Design and Implementation of FPGA-based Systolic Array for LZ Data Compression

Mohamed A. Abd El ghany
Electronics Dept.
German University in Cairo
Cairo, Egypt
E-mail: mohamed.abdel-ghany@guc.edu.eg

Aly E. Salama and Ahmed H. Khalil
Electronics and Communication Dept.
Cairo University
Cairo, Egypt
E-mails: alisalama@cu.edu.eg, eoip@gega.net

Abstract— Hardware Implementation of Data compression algorithms is receiving increasing attention due to exponentially expanding network traffic and digital data storage usage. Among lossless data compression algorithms for hardware implementation, Lempel–Ziv algorithm is one of the most widely used. The main objective of this paper is to enhance the efficiency of systolic-array approach for implementation of Lempel–Ziv algorithm. The proposed implementation is area and speed efficient. The compression rate is increased by more than 40% and the design area is decreased by more than 30%. The effect of the selected buffer’s size on the compression ratio is analyzed. An FPGA implementation of the proposed design is carried out. It verifies that data can be compressed and decompressed on-the-fly.

Index Terms — CAM, data compression, LZ77, LZ78, LZSS, LZW, systolic array, FPGA.

I. INTRODUCTION

Data compression is becoming an essential component of high speed data communications and storage. Lossless data compression is the process of encoding ("compressing") a body of data into a smaller body of data which can at a later time be uniquely decoded ("decompressed") back to the original data. In lossy compression, the decompressed data are some approximation of the original data.

Many lossless data compression techniques have been proposed in the past and widely used, e.g., Huffman code [1]-[2], arithmetic code [3]-[4], run-length code [5], and Lempel–Ziv (LZ) compression algorithms [6]-[9]. 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 principle of the LZ algorithms is to find the longest match length in the buffer containing 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, more concise representation is achieved and compression is done. To fulfill real-time requirements, several works on hardware realization of LZ77 or its variants have been presented in the literature. Some hardware architectures, including content addressable memory (CAM) [10], Systolic array [11]-[13], and embedded processor [14], have been proposed in the past. The microprocessor approach is not attractive for real-time applications, since it does not fully explore hardware parallelism [13]. CAM has been considered the fastest architecture among all proposed hardware solutions for searching for a given string, as required in LZ77. A CAM-based LZ77 data compressor can process one input symbol per clock cycle, no matter what the buffer size and string length are. Thereby it achieves optimal speed for compression. However, CAM's are used high hardware complexity and high power consumption. The CAM approach performs string match by full parallel searching, while the systolic-array approach does it by pipelining. As compared with CAM-based designs, systolic–array compressors are slower, but better in hardware cost and testability [13]. A first systolic–array design for LZ-based data compression was proposed in [11], in which thousands of processor elements (PEs) are required for a usable scale. High speed designs were then reported later [12]-[13], requiring only tens of PEs. In this paper, we present an efficient systolic array design which is high speed and area efficient.

The organization of this paper is as follows: In Section II, we briefly explain the LZ compression algorithm. The results and comments about some software simulations are discussed. Then we show the dependence graph (DG) to investigate the data dependency of every computation step in the algorithm. Section III, we show the most recent systolic array architecture and propose an area and speed efficient architecture. In Section IV, the proposed systolic array structure is compared with the most recent structures [13] in terms of area, and latency. An FPGA implementation for our architecture showing the real time operations is demonstrated in Section V. Finally, conclusions are given in section VI.
II. LEMPEL-ZIV CODING ALGORITHM

The LZ77 algorithm was proposed by Ziv and Lempel in [15]. Since the buffer size (n) and match length (Ls) determine the compression efficiency, the relationship between n and Ls for optimal compression performance is briefly examined. Then we investigate the data dependency of every computation step in the LZ compression algorithm.

