Design and Implementation of Lempel-Ziv Data Compression using FPGA

Gehad Mohey
Electronics and Communication Department,
El-Madina Higher Institute for Engineering and Technology, Giza, Egypt

Abdelhalim Zekry
Electronics and communications department, Faculty of Engineering,
Ain shams University, Egypt

Hatem Zakaria
Electrical Engineering Department,
Benha Faculty of Engineering,
Benha University, Egypt

ABSTRACT
When transmitting the data in digital communication, it is well desired that the transmitting data bits should be as minimal as possible, so many techniques are used to compress the data. In this paper, a Lempel-Ziv algorithm for data compression was implemented through VHDL coding. The Lempel-Ziv algorithm is one of the most widely used among lossless data compression algorithms for hardware implementation. The work in this paper is devoted to improve the compression rate, space-saving, and utilization of the Lempel-Ziv algorithm using a systolic array approach. The developed design is validated with VHDL simulations using Xilinx ISE 14.5 and synthesized on Virtex-6 FPGA chip. The results show that our design is efficient in providing high compression rates as well as improved utilization. The Throughput is increased by 50% and the design area is decreased by more than 23% with a high compression ratio compared to comparable previous designs.

Keywords
Data Compression, Lossless compression, VHDL, FPGA, Design, Utilization, CODEC, LZSS, LZ77, Systolic array design

1. INTRODUCTION
Computers can handle many kinds of information some of which require a huge amount of data and may take a long time to transfer across a network. For this reason, it is interesting to see if the information can somehow be rewritten in such a way that it takes up less space. There are two general classes of methods, those that do not change the information, so that the original file can be reconstructed exactly, and those that allow small changes in the data. Data compression is used in audio compression, video compression, image processing, data transfer, digital telecommunication network[1][2]. Compression methods in the first class are called lossless compression methods while those in the second class are called lossy compression methods. Lossy methods may sound risky since they will change the information, but for data like sound and images, small alterations do not usually matter. On the other hand, kinds of information like for example, a text that cannot tolerate any change so lossless compression methods must be used [3] [4]. Some of the lossless data compression techniques have been proposed and widely used [5][4], e.g., Huffman code[6], [7], run-length code [8], arithmetic code [9], and Lempel–Ziv (LZ) compression algorithms [10]. Among them, the LZ algorithms are the most popular scheme when no prior knowledge or statistical characteristics of the data being compressed are available. The idea behind the LZ algorithms is to find the longest match length in the buffer containing a recently received data string with the incoming string and represent it with the position and length of the longest match in the buffer. Since the repeated data is linked to an older one, a more concise representation is achieved and compression is done.

Hardware implementation is required for on-the-fly compression and decompression. Many hardware Implementation has been proposed in the past, either statistical or dictionary-based. On one hand, statistical lossless data compressors are more expensive than dictionary-based implementations, essentially in area requirements, although they provide better compression ratios [11]. On the other hand, three approaches are distinguished in the hardware implementation of dictionary-based methods: the microprocessor approach [12], CAM (Content Addressable Memory) approach [13], and the systolic array approach [14] [15] [16]. The first approach does not fully explore parallelism and is not attractive for real-time applications. The second one is very fast but it is costly in terms of hardware requirements and power consumption. The systolic array approach is not as fast as the CAM approach, but its hardware requirements are lower and testability is better. The CAM approach performs a string match by full parallel searching, while the systolic-array approach does it by pipeline. The main advantage of the Systolic array approach is that it can achieve a higher clock rate and it is easily implemented [14].

This paper is organized as follows: the related work is explained in Section II. Section III is addressed to LZ77 compression algorithm and its Improvements. Section IV describes the proposed based systolic array design. Section V contains the simulation and implementation results of our design. Finally, conclusions are given in Section VI.

2. RELATED WORK
There are two major compression categories of data, the lossy and lossless compression [9]. Since lossy compression allows a loss of accuracy to an appropriate level, it is typically used in multimedia applications where errors can be tolerated [10]. Lossless compression can compress and then recover data from compressed data without any information loss and it is used for applications where it is difficult to accept even a single bit difference between the original and reconstructed data [17]. LZ compression is a dictionary method based on replacing text substrings by previous occurrences thereof. The dictionary of LZ compression starts in some predetermined state but the contents change during the encoding process, based on the data that has already been encoded. LZ methods are popular for their speed and economy of memory, the two most famous algorithms of this family are called LZ77[10] and LZ78 [18]. One of the most popular versions of LZ77 is LZSS [19][20][21].

In [22], Marsh and Knapp presented a detailed analysis of how the size of the buffers in the LZ77 algorithm affected the throughput and compression ratio. By choosing a specific buffer size, the required area can be evaluated, the compression ratio,

and the throughput that achieved by the compressor. A prototype of the compressor was implemented in a Xilinx XC2V1000 FPGA device using a 512-byte search buffer and a 15-byte coding buffer. Based on post-layout simulations, architecture achieved an 11 Mbps throughput while occupying 90% of the FPGA resources.

In [16], Mohamed A. Abd El Ghany described a parallel algorithm for LZ based data compression by transforming a data-dependent algorithm to a data-independent regular algorithm. He used a control variable to indicate early completion to further improve the latency. The proposed implementation was area and speed efficient. The compression rate was increased by more than 40% and the design area is decreased by more than 30%. His compression rate was about 13Mbps.

3. LZ77 COMPRESSION ALGORITHM

The LZ77 algorithm was proposed by Jacob Ziv and Abraham Lempel in 1977, [10]. It is a dictionary-based algorithm for lossless data compression that can achieve an average compression ratio and is considered universal, such 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 a symbol's statistics. Since the buffer size (n) and match length (Ls) determine the compression efficiency, the relationship between n and Ls for optimal compression performance was briefly examined, [16].

