On the Design and Implementation of an FPGA-based Lossless Data Compressor

Miguel Morales-Sandoval and Claudia Feregrino-Uribe

National Institute of Astrophysics, Optics and Electronics
Computer Science Department
Luis Enrique Erro # 1, Sta. María Tonantzintla, Pue., 72840, México
{mmorales, cferegrino}@inaoep.mx

Abstract. This paper presents a hardware design and implementation for Lempel-Ziv data compression. The implementation is based on the systolic array approach employing two simples processing elements (PE's). Previously to implement the design, we select the buffer size based on software simulations. By selecting a specific size of the buffer, we can estimate how much area will be required, what compression ratio the compressor will achieve and also, what throughput the compressor can reach. Based on such simulations, a prototype of the compressor was implemented in a Xilinx XC2V1000 FPGA device employing a 512-byte searching buffer and a 15-byte coding buffer. The architecture can achieve a throughput of 11 Mbps while occupying 90% of the FPGA resources. An immediate application of this compressor is to work jointly with a public key cryptographic module.

Resumen. Se presenta la implementación en FPGA de algoritmo LZ77 para compresión de datos sin pérdida basada en un estudio realizado del impacto que presenta la selección del tamaño de los buffers. La arquitectura del compresor se implementa mediante el enfoque de arreglos sistólicos empleando dos elementos de procesamiento simples. Un prototipo del compresor de datos se realiza en un FPGA Xilinx XC2V1000 empleando el 90% de los recursos disponibles y logrando un rendimiento de 11 Mbps. Debido a su bajo costo en área, una aplicación inmediata de la arquitectura desarrollada es para operar conjuntamente con un módulo de cifrado de llave pública.

Key words: LZ77, FPGA, Systolic array.

1 Introduction

A common problem in computer data networks has been always the data rate. In this environment, the most important technique to improve the performance of a network is data compression. Data compression benefits in the sense that the process compression-transmission-decompression is faster than the process of transmitting data without compression. The main motivation to compress data is the cost reduction for transmitting and storing data.

Data compression is the codification of a data body D into a smaller data body D' [1]. The compression is lossless if D can be recovered (decompressed) entirely from D'. Data can be compressed only if it has redundancy, so, when

implementing compression algorithms, the search for redundancy implies a lot of operations, many times complex, and current processors do not have machine instructions to perform these operations efficiently. For this reason, a hardware solution is well suited to implement this kind of algorithms, especially for real time data processing.

Compression performance is measured according to the compression ratio the methods achieve; this measurement is obtained following equation 1. A good data compressor must achieve a compression ratio less or equal to 0.5.

$$R_c = \frac{Size_{out}}{Size_{in}} \tag{1}$$

For lossless data compression, algorithms can be dictionary-based or statistical. A statistical compressor can achieve better compression ratio than a dictionary-based method but its computational complexity is higher. In both statistical and dictionary-based methods, a trade off between compression ratio and execution time needs to be established. Depending on the application, do not always the best compression method is required.

In this paper, we evaluate the LZ algorithm and implement a variant [5] of its first proposal [2]. We analyze how compression ratio and throughput are affected depending on the buffer's size selected. This study provides a good reference for making decisions when implementing such algorithm in both hardware and software. Moreover, we present the results of a hardware implementation of this algorithm on a FPGA device to show the area requirements.

The paper is organized as follows: Section 2 explains the compression algorithm, Section 3 comments three approaches when implementing LZ-based algorithms in hardware and summarizes some related papers, Section 4 shows the performance of the algorithm depending on the selected buffer's size, Section 5 describe the hardware architecture of the implemented LZ compressor and summarizes synthesis results, finally, section 6 concludes this work.

2 The LZ77 algorithm

The LZ77 algorithm was proposed by Ziv and Lempel in 1977 [2]. It is a dictionary-based algorithm for lossless data compression that can achieve an average compression ratio and is considered universal, that is, it does not depend on the type of data being compressed. LZ77 algorithm was the first proposal of data compression based on a string dictionary instead of symbols's statistics. Since its proposal, this algorithm has been improved in order to achieve better compression ratios and to reduce the required processing time [3], [4], [5].

