An Efficient Implementation of LZW Compression in the FPGA

Xin Zhou, Yasuaki Ito and Koji Nakano

Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527 JAPAN
{zhou,yasuaki,nakano}@cs.hiroshima-u.ac.jp

Abstract. The main contribution of this paper is to present a new hardware architecture for accelerating LZW compression using an FPGA. In the proposed architecture, we efficiently use dual-port block RAMs embedded in the FPGA to implement a hash table that is used as a dictionary. Using independent two ports of the block RAM, reading and writing operations for the hash table are performed simultaneously. Additionally, we can read eight values in the hash table in one clock cycle by partitioning the hash table into eight tables. Since the proposed hardware implementation of LZW compression is compactly designed, we have succeeded in implementing 24 identical circuits in an FPGA, where the clock frequency of FPGA is 163.35MHz. Our implementation of 24 proposed circuits attains a speed up factor of 23.51 times faster than a sequential LZW compression on a single CPU.

Keywords: LZW compression, Hardware algorithm, FPGA, Block RAM

1 Introduction

Data compression is one of the most important tasks in the area of computer engineering. It is always used to improve the efficiency of data transmission and save the storage of data. In this paper, we focus on LZW compression [11]. LZW compression is included in TIFF standard [1], which is widely used in the area of commercial digital printing. The LZW compression algorithm converts an input string of characters into a series of codes using a dictionary that maps strings into codes. Since dictionary tables are created by reading input data one by one, LZW compression is hard to parallelize. The main goal of this paper is to develop an efficient hardware architecture of LZW compression and implement it in an FPGA (Field Programmable Gate Array).

An FPGA is an integrated circuit designed to be configured by a designer after manufacturing. It contains an array of programmable logic blocks, and the reconfigurable interconnects allow the blocks to be inter-wired in different configurations. Since any logic circuits can be embedded in an FPGA, it can be used for general-purpose parallel computing. Recent FPGAs have embedded block RAMs. A block RAM is an embedded dual-port memory supporting synchronized read and write operations, and can be configured as a 36k-bit or two

18k-bit dual port RAMs [13]. Since FPGA chips maintain relatively low price and its programmable features, it is suitable for a hardware implementation of image processing method to a great extent.

Numerous implementations of variety of LZW compression on FPGAs or VLSIs [3, 5, 6, 8, 9], GPUs [2, 10], multiprocessor [4] and cluster systems [7] have been proposed to accelerate the computation. However, as far as we know, there is no hardware implementation of the original LZW compression algorithm since it is not easy to implement it.

The main contribution of this paper is to present an efficient hardware architecture for LZW compression algorithm and to implement it in an FPGA. In the proposed architecture, we efficiently use dual-port block RAMs embedded in the FPGA to implement a hash table that is used as the dictionary. According to the experimental results, the throughput of the proposed circuit is 118.73MBytes/s when the compression ratio (original image size : compressed image size) is 1.43:1. On the other hand, the throughput is 86.79MBytes/s when the compression ratio is 36.72:1. Furthermore, since the proposed circuit of LZW compression uses a few FPGA resources, we have succeeded in implementing 24 identical circuits in an FPGA, where the frequency is 163.35MHz and each circuit has independent input/output ports that work in parallel. Hence, the implementation of 24 proposed circuits attains a speed up factor that surpasses 23.51 times over a sequential implementation on a CPU.

2 LZW Compression Algorithm

The main purpose of this section is to review LZW compression algorithm. The LZW (Lempei-Ziv-Welch) [11] lossless data compression algorithm converts an input string of characters into a series of codes using a dictionary table that maps strings into codes. If the input is an image, characters may be 8-bit unsigned integers. It reads characters in an input image string one by one and adds an entry in a dictionary table. At the same time, it writes an output series of codes by looking up the dictionary table. Let $X = x_0x_1 \cdots x_{n-1}$ be an input string of characters and $Y = y_0y_1 \cdots y_{m-1}$ be an output string of codes. For simplicity, we assume that an input string is a string of 4 characters a, b, c and d. Let C be a dictionary table, which determines a mapping of a code to a string, where codes are non-negative integers. Initially, $C(0) = a$, $C(1) = b$, $C(2) = c$ and $C(3) = d$. By operation AddTable, a new code is assigned to a string.