The Principle of the LZ77 algorithm is accomplished by using a sliding window that moves along with the cursor. Replacing a symbol string by a pointer or position in a dictionary where such strings occur. The window can be divided into two parts, the part before the cursor, called the dictionary, and the part starting at the cursor, called the look-ahead buffer. The sizes of these two parts are input parameters to the compression algorithm. Data compression can be achieved by performing the following simple steps and loops executable:

- Find the longest match of a string starting at the cursor and completely contained in the look-ahead buffer to a string starting in the dictionary.
- Output a triple (I_p, L_{max}, S) containing the position I_p of the occurrence in the window, the length L_{max} of the match and, the next symbol S past the match.
- Move the cursor L_{max} + 1 symbols forward.

Note that LZSS variant doesn't include the next symbol in the triple output.

Consider an example with a window size of (n = 9) shown in Fig. 1 and look-ahead buffer size (L_s = 3). Assume that the content of the window be represented as X_i, i = 0, 1, ⋯, n–1 and that of the look-ahead buffer be Y_j, j = 0, 1, ⋯, L_s–1 (i.e., Y_j = X_{i+n-Ls}). According to the LZ algorithm, the content of the look-ahead buffer is compared with the dictionary content starting from X_0 to X_{n-Ls-1} to find the longest match length. If the best match in the window is found to start from position I_p and the match length is L_{max}. Then L_{max} symbols will represent by a codeword (I_p, L_{max}). The code word length L_c is given by:

$$L_c = \log_2(n - L_s) + \text{Log}_2(L_s) + 1 \text{ bits} \quad (1)$$

L_c is fixed. Assume w bits are needed to represent a symbol in the window, l = $\log_2(L_s)$ bits are required to represent L_{max}, and p = $\log_2(n - L_s)$ bits are needed to represent I_p. Then the compression ratio is $(l+p)/(L_{max}w)$, where $0 \le L_{max} \le L_s$. Then the compression ratio depends on the match situation.

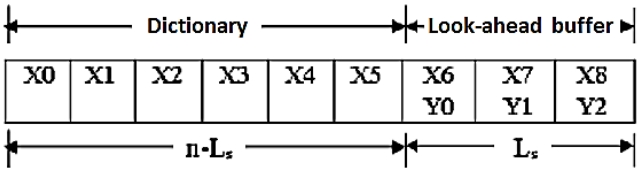


Fig.1. Window of the LZ compressor example.

The choice of window size and the code word design is crucial in achieving maximum compression. The LZ technique involves converting variable-length substrings into fixed-length code words that represent the pointer and the length of the match. Hence, the selection of values of n and L_s can greatly influence the compression efficiency of the LZ algorithm.

The LZ77 algorithm has a major advantage among the known lossless data compressor algorithms, it does not need previous knowledge or statistical characteristics of the symbols. This fact lets faster compression because a second pass over the data is not required as occurs in some statistical methods. For this reason, LZ77 is main concern for this paper to be implemented with a deep study of its competitive lossless data compressor.

Before describing the hardware architecture, an overview of the LZSS algorithm implementation that has been used will be presented [19]. It has minor differences from the original LZ77 [10]. The mandatory inclusion of the following non-matching symbol into every code word will cause situations in which the symbol is being explicitly coded despite the possibility of it being part of the subsequent match.

Example: In "abbca|caabb", the first match is a reference to "ca" (with the first non-matching symbol being "a") and then the next match is "bb" while it could have been "abb" if there were no necessity to explicitly code the first non-matching symbol. The popular modification by Storer and Szymanski (1982) removes this requirement. Their algorithm uses fixed-length code words consisting of offset (into the search buffer) and length (of the match) to denote references. Only symbols for which no match can be found or where the references would take up more space than the codes for the symbols are still explicitly coded.

In general, the maximum parallelism in an algorithm can be realized by studying the data dependencies in computations. As described in [14] dependence graph (DG) is a graph that shows the dependence of the computations that happen in an algorithm. In Fig. 2 the global DG of the LZ algorithm is depicted. In the DG, L (match length) and E (match signal) are propagated from cell to cell. X (the content of the window) and Y (the content of the look-ahead buffer) are broadcast horizontally and diagonally to all cells, respectively. The processor assignment can be done by a projection of the DG into the surface normal to the projection vector selected. After the processor assignment, the events are scheduled using a schedule vector.

4. SYSTOLIC ARRAY DESIGN FOR LZ DATA COMPRESSION

The hardware architectures of LZ data compression prove that systolic array compressors are better in testability and hardware cost (due to regularity and homogeneity) and it can achieve a higher clock rate (due to nearest-neighbor communication). This section explains the most recent designs of a systolic array architecture for efficient implementation of the LZ77 compression technique, [15].