The idea behind the LZ77 algorithm is to compress by replacing a symbol string by a pointer or position in a dictionary where such strings occur. The algorithm uses two buffers called searching-buffer and coding-buffer, see figure 1 a). Initially, the coding buffer is filled up with the first input symbols. Data compression is achieved by performing two steps. Step one consists of finding the longest substring in the searching buffer being the prefix in the coding buffer. If such string exists, a codeword is generated consisting of the pointer (P) or position

in the searching buffer of the matched string and its length (L). Steps 2 is depicted in figure 1 b).

In the second step, if a codeword was generated in the previous step, L new symbols are entered to the coding buffer by shifting to the left the symbols in both searching and coding buffers. Figure 1 c) depicts the second step for compression.

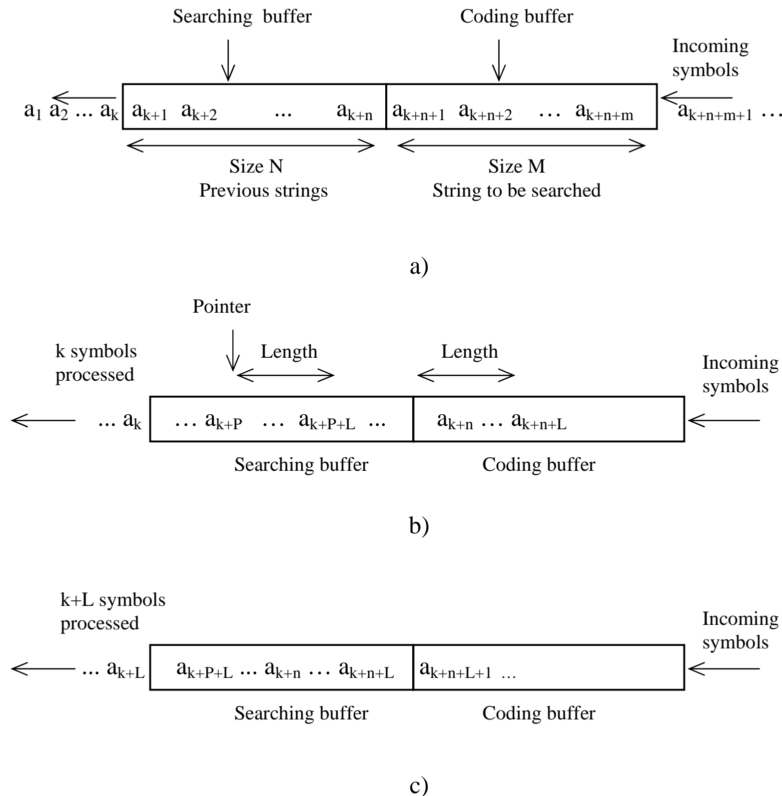

Fig. 1. LZ77 algorithm. a) The buffers, b) step 1 and c) step 2 in the compression process.

To avoid expansion in the output file, an especial action must be performed when a codeword substring is not found or when the length of the string found is less than the codeword size. Often, the action taken is to store the first symbols in the coding buffers whom size is less than the codeword and without compressing them. This implies the use of an extra bit to differentiate between compressed and uncompressed codes.

Finding the longest substring is a key operation in the algorithm and also, the most time consuming. Searching for all possible substrings sequentially is a problem of complexity *O(MN)*, so the execution time depends strongly on the size of the buffers.

The LZ77 algorithm has two major advantages among the known lossless data compressors. The first one is that it does not require prior knowledge or statistical characteristics of the symbols. This fact lets faster compression because a second pass over the data in not required as occurs in some statistical statistical methods. The second advantage is that the decompression process is easier and faster than the compression one. These two reasons made LZ77 attractive for us to implement it and study it as a competitive lossless data compressor to be used previous to an elliptic curve cryptographic system.

Current dictionary-based lossless data compressors are based on the ideas of Ziv and Lempel and software implementation of such algorithms can be found in applications such as compress, zoo and pkzip.