The LZW compression algorithm finds the longest prefix Ω of the current input that is already added in the dictionary table, and outputs the code of Ω. Let x be the following character of Ω. Since $\Omega \cdot x$ is not in the dictionary table, it is added to the dictionary, where "$\cdot$" denotes the concatenation of string/character. The same procedure is repeated from x. Let $C^{-1}(\Omega)$ denote the index of C where Ω is stored. The LZW compression algorithm is described in Algorithm 1 and Table 1 shows the compression flow of an input string "*cbcbcbcda*". It should have no difficult to confirm that 214630 is output by this algorithm.

Algorithm 1 LZW compression algorithm

```
1: Ω ← x_0;
2: for i ← 1 to n − 1 do
3:     if Ω · x_i is in C then
4:         Ω ← Ω · x_i;
5:     else
6:         Output(C^{-1}(Ω)); AddTable(Ω · x_i); Ω ← x_i;
7:     end if
8: end for
9: Output(C^{-1}(Ω));
```

Table 1. LZW compression flow for input string $X = cbcbcbcda$

i	0	1	2	3	4	5	6	7	8	-
x_i	c	b	c	b	c	b	c	d	a	-
Ω	-	c	b	c	cb	c	cb	cbc	d	a
S	-	$cb(4)$	$bc(5)$	-	$cbc(6)$	-	-	$cbcd(7)$	$da(8)$	-
Y	-	2	1	-	4	-	-	6	3	0

Next, let us discuss implementations of dictionary table C. The following operations for a string Ω of characters and the following character x must be supported for LZW compression; determining if $\Omega \cdot x_i$ is in C, returning the value of $C^{-1}(\Omega)$, and performing AddTable($\Omega \cdot x_i$). A straightforward implementation of the dictionary table C, which uses an array such that i-th ($i \geq 0$) element stores $C(i)$. However, since the lengths of strings in C are variable, the straightforward implementation of dictionary C is not efficient. All values of $C(i)$ may be accessed to compute $C^{-1}(\Omega)$. We can use an associative array with keys $C(i)$ and values i, which can be implemented by a balanced binary tree or a hash table. However, these operations take more than $O(|\Omega|)$ time. If the compression ratio is high, Ω may be a long string. Hence, it is not a good idea to use a conventional associative array to implement C.

In this paper, we use a pointer-character table to implement the dictionary table C as shown in Table 2. In this table, a pointer $p(j)$ and a character $c(j)$ are stored for each code j. Also, a back-pointer $q(j, x)$ for every code j and character x is used. Back-pointer table q can be implemented using an associative array which we will discuss later. We can obtain a string $C(j)$ by traversing p until we reach NULL. More specifically, $C(j)$ can be obtained from p and c by the following definition:

$$C(j) = \begin{cases} c(j) & \text{if } p(j) = \text{NULL} \\ C(p(j)) \cdot c(j) & \text{otherwise} \end{cases} \tag{1}$$

We implement operation AddTable($\Omega \cdot x_i$) for dictionary C by performing operation AddTable(j,x_i) for the pointer-character table. If AddTable(j,x_i) is performed, a new entry k with $p(k) = j$ and $c(k) = x_i$ is added to the pointer-character table. In other words, the value k is written in $q(j, x_i)$ of back-pointer table. Using the back-pointer table, we can rewrite LZW compression algorithm in Algorithm 2.

We show how Table 2 is created. First, $j \leftarrow c^{-1}(x_0) = 2$ is executed. Next, since $q(j, x_1) = q(2, b)$ is NULL, Output(2) and AddTable(2,b) are executed.

Table 2. A pointer-character table and a back-pointer table

j	0	1	2	3	4	5	6	7	8	9
$p(j)$	NULL	NULL	NULL	NULL	2	1	4	6	3	0
$c(j)$	a	b	c	d	b	c	c	d	a	-
$q(j,a)$	NULL	NULL	NULL	8	NULL	NULL	NULL	NULL	NULL	NULL
$q(j,b)$	NULL	NULL	4	NULL	NULL	NULL	NULL	NULL	NULL	NULL
$q(j,c)$	NULL	5	NULL	NULL	6	NULL	NULL	NULL	NULL	NULL
$q(j,d)$	NULL	NULL	NULL	NULL	NULL	7	NULL	NULL	NULL	NULL
$C(j)$	a	b	c	d	cb	bc	cbc	$cbcd$	da	-