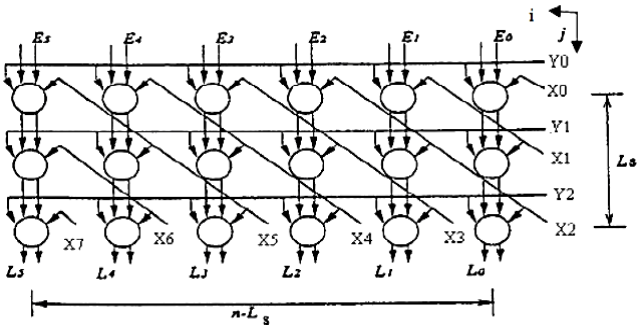


Fig. 2. Dependence graph of the LZ compression algorithm

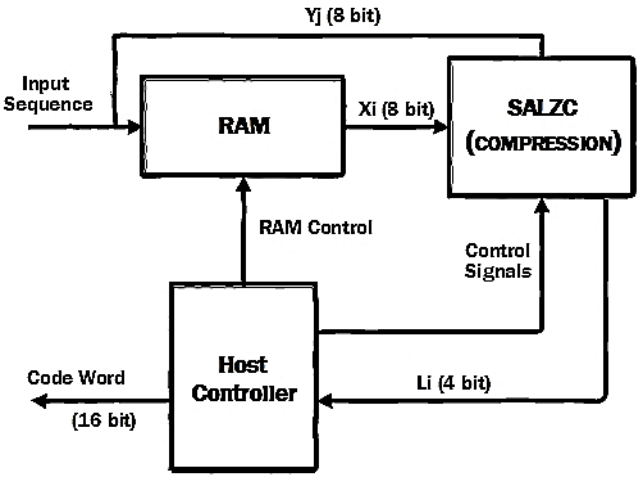


Fig.3. Lempel-Ziv compression chip

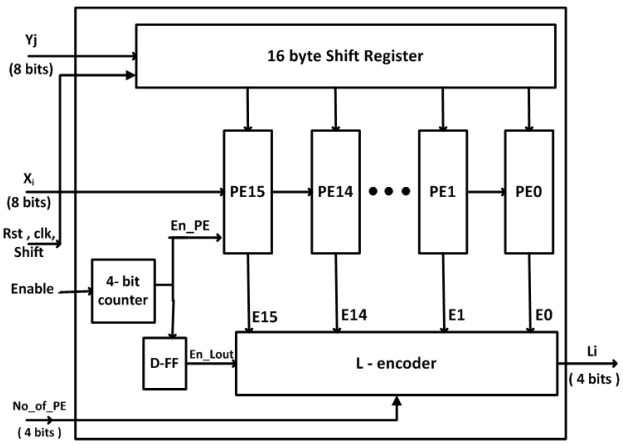


Fig. 4. SALZC module block diagram

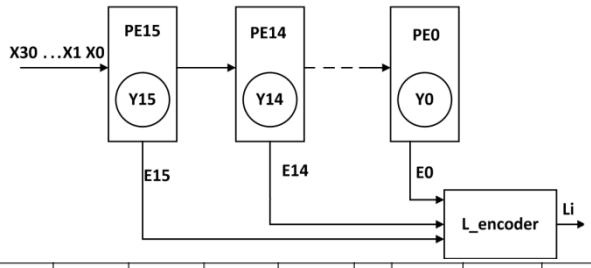
The compression design of LZ is shown in Fig. 3. The systolic array design architecture consists of three major components: the SALZC compressor module, the RAM block, and the host controller. SALZC module doesn't include block RAM. Thereby the dictionary size can be increased by directly replacing the block RAM with a larger one. Also, the host controller is not combined into the SALZC module, to be able to modify when the dictionary size is changed. In our implementation, the window size n is of length 1K, and the look-ahead buffer is of length Ls = 16.

4.1 SALZC Module

SALZC module contains 16 processor elements (PES), one L-encoder, 16 bytes shift register, and 4 bits counter. The block diagram of the SALZC module is depicted in Fig. 4.

From the DG shown in Fig. 2, all the nodes in a specific row are integrated into a single processor element (PE). This produces a linear array of length Ls; the layout is easy since the array is very regular. Only a single cell (PE) was hand – laid out. The other 15 PEs are just its copies. Since the array is systolic, routing also is simplified. The resulting array of Design-P and the space-time diagram are given in Fig. 5.

In Fig. 5 the architecture consists of 16 Ls processing elements used for comparison, and L-encoder used for matching length output. Thus, the look-ahead buffer symbols Yj that do not change during the encoding step remain in PEs. The Xi dictionary variable moves systolically from left to right, with 1 clock cycle delay. The processing element's match signal Ei moves into the L-encoder. The encoder's output Li is the matching longitude resulting from the i–1 comparison. After one clock cycle, the first Li will be obtained and each clock cycle will be obtained for the following ones. The Yj is preloaded in order to be processed before the encoding process and this will take Ls extra cycles. The time to preload new source symbols during the encoding process depends on how many source symbols were compressed in the preceding compression step, Lmax.



space time	PE15	PE14	PE13	PE12	....	PE1	PE0	Li
1	X15– Y15	X14– Y14	X13– Y13	X12– Y12	....	X1– Y1	X0– Y0	
2	X16– Y15	X15– Y14	X14– Y13	X13– Y12	....	X2– Y1	X1– Y0	L0
3	X17– Y15	X16– Y14	X15– Y13	X14– Y12	....	X3– Y1	X2– Y0	L1
...	X– Y15	X– Y14	X– Y13	X– Y12	....	X– Y1	X– Y0	...
16	X30– Y15	X29– Y14	X28– Y13	X27– Y12	....	X16– Y1	X15– Y0	L14
17								L15

Fig.5. Space-time diagram and an array of Design-P.

The functional block of the PE is shown in Fig. 6. The comparison of Yj and incoming Xi requires only one equality comparator. The Ei (match signal) result for the comparator propagates to the L-encoder. The L-encoder block diagram is depicted in Fig. 7. L-Encoder computes the match-length Li corresponding to position i according to Ei (match signals).