3 Related work

Many lossless data compression hardware implementations have been reported, either statistical or dictionary based. On one hand, statistical lossless data compressors have been shown to be more expensive than dictionary based implementations, essentially in area requirements, although they provide better compression ratios [6], [7]. On the other hand, three approaches are distinguished in the hardware implementation of dictionary-based methods: the microprocessor approach, CAM (Content Addressable Memory) approach and systolic array approach [8]. The first approach does not explore full parallelism and is not attractive for real time applications. The second one is very fast but it is costly in terms of hardware requirements. The systolic array approach is not as fast as the CAM approach but its hardware requirements are lower and testability is better. The main advantage of this approach is that it can achieve a higher clock rate and it is easy implemented.

Some papers, where a systolic approach for implementing the LZ77 algorithm is selected, have been reported. In these papers, the parallelism of the LZ77 algorithm is achieved by studying the data dependences in the computations. A dependence graph is drawn and form it a processor array is derived.

In [8], the systolic array is composed of two different types of processing elements designed in such way that each one consumes few resources. The number of type 1 PE's is only determined by the size of the coding buffer. This design was implemented on an ASIC platform using a 4.1K SRAM for the dictionary and 32 processing elements. The reported throughput of the design was 100 Mbps operating at a clock rate of 100 MHz. In [9], an FPGA LZ77 implementation is reported. The implementation requires four Xilinx 4036XLA FPGAs to achieve 100 Mbps throughput. The buffer's size was 512 for the searching buffer and 63 for the coding one. In [10], a VLSI chip is fabricated. It contains 16 processing elements to find the longest substring in the LZ77 algorithm. The chip operates at 100 MHz and has a throughput of 10 Mbps if a 1K-searching buffer and a 16-coding buffer are used. As mentioned in the paper, if ten chips are connected in parallel, a compression rate of about 100Mbps can be achieved.

These reported papers give us an idea of what can be expected from a hardware implementation but does not give us a guide to select the better parameters according to a specific application. In the following section, we show the results and comments about some software simulations we have carry out of the LZ77

algorithm in order to find the better choices of the size of the buffers in a LZ77 implementation according to a specific application.

4 The LZ77 algorithm: implementation issues

In this section we present the simulation results and some comments about the performance of the LZ77 algorithm for different buffers sizes. The code was written in C and the performance was calculated according to simulations results using the Calgary Corpus [11].
Two aspects have to be considered when implementing the LZ77 algorithm. In some cases, we will be interested in reaching a high compression ratio and in other ones we will sacrifice compression ratio in order to perform the compression faster. Another aspect is the throughput of the compressor. This is an important issue when the compressor is going to work jointly with another module. For that reason the compression performance needs to be known and it needs to be evaluated if it can operate transparently with another modules.

Depending on the systolic array design, the latency to get a new codeword varies. The throughput and compression ratio improves when longer strings are found (matched), but this happens when a large searching buffer is used, what implies greater latency and longer codewords. Besides, a smaller buffer implies latency reduction but compression ratio gets worst. In figure 2, a graphic that shows what compression ratio can be achieved for different sizes of N and M is depicted. As shown in figure 1 a), N is the size of the searching buffer and M is the size of the coding buffer.