A. The compression algorithm:

The LZ77 algorithm and its variants use a sliding window that moves along with the cursor. 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 compression algorithm. The basic algorithm is very simple, and loops executing the following steps:

1. 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.
2. Output a triple (Ip, Lmax, S) containing the position Ip of the occurrence in the window, the length Lmax of the match and the next symbol S past the match.
3. Move the cursor Lmax + 1 symbols forward.

Let us consider an example with window size of (n=9) and look-ahead buffer size (Ls=3) shown in Figure 1.

Let the content of the window be denoted as Xr, i = 0,1,...,n-1 and that of the look-ahead buffer be Yj, j = 0,1,...,Ls-1 (i.e., Yj = Xr+n-Ls). According to LZ algorithm, the content of look-ahead buffer is compared with the dictionary content starting from X0 to Xn-Ls to find the longest match length. If the best match in the window is found to start from position Ip and the match length is Lmax. Then Lmax symbols will represent by a codeword (Ip, Lmax). The codeword length is:

\[ Lc = 1 + \lceil \log_2 (n-Ls) \rceil + \lceil \log_2 Ls \rceil \]  
\[ \text{(1)} \]

Lc is fixed. Assume w bits are required to represent a symbol in the window, 1 = \lceil \log_2 Ls \rceil bits are required to represent Lmax, and p = \lceil \log_2 (n-Ls) \rceil bits are required to represent Ip. Then the compression ratio is \( (l + p) / (Lmax \times w) \), where 0 \leq Lmax \leq Ls. Hence the compression ratio depends on the match situation.

The codeword design and the choice of widow size are crucial in achieving maximum compression. The LZ technique involves the converting of variable length substrings into fixed length codewords that represent the pointer and the length of match. Hence, selection of values of n and Ls can greatly influence the compression efficiency of the LZ algorithm.

B. Compression Algorithm Paramters Selection

Simulation for the performance of the LZSS algorithm for different buffers sizes is performed using the Calgary corpus and the Silesia Corpus [16], as shown in Figure 2 and Figure 3.

From Figure 2 and Figure 3, Improvement of the compression ratio can be achieved when n is greater than 1024. The above improvement can be obtained only when Ls = 8, 16, or 32. Based on the results, the best Ls for a good compression ratio is 2^4. Increasing Ls beyond that will require a much faster growing n (as well as hardware cost and computation time), with saturating or even worsening compression effect. The reason is that repeating patterns tend to be short, and that increasing Ls and n also increases the codeword length \( (l + p) \). To achieve a good performance for different data formats, Ls may range from 8 to 32, while n may be from 1k to 8 k. The simulation results verify the results proposed in [17].

C. Dependency graph:

A dependence graph (DG) is a graph that shows the dependence of the computations that occur in an algorithm. The DG of the LZ algorithm can be obtained as shown in Figure 4. In the DG, L (match length) and E (match signal) are propagated from cell to cell. X (content of the window) and Y (content of the look-ahead buffer) are broadcast horizontally and diagonally to all cells, respectively. The DG shown in Figure 4 is called a global DG, because it contains global signals. The global DG can be transformed into a localized DG, by propagating the input data Y and X from cell to cell instead of broadcasting them. Processor assignment can be done by projection of the DG onto the surface normal to the projection vector selected. After
processor assignment, we schedule the events using a schedule vector.

![Dependence graph of the LZ compression algorithm](image)

**III. SYSTOLIC ARRAY ARCHITECTURE**

A. The Interleaved Design:

From the dependence graph shown in Figure 4, the interleaved design is obtained by projecting all the nodes in a particular row to a single processor element. This design was first proposed in [13]. The space-time diagram and the resulting array of the interleaved design are given in Figure 5. The Y \textsubscript{s} which are needed to be accessed in parallel do not change during the encoding step. The special buffer is needed to generate the interleaving symbols of X \textsubscript{i} and X \textsubscript{i+[(n-L\textsubscript{S})/2]} as shown in Figure 6. The first L \textsubscript{i} will be obtained after L \textsubscript{S} clock cycles from the leftmost PE and subsequent ones will be obtained every clock cycle.