4.2 Host Controller

The Host controller includes match results block (MRB), code word generator, and end of processing block (EOPB), as shown in Fig. 8

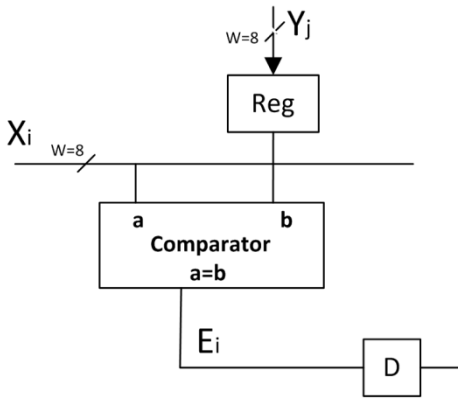


Fig. 6. Functional Block of a processing element.

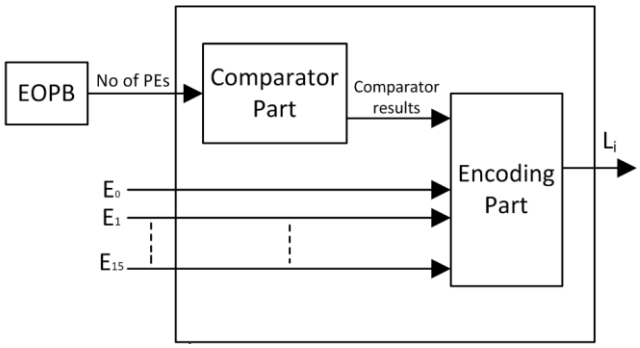


Fig. 7. L-encoder block diagram

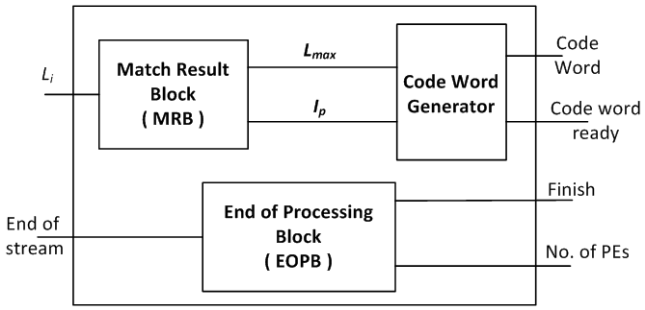


Fig.8. Host controller block diagram

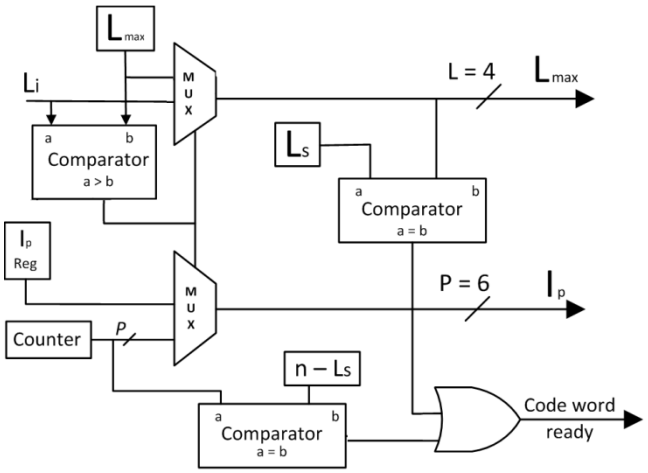


Fig.9. Match results block

From Fig. 5, it is clear that the L-encoder doesn't generate the maximum matching length. So, to determine Lmax among the serially generated Lis', a match results block (MRB) is needed, as shown in Fig. 9. The PEs also don't need to store their ids to record the Lis' (Ip) location. Since p =log2(n–Ls) bits are required to represent Ip, the position i associated with each Li requires only a p-bit counter, since Li is produced corresponds to its position MRB uses a comparator to compare the current Li input to the current longest Lmax match length stored in the register If the current input Li is greater than Lmax, then Li will be loaded into the register and the location counter information will also be loaded into another register that will be used to store the present Ip. Another comparator is used to check that the whole window is completely scanned. It compares the content of the position counter with n–Ls, whose output is used as the codeword ready signal. When i <(n–Ls), i.e., Li could be equal to Ls during the search process. The content in the Look-ahead buffer can be fully matched to the dictionary sub-set, so it is not always necessary to search the entire window. An additional comparator is used to decide whether Lmax is equal to Ls, and then the string-matching cycle is completed. So, encoding a new collection of data can be started straight away. This will reduce the average compression time. The number of clock cycles required to produce a code word is (n–Ls) +1 clock cycles, so each PE's utilization rate is (n–Ls) / [(n–Ls) +1], nearly equal to one. This result is consistent since the PE is busy once Li is determined until the time at which the code word is produced.

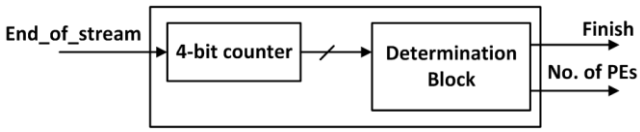


Fig.10. End of processing Block (EOPB)

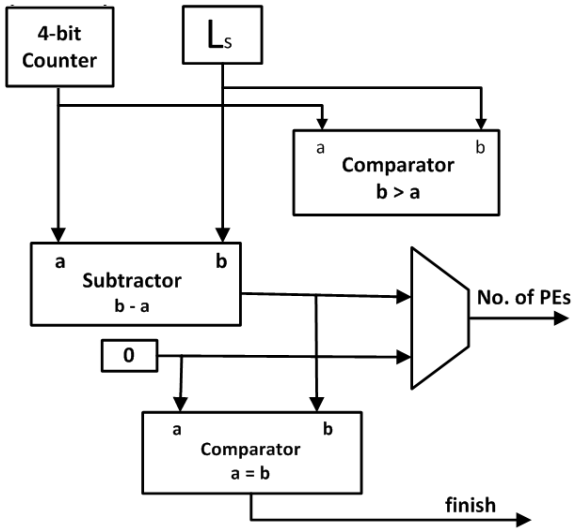


Fig.11. Determination Block (DB)

The end of processing block as shown in Fig. 10 includes a 4-bit counter and Determination Block (DB). This counter is needed to successfully handle the last part of the data stream. The end of stream signal does not mean the end of the compression operation, but once the end of stream signal is generated using the 4-bit counter it is used to trigger the encoding process of the unprocessed data in the look-ahead buffer. After receiving the enable signal the counter will count the number of shift operations. DB determines the number of process elements that will operate during the encoding step according to the counter output and generates the end signal after the compression operation is complete.