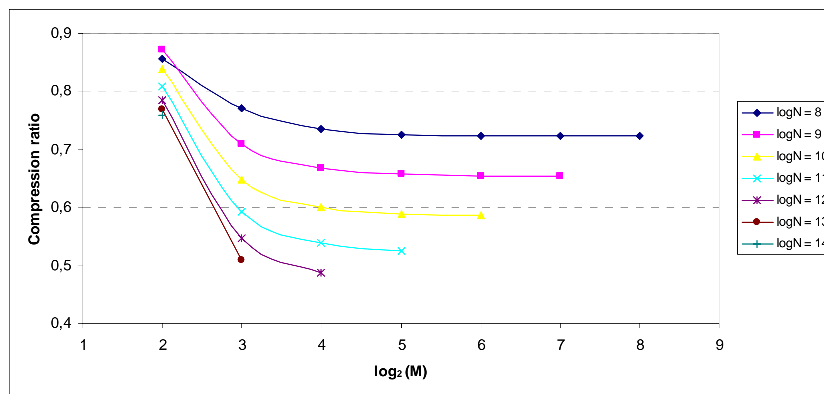

Fig. 2. Compression ratio for different buffer sizes

In this experiment, the codeword is up to 2 bytes long. From figure 2, we can infer that we can only achieve good compression ratios for a searching buffer greater that 512 and a coding buffer greater than 15. Also, notice how a better compression ratio is achieved for a searching buffer greater than 4K and coding buffer greater than 7. In figure 2, for any value of N, as the number of bits for the coding buffer increases, the compression ratio improves but only up to a specific threshold. We can see that every line in the graph in bounded, so, compression

ratio improves a little bit when the coding buffer in greater that 32. According to the required application, we can chose values for N and M greater that 512 and 32 respectively.

Other important aspect to consider is the expected throughput that the compressor can achieve. Theoretically, data rate can be estimated by the equation 2.

$$D_r = clk\frac{L_s \cdot w}{N + M} \tag{2}$$

In equation 2, clk is the frequency (cycles per second), L_s is the longest match that can be found and w is the size in bits of the symbols being compressed.

This equation gives us an estimation of the compression throughput and it is valid only for the best case that happens when the length of the strings matched is equal to the maximum value Ls. However, we also need to known what occurs in the average case. The average number of symbols processed in each codification step was calculated and it is shown in figure 3.

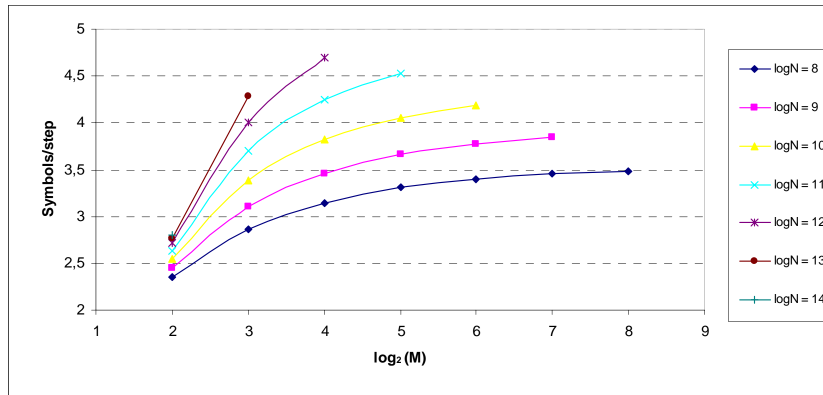

Fig. 3. Average of symbols processed in each codification step

According to the graphic in figure 3, the best throughput is obtained when the searching buffer is 4K and coding buffer is 16. As in figure 2, average number of processed symbols does not increase when coding buffer is large. In the case of a 512-searching buffer, the coding buffer can be 32, 64 or 127.

Another important fact derived of our simulations is that when the size of the searching buffer is fixed and the size of the coding buffer varies, the compression ratio always improves if N grows. In some cases, varying the size of the coding buffer may be will not improve the compression ratio but the throughput will be increased.

5 Compressor Architecture

A systolic array, used in this work, for the compressor is based on [9]. The compressor architecture is depicted in figure 4. The (M+N)-buffer is composed of two buffers: the first one is called the up-buffer to implement both the searching and the coding buffer, the second one called the shifter-buffer, that feeds the (M+1)-PE array. One N-bit counter is required to keep the current pointer in the searching buffer for the substring being matched. The codeword module checks for data expansion. A codeword is output only if the length of the last substring matched is greater that the length of the codeword. The control module coordinates the compression process performing the two steps in the LZ77 algorithm. This module is implemented as a finite state machine.