Algorithm 2 LZW compression algorithm with the back-pointer table

```
1: j ← c^{-1}(x_0);
2: for i ← 1 to n − 1 do
3:     if q(j, x_i) ≠ NULL then
4:         j ← q(j, x_i);
5:     else
6:         Output(j); AddTable(j, x_i); j ← c^{-1}(x_i);
7:     end if
8: end for
9: Output(j);
```

The pointer-character table has new entry $p(4) = 2$ and $c(4) = b$. Also, the value 4 is stored in $q(2, b)$, and operation $j \leftarrow c^{-1}(b) = 1$ is executed. In the next iteration of the for-loop, since $q(1, c)$ is NULL, Output(1) and AddTable(1,c) are executed. The pointer-character table has new entry $p(5) = 1$ and $c(5) = c$, and the value 5 is added in $q(1, c)$. Similarly, we can confirm that a series of codes 214630 is output by this algorithm.

3 Our FPGA Architecture for LZW Compression

This section describes our FPGA architecture of the LZW compression algorithm with back-pointer table using block RAMs in Xilinx Virtex-7 FPGA. We use Xilinx Virtex-7 Family FPGA XC7VX485T-2 as the target device [12]. In the following, we use image data in a TIFF image file to be compressed.

First, we show the implementation of the back-pointer table q for TIFF LZW compression. As shown in the above, the back-pointer table needs $2^{20} \times 12$bits = 1.5MBytes. Since the size of the internal memory in the FPGA is limited and most entries of the table are not used, we use a hash table to implement the back-pointer table q.

In the proposed FPGA implementation, we use a hash table that is suitable for FPGA implementation. The hash table consists of 1024 buckets B_s ($0 \leq s \leq 1023$) and each bucket B_s has 8 entries $e_{s,0}, e_{s,1}, \ldots, e_{s,7}$. To implement this hash table, we use two tables, *number table* and *data table*. Let $|B_s|$ denote the number of values stored in bucket B_s. Each element of the number table stores $|B_s|$. Also, the data table stores values of back-pointers. The table is partitioned into 8 tables, each of which stores one of the 8 entries. Each entry stores 12-bit pointer j, 8-bit character x and 12-bit back-pointer $q(j, x)$. Figure 1 illustrates the structure of the hash table.

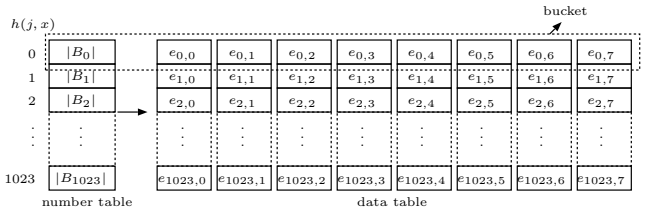

Fig. 1. The arrangement of hash table

Let $h(j, x)$ be a hash function returning a 10-bit number, where pointer j is 12 bits and character x is 8 bits. To specify a 10-bit number, we use a hash function $h(j, x) = ((j << 4) \oplus (j >> 6) \oplus (x << 1) \wedge 0\text{x}3\text{FF}$. Using this hash function, we select a bucket in address $h(j, x)$ and store the value of back-pointer in one of the eight entries in the bucket. However, the bucket may be full, that is, eight values are already stored in the bucket. If this is the case, called *conflict*, the current value of each address $(h(j, x) + i) \wedge 0\text{x}3\text{FF}$ is read for $i = 1, 2, \ldots$ until a bucket that has unused entries is found. We can easily find whether the bucket B_s is full or not by referring $|B_s|$ in the number table. Regarding the size of the hash table, since the total size of the hash table is 8192 and at most 3837 elements are added, conflict may occur, but it is clear that the hash table can store all data.

In the LZW compression, it is necessary to find whether a value of back-pointer is already stored or not. Since the data table is partitioned into 8 tables, we read 8 values at the same time. Therefore, given an address of bucket from the hash function, we can find whether a value that includes the back-pointer is stored or not without checking eight entries in the bucket one by one.

On the other hand, the number table consists of 1024 entries with 4 bits that represent the number of used entries in each bucket. Using the number table, we can simply determine an element whether it is already stored or not. Recall that we need to initialize all entries in the