DB is shown in Fig 11. Without the DB the last part will be compressed incorrectly. The number of PEs in the forward buffer should be equal to the number of unprocessed data.

Comparator and Subtractor are the principal components of DB. If the counter output (the number of data processed in the look-ahead buffer) is less than the number of PEs, they can be subtracted by the Subtractor. The number of PEs is created which will operate during the encoding stage. If the counter output is equal to the PES number, it means the entire look-ahead buffer data is processed. Hence the end signal (finish) will generate.

4.3 Block RAM

Block RAM is used as a data buffer. The dictionary size is a parameter for a wide range of applications, from text compression to lossless images compression.

The block RAM is used as first-in-first-out (FIFO), so two counters need to be used, as illustrated in Fig. 12. The first one is to generate a write address. At first, it is loaded by the look-ahead buffer's first address, then it counts to initialize the look-ahead, buffer. Afterward, it will point to where an input symbol should be inserted. The second one is to generate the address for reading. It will point to the FIFO's first location (equal to the address written + 1). Upon reaching the maximum value one of two counters. In the next step, it'll immediately go down to 0.

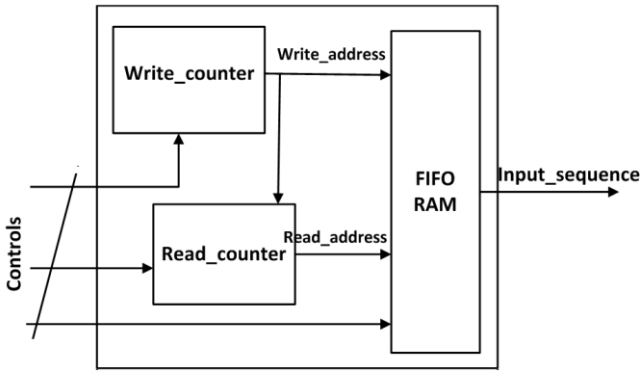


Fig.12. First-in-first-out (FIFO)

5. RESULTS

5.1 Software Simulations

The RTL architecture of SALZC module depicted in Fig. 4 is VHDL modeled with its simulation result as shown in Fig. 13. The SALZC receives a sequence of 16 bytes of data from a text vector file. Thus, the first 16-bytes of data stored in Yj then it reads Xi and then it compared Yj with Xi and the result is in Li and Y0-out since Li = 1111 and this is due to the first 16-byte of Xi equal the first 16-bytes of Yj and Y0-out = 01110011 and this is due to the first byte of file = 01110011.

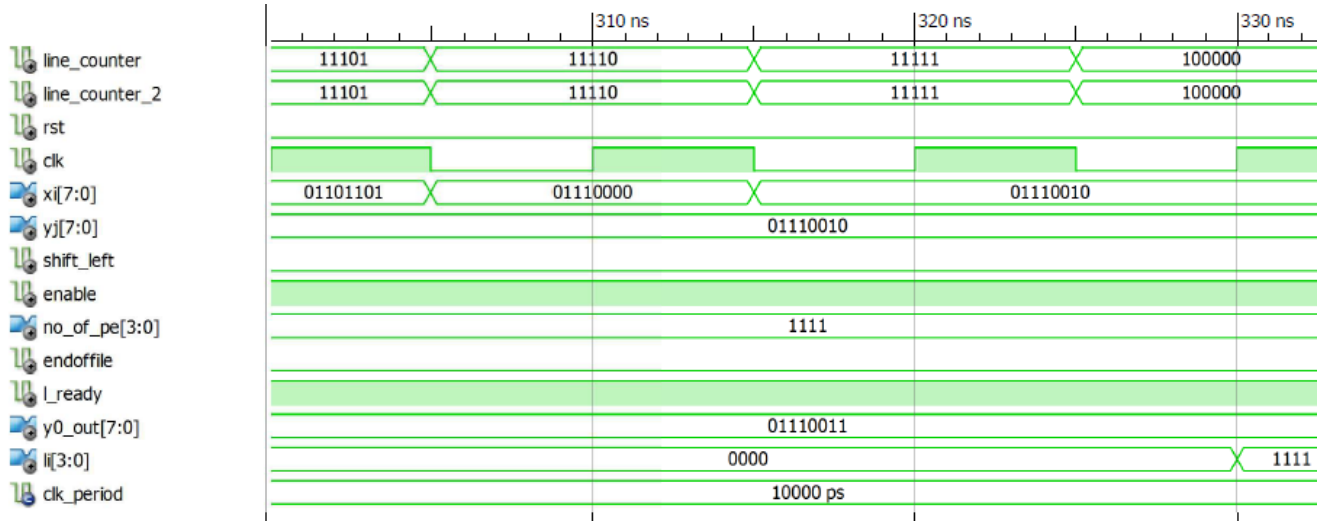


Fig.13. Simulation results of SALZC module.