![Space-time diagram and resulting array of interleaved design](image)

Before the encoding process, we preload the Y \textsubscript{s} which take L \textsubscript{S} extra cycles. During the encoding process, the time to preload new source symbols depend on how many source symbols were compressed in the previous compression step, L\textsubscript{max}. The block diagram of the processor element is shown in Figure 7.

![Buffer design for generating the interleaving symbols](image)

![Block diagram of interleaved design PE](image)

Match results block MRB is needed to determine L\textsubscript{max} among the serially produced L\textsubscript{s}. MRB is shown in Figure 8. The PEs need not store their ids to record the position of the L\textsubscript{s}. A special counter is needed to generate the sequence which interleaves the position of the first half of L\textsubscript{s} and the position for the second half as shown in Figure 9. The compression time of the interleaved design is n clock cycles.

![Match results block](image)

![Special position counter in MRB for interleaved Design](image)

B. Proposed Design (Design- P)

From the dependence graph shown in Figure 4, we project all the nodes in a particular row to a single processor element. This produces an array of length L\textsubscript{s}. The space-time diagram and the resulting array of Design-P are given in Figure 10, where D represents a unit delay on the signal line between two processing elements.

As shown in Figure 10, the architecture consists of L\textsubscript{s} processing elements which are used for the comparison and L-encoder which is used to produce the matching length. Consequently, the look-ahead buffer symbols Y \textsubscript{s} which do not change during the encoding step, stay in PEs. The dictionary element X \textsubscript{i} moves systolically from left to right with a delay of 1 clock cycle. The match signal E \textsubscript{i} of the processing element moves to the L-encoder. The output L \textsubscript{i} of the encoder is the matching length resulting from the comparisons at step i-1. The first L \textsubscript{i} will be obtained after one clock cycle and the subsequent ones will be obtained every clock cycles. Before the encoding process, we preload the Y \textsubscript{s} to be processed and this takes L\textsubscript{S} extra cycles. During the encoding process, the time to preload new source symbols depends upon how many source symbols were compressed in the previous compression step, L\textsubscript{max}.

The functional block of the PE is shown in Figure 11. Only one equality comparator is needed for comparing Y \textsubscript{i} and incoming X \textsubscript{i}. The comparator result E \textsubscript{i} (match signal) propagates to L-encoder. The block diagram of L-encoder is shown in Figure 12. According to E \textsubscript{s} (match signals), L-encoder computes the match length L \textsubscript{i} corresponding to position i.

![Functional block of the PE](image)

![Block diagram of L-encoder](image)
length is not produced by the L-encoder. So, we need a ready signal. During the searching process, L can be fully matched to a subset of the dictionary, and hence therefore, we can immediately start encoding a new set of data. This will reduce the average compression time. The number of clock cycles needed to produce a codeword is \((n-L_s)/n\) clock cycles, so the utilization rate of each PE is 1, which almost equals to one. This result is consistent since the PE is busy once \(L_i\) is determined until the time at which the codeword is produced.

### Table 1 The comparison between Design-P and Design-I.

<table>
<thead>
<tr>
<th></th>
<th>Interleaved Design</th>
<th>Design-P</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEs</td>
<td>((n-L_s)/n)</td>
<td>((n-\text{ suffix size})/(n-L_s+1))</td>
</tr>
<tr>
<td>Utilization</td>
<td>(L_s)</td>
<td>((n-\text{ suffix size}))</td>
</tr>
<tr>
<td>Latency</td>
<td>(n)</td>
<td>(n-L_s+1)</td>
</tr>
<tr>
<td>DFFs per PE</td>
<td>(2n+1)</td>
<td>(2n+1)</td>
</tr>
<tr>
<td>Comparator per PE</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Counter per MRB</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Multiplier per MRB</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Multiplier per buffer</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Equality comparator per MRB</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### IV. The comparison between the Design-P and Interleaved Design (Design-I)

The comparison of two designs is given in Table 1. Our Design (Design-P), which has the maximum utilization rate, minimum latency time (high compression speed) and minimum area, is the best array architecture for LZSS.

