfiguration.
Moreover, it might be useful to implement and simulate it on board, which is going to help us to see the full operation of this device and how useful it may be.
Another idea would be to connect many FPGAs in parallel in order to improve the performance of the system, because parallel processing will enable us to increase the throughput.
Furthermore, it may be useful a further study in I/O Interface. The first idea would be to choose external pins for being responsible for the communication of the system with the external world, but, this may cause to the performance of the system. So we must find other solutions for implementing the I/O issue. Serial and Ethernet ports, which are contained in modern FPGA devices, would be alternative solutions for solving this problem, but we do not know if there will be an impact on the performance.
Another subject for further research will be to make some measurements for the energy cost. This device trades area and memory cost for improving its performance (throughput), so it will be very useful to know the amount of the energy which is spent for achieving this high value of throughput.
To sum up, these extensions lead closer to an efficient FPGA implementation of Titan-R.

Bibliography

[1] I. Papaefstathiou, "Increasing packet network bandwidth through low level compression".

[2] I. Papaefstathiou, "Titan II: An IPcomp Processor for 10-Gbps Networks", *IEEE Design & Test of Computers*, November – December 2004.

[3] I. Papaefstathiou, "An Ultra High-Speed Compressor for Packet Networks".

[4] I. Papaefstathiou, "Accelerating ATM: On-line Compression of ATM Streams", *18th IEEE IPCCC'99, Phoenix, Arizona*, 10-12 February 1999.

[5] I. Papaefstathiou, "Compressing ATM streams", *IEEE Data Compression Conference 1999 (DCC'99), Utah*, 29-31 March 1999.

[6] I. Papaefstathiou, "Measurement based Connection Admission Control algorithm for ATM networks that use low level compression", *7th International Conference on Intelligence in Service and Networks, IS&N 2000*, February 25-28 2000, Athens, Greece.

[7] I. Papaefstathiou, "Compression simulation results", *White paper*, Cambridge University, http://www.cl.cam.ac.uk/ip2007/res.html.

[8] J. Nunez and S. Jones, "Gbit/s Lossless Compression Hardware", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 3, June 2003.

[9] J. Nunez, "X-MatchPROvw Compression/Decompression FPGA Processor", May 2002.

[10] J. Nunez and S. Jones, "The X-MatchPRO 100 Mbytes/second FPGA-Based Lossless Data Compressor".

[11] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression", *IEEE Trans. Information Theory*, vol. IT-23, no. 3, May 1978, pp 337-343.

[12] J. Hennessy and D. Patterson, "Computer Architecture A Quantitative Approach", 3rd ed., San Fransisco: Morgan Kaufmann Publishers, 1990.

[13] S. Sjoholm and L. Lindh, "VHDL For Designers", Prentice Hall, 1997.

[14] S. Brown and Z. Vranesic, "Fundamentals of Digital Logic with VHDL Design", McGraw-Hill, 2000.

[15] Xilinx, “Spartan-3 FPGA Family: Complete Data Sheet”, May 25, 2007.

[16] Xilinx, “Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet”, October 10 2005.

[17] Xilinx, “Virtex-4 Family Overview”, October 10 2006.

[18] Xilinx, “Virtex-5 Family Overview: LX and LXT Platforms”, January 4 2007.

[19] Xilinx, “Development System Reference Guide”.

[20] Xilinx, “ISE 9.1i Quick Start Tutorial”.

[21] Xilinx, “XST User Guide”.

[22] Xilinx, “Synthesis and Simulation Design Guide”.

[23] Modelsim, “Modelsim SE User’s Manual”.

[24] Modelsim, “Modelsim SE Reference Manual”.

[25] Modelsim, “Modelsim SE Tutorial”.

[26] Modelsim, “Modelsim Quick Guide”.