The simulation result of the Host controller is shown in Fig. 14. The code will output the code word due to the received signal from SALZC since if there is no match it will output codeword that contains M0 if there is a match it will output the code word that contains Length of the match and its pointer. The first bit in the codeword specifies that if there is a match or not.

The code also will do shift if en-shift = 1 or if load =1 since en-shift is a control signal to do 16 shifts initially then if load =1 it will load a new byte. The Host controller output also depends on L-ready, which shows that the match is ready or not. Li shows the length of the match and according to this length, the code will do shift Ready = 1 then shift-left = 1 for 6 clock cycle then shift left return to zero waiting for a new condition of Li or load if there is no match.

After verifying the VHDL code of all the component Window, SALZC and Host controller, the match-length of comparison and the first byte stored in the first PE is fed to Host controller then it decides if it was a match-length then it compares it with the maximum length stored previously then it outputs the codeword that consists of (16-bits) contain match-length of compression and the pointer of this length, then it does several shifts equal to the match-length and load a new number of byte to the shift register and compare again. If it wasn't a successful comparison it output the first byte that was stored in the first PE and it does one shift (load one new byte) and do the comparison again. If it has a match-length after the comparison, SALZC module has a signal that shows that the code has a match-length as shown in Fig. 16 (Q_ready) signal = 1 at the time the output has a length and pointer and the first bit of the codeword equal one this is another verify for the output, but if the output has zero length it will out a signal (load) = 1 that verify there is no correct comparison.

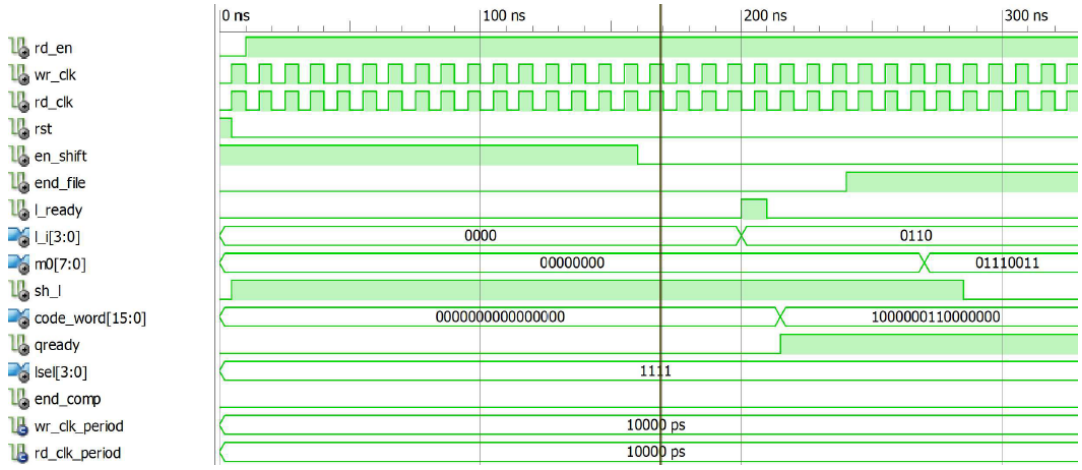


Fig.14. The Host controller simulation result.