Parallel compression can be achieved by using an appropriate number of Design-P modules. For example, we can use two modules of Design-P, as shown in Figure 14. The input sequence of the first module (Xi) is obtained from the first position of Buffer. The input sequence of the second module (Xi+((n−Ls)/2)) is obtained starting at ((n−Ls)/2) position in the Dictionary. Note that the MRB now needs to determine \(L_{\text{max}}\) among \(L_I, L_{II}\) that are produced at the same time. So we need to modify MRB. The speed is two times the Design-P array.

### V. FPGA Implementation

In this section, we present the implementation methodology of the design utilizing FPGA. As shown in Figure 15, the architecture consists of 3 major components: systolic-array LZ component SALZC, block RAM, and host controller. In our implementation, the length of window size \(n\) is 1K, and the length of look-ahead buffer \(L_s\) is 16. SALZ component contains 16 PEs and implements the most cost effective array architecture (Design-P). Full − custom
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. Block RAM that is used as the data buffer (Dictionary) is not included in SALZ component. Thereby we can increase the dictionary size by directly replacing the block RAM with a larger one. The dictionary size is a parameter to cover a broad range of applications, from text compression to lossless image compression.

The implementation of our architecture (Design-P) and interleaved design (Design-i) are carried out using XILINX (Spartan II XC200) FPGA, for \( n = 1 \text{ k}, L_s = 16, w = 8 \). The implementation results are shown in Table 2.

<table>
<thead>
<tr>
<th>Number of Slices</th>
<th>Number of Slice Flip Flops</th>
<th>Number of 4-input LUTs</th>
<th>Number of BRAMs</th>
<th>Maximum Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design-P</td>
<td>330</td>
<td>13%</td>
<td>408</td>
<td>8%</td>
</tr>
<tr>
<td>Design-i</td>
<td>472</td>
<td>20%</td>
<td>511</td>
<td>10%</td>
</tr>
</tbody>
</table>

The compression rate of a compressor is defined as the number of input bits which can be compressed in one second. The compression rate \( (R_c) \) can be estimated as follows:

\[
R_c = \text{clk} \times \left( \frac{\text{Ls} \times \text{W}}{(n-L_s+1)} \right)
\]  

Where \( \text{clk} \) is the clock rate. Note that only estimated \( R_c \) can be obtained, since it depends on the input data. We can not predict exactly how many words will be compressed \( (L_s \) at most) and how many clock cycles will be required \( (n - L_s + 1) \) at most) for every compression step. In our implementation, if the window size \( (n) \) is 1k, \( L_s = 16, w = 8 \), and \( \text{clk} = 105 \text{ MHz} \), the \( R_c \) is about 13 M bit per second.

If we want to use the parallel scheme to increase the compression rate, we can directly modify the host controller and connect an appropriate number of LZ compressor components in parallel, as shown in Figure 16. Note that the MRB now needs to determine \( L_{\text{max}} \) among \( L_t, L_H \) and \( L_{\text{THR}} \), e.g., ten components could be implemented in one chip of large size to achieve a compression rate about 130 M bits per second. Moreover, by modifying the host controller and including, e.g., dictionaries, our chip can be used for other string-matching based LZ algorithms, such as LZ78 and LZW. Our architecture is flexible.

**VI. CONCLUSIONS**

In this paper, we described a parallel algorithm for LZ based data compression by transforming a data-dependent algorithm to a data-independent regular algorithm. We propose our structure for LZ compression. To further improve the latency, we used a control variable to indicate early completion. The proposed implementation is area and speed efficient. The compression rate is increased by more than 40% and the design area is decreased by more than 30%. The design can be integrated into real-time systems so that data can be compressed and decompressed on-the-fly.

**REFERENCES**