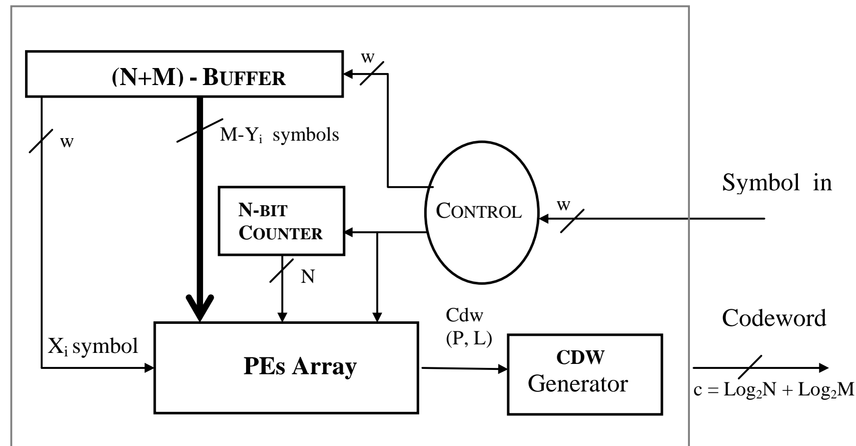

Fig. 4. General architecture of the LZ compressor

The buffer is composed of registers connected in cascade. The up-buffer is controlled by an enable signal to permit entering new symbols to the buffer each clock cycle. The shifter-buffer has a similar structure to the up-buffer but it incorporates a multiplexer in each register. Also, an enable signal allows emitting a new Xi symbol to the PEs array every clock cycle. The block diagram of the buffer is shown in figure 5. In this figure, R represents a w-bit register and X_i represents the current value of each element in the buffer. The number of PEs in the array is determined only by the size of the coding buffer. The PEs array is composed of M Type-I PEs and 1 Type-II PE. All Type I PEs are connected forming an array; the type II PE is placed at the end of such array. This last PE keeps the pointer and length that are identified in the systolic array while symbols in the searching buffer enter serially to the type I PE array.

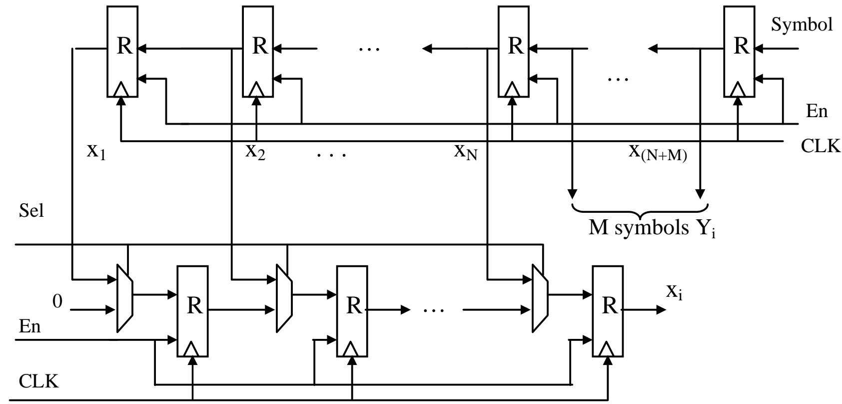

Fig. 5. Searching and coding buffer implementation

Processing elements are depicted in figure 6. The type I processing element consist of one w-bit equal comparator, one $\log_2 N$-bit, $\log_2 M$-bit and w-bit register, two flip-flops, one $\log_2 M$ multiplexer and one 4-input AND gate. Type II PE consists of one $\log_2 N$-bit and $\log_2 M$-bit multiplexers, one $\log_2 N$-bit and $\log_2 M$-bit registers and a $\log_2 M$-bit greater-comparer. The systolic array of this compressor consumes few resources and the number of PEs is only determined by the size of the coding buffer.