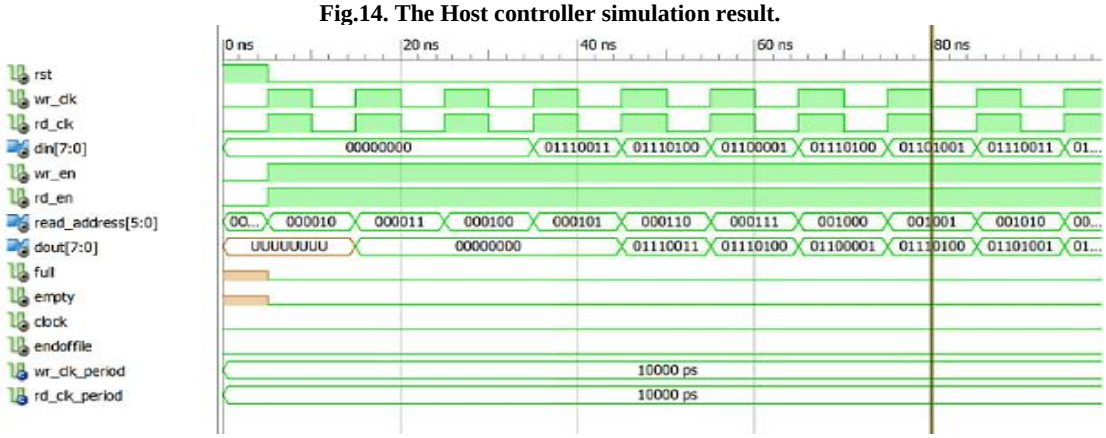


Fig.15. Window Simulation Result

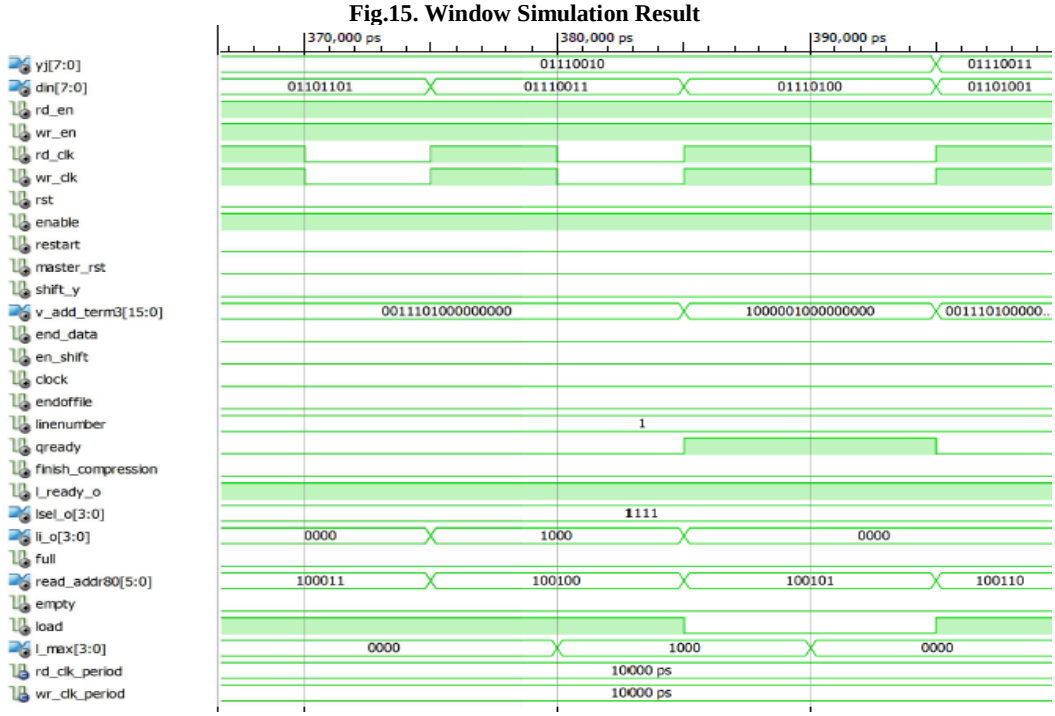


Fig.16. The LZ compression chip simulation result.

5.2 Performance Parameters of the Proposed Design

In this section, the achieved design is presented lossless compression efficiency. The implementation of our design is carried out using Xilinx Virtex-6 FPGA, for n = 1k, Ls = 16, and w = 8. FPGA utilization summary is shown in Fig. 17.

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	373	301440	0%
Number of Slice LUTs	262	150720	0%
Number of fully used LUT-FF pairs	117	518	22%
Number of bonded IOBs	65	600	10%
Number of Block RAM/FIFO	1	416	0%
Number of BUFG/BUFGCTRL/BUFHCEs	3	176	1%

Fig.17. Implementation result of the proposed Design

The chip FPGA verification has been done by using the chips-cope pro analyzer Xilinx tool to verify that the SALZC works successfully as demonstrated in Fig.15.

A compressor's compression rate is defined as the number of input bits that can be compressed in just one second. The compression rate Rc can be estimated as follows:

$$Rc = clk \times [(Ls*W)/(n–Ls+1)] \quad (2)$$

where clk is the clock rate. Note that only estimated Rc can be obtained, as it depends on the input data. The number of needed words that will be compressed can't be predicted exactly (Ls at most) and how many clock cycles will be required (n–Ls+1) at most) for every compression step. In our implementation, window size (n) 1K is used, Ls = 16, w = 8, and clk = 175.408, The Rc is about 22.25 Mbps. Our module does space-saving about 55% and on average compression rate up to 25.75Mbps. Saving percentage in our FPGA implementation is 55% and the compression ratio is 67.8%. The total on-chip power is 3.422W.

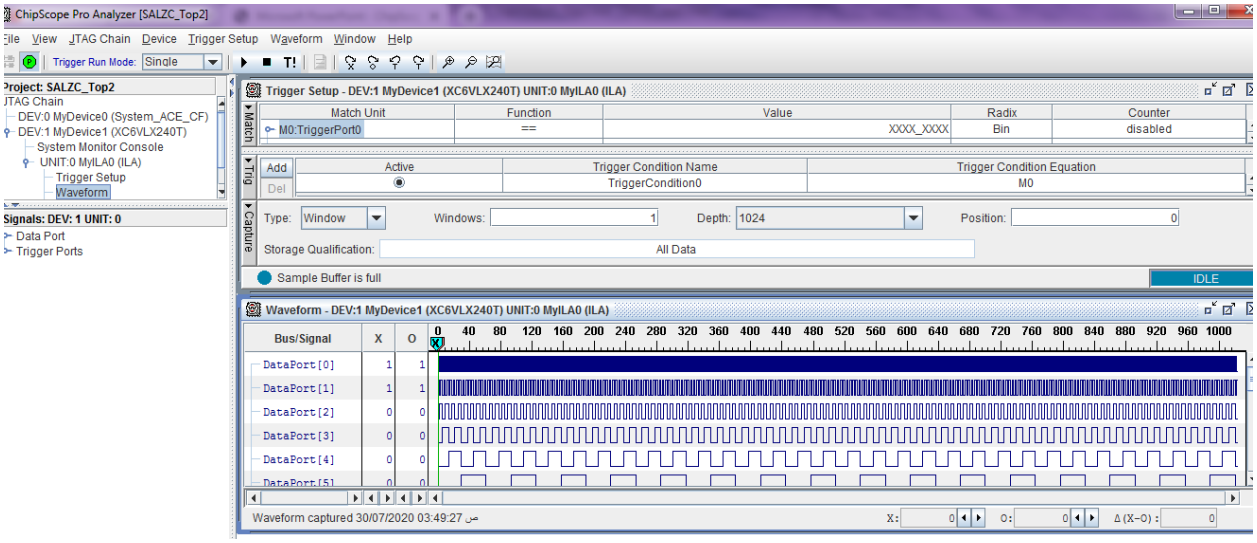


Fig.18. waveform window of analyzed Results using Chipscope

6. CONCLUSIONS

In this paper, the design and implementation of lossless data compression was described using the LZ algorithm. Xilinx ISE 14.5 tool is used. The programming is done in VHDL language and the whole algorithm is described in that language. Our systolic array LZ compression (SALZC) module provides space-saving about 55% and on average compression rate up to 25.75 Mbps. Compared to the results in [16] the throughput is increased by 50% and the design area is decreased by more than 23% that provides an excellent platform for Real-time compression applications. As future work, one may modify the host controller since it can be used for other string-matching based LZ algorithms, such as LZ78 and LZW.

7. REFERENCES

[1] J. Latif and Z. Ali, ‘An Efficient Data Compression Algorithm for Real-Time Monitoring Applications in Healthcare’, pp. 71–75, 2020.

[2] J. Uthayakumar and T. Vengattaraman, ‘Performance Evaluation of Lossless Compression Techniques : An Application of Satellite Images’, *2018 Second Int. Conf. Electron. Commun. Aerosp. Technol.*, no. Iceca, pp. 750–754, 2018.

[3] H. D. Kotha, M. Tummanapally, and V. K. Upadhyay, ‘Review on Lossless Compression Techniques’, *J. Phys. Conf. Ser.*, vol. 1228, no. 1, 2019, doi: 10.1088/1742-6596/1228/1/012007.

[4] A. Gopinath, ‘Comparison of Lossless Data Compression Techniques’, pp. 628–633, 2020.

[5] J. Uthayakumar, T. Vengattaraman, and P. Dhavachelvan, ‘A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications’, *J. King Saud Univ. - Comput. Inf. Sci.*, 2018, doi: 10.1016/j.jksuci.2018.05.006.

[6] A. Moffat, ‘Huffman Coding’, vol. 52, no. 4, 2019.

[7] S. T. Klein, S. Saadia, and D. Shapira, ‘Forward Looking Huffman Coding’, 2020.

[8] M. Pandey, S. Shrivastava, and S. Pandey, ‘An Enhanced Data Compression Algorithm’, pp. 1–4, 2020, doi: 10.1109/ic-ETITE47903.2020.223.

[9] C. W. Huang and J. J. Ding, ‘Efficient EEG Signal Compression Algorithm with Long Length Improved Adaptive Arithmetic Coding and Advanced Division and Encoding Techniques’, *Int. Conf. Digit. Signal Process. DSP*, vol. 2018-Novem, no. 1, pp. 1–5, 2019, doi: 10.1109/ICDSP.2018.8631886.

[10] J. Ziv and A. Lempel, ‘A Universal Algorithm for Sequential Data Compression’, *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, 1977, doi: 10.1109/TIT.1977.1055714.

[11] A. Gupta, A. Bansal, and V. Khanduja, 'Modern lossless compression techniques: Review, comparison and analysis', *Proc. 2017 2nd IEEE Int. Conf. Electr. Comput. Commun. Technol. ICECCT 2017*, vol. XI, no. Xii, pp. 44–51, 2017, doi: 10.1109/ICECCT.2017.8117850.

[12] U. K. H, 'Design and Implementation of Lossless Data Compression Coprocessor using FPGA', vol. 4, no. 05, pp. 818–822, 2015.

[13] K. Pagiamtzis and A. Sheikholeslami, 'Content-addressable memory (CAM) circuits and architectures: A tutorial and survey', *IEEE J. Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2006, doi: 10.1109/JSSC.2005.864128.

[14] S. A. Hwang and C. W. Wu, 'Unified VLSI systolic array design for LZ data compression', *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 9, no. 4, pp. 489–499, 2001, doi: 10.1109/92.931226.

[15] N. Ranganathan and S. Henriques, 'A systolic chip for LZ based data compression', *Proc. IEEE Int. Conf. VLSI Des.*, pp. 310–311, 1991, doi: 10.1109/ISVD.1991.185144.

[16] M. A. Abd El Ghany, A. E. Salama, and A. H. Khalil, 'Design and implementation of FPGA- Based systolic array for LZ data compression', *Proc. - IEEE Int. Symp. Circuits Syst.*, pp. 3691–3695, 2007, doi: 10.5772/8872.

[17] B. Jaysree, D. Harshith, P. Deekshith, and I. B. Mahapatra, 'Design and implementation of a reconfigurable hardware for data compression techniques : a Survey', vol. 10, no. 03, pp. 33–40, 2020, doi: 10.9790/9622-1003023340.

[18] J. Ziv and A. Lempel, 'Compression of Individual Sequences via Variable-Rate Coding', *IEEE Trans. Inf. Theory*, vol. 24, no. 5, pp. 530–536, 1978, doi: 10.1109/TIT.1978.1055934.

[19] J. A. Storer and T. G. Szymanski, 'Data Compression via Textual Substitution', *J. ACM*, vol. 29, no. 4, pp. 928–951, 1982, doi: 10.1145/322344.322346.

[20] S. Belu and D. Coltuc, 'RoLZ-The reduced offset LZ data compression algorithm', *ISSCS 2019 - Int. Symp. Signals, Circuits Syst.*, pp. 1–4, 2019, doi: 10.1109/ISSCS.2019.8801741.

[21] G. Wang, H. U. A. Peng, and Y. Tang, 'Repair and Restoration of Corrupted LZSS Files', *IEEE Access*, vol. 7, pp. 9558–9565, 2019, doi: 10.1109/ACCESS.2019.2891764.

[22] E. R. Marsh and B. R. Knapp, 'On the design and implementation of an instrumented grinding testbed', *Sens. Rev.*, vol. 25, no. 1996, pp. 155–161, 2005, doi: 10.1108/02602280510585754.