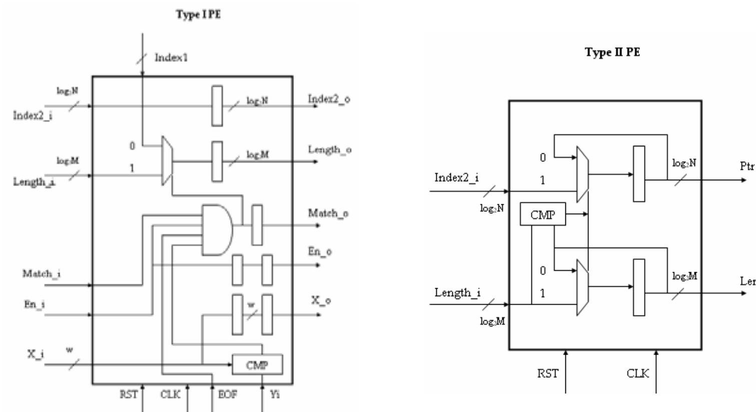

Fig. 6. Type I and Type II processing elements (PEs)

5.1 Implementation

The compressor was fully described in VHDL language and synthesized for the Xilinx XC2v1000 FPGA. The most area consuming module of the compressor is the buffer. The basic cell in the shifter buffer is composed of a w-bit register and a w-bit multiplexer, for w = 8, each one of this cells consumes 5 slices. On the contrary, the systolic array occupies fewer resources, for $\log_2 N = 9$ and $\log_2 M = 4$, type I PE only requires 20 slices while type II PE requires 10 slices. The main synthesized modules are summarized in table 1. In the design, the critical path is determined by the comparer included in the type II PE.

Since latency in each codification step is M+N, and all the architecture is dominated by a 219 MHz clock, we can derive, based on figure 3, that throughput of the compressor is 11Mbps.

Table 1. Basic elements of the compressor architecture

Module	#Slices
Type I- EP	20
Type II- EP	10
Buffer cell	5

6 Conclusions

We presented an extensive study about how the size of the buffers affects the throughput and compression ratio in the LZ77 algorithm. Knowing how the size of the buffers affects the compression ratios, throughput and latency is very important to make the better decision when implementing this algorithm. A prototype was implemented for buffers of different sizes that showed to be better for implementing a variant of the LZ77 algorithm in the FPGA we used, because of the high area requirements of the buffer. The systolic array, which performs the most time consuming part in the LZ77 algorithm occupies fewer resources than the buffer, so it is good idea to consider the buffer as a separate entity in the design.

We are integrating the developed compressor to a public key cipher module. According to the area resources, we can choose the better parameters for the LZ77 compressor for integrating both modules compression and encryption in a single chip.

References

1. Fowler, J. and Yagel, R., Lossless Compression of Volume Data, Symposium on Volume Visualization, pp. 43--50, Oct. 1994.
2. Ziv, J. and Lempel, A., A Universal Algorithm for Sequential Data Compression, IEEE Trans. Information Theory, vol. 23, pp. 337-343, May 1977.
3. Ziv, J. and Lempel, A., Compression of Individual Sequences via Variable-Rate Coding, IEEE Transactions on Information Theory, vol. 24, pp. 530-536, 1978.

4. Welch, T., A Technique for High-Performance Data Compression, IEEE Computer, vol. 17, pp. 8-19, June 1984.
5. Storer, J.A. and Syzmanski, T.G., Data Compression via Textual Substitution, Journal of the ACM, No 29, pp. 928-951, 1982.
6. Park, H., and Prasanna, V.K., Area Efficient VLSI Architecture for Huffman Coding, IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, Vol. 40, No. 9, pp. 568-575, September 1993.
7. Liu, L., et. al., CAM-Based VLSI Architectures for Dynamic Huffman Coding, IEEE Transactions on Consumer Electronics, pp. 282-288, August 1994.
8. Jung, B., and Burleson, W.P., Efficient VLSI for Lempel-Ziv Compression in Wireless Data Communication Networks, IEEE Transactions on Very Large Scale Integration Systems, Vol. 6, No. 3, pp. 475-483. September 1998.
9. Hwang, W. and Saxena, N., A Reliable LZ Data Compressor on Reconfigurable Coprocessors, IEEE Sym. On Field Programmable Custom Computing Machines, 200.
10. Hwang, S.A., and Wu, C.W., Unified VLSI Systolic Array Design for LZ Data Compression, IEEE Transactions on Very Large Scale Integration Systems, Vol. 9, No. 4, August 2001.
11. Canterbury Corpus. Available from http://corpus.canterbury.ac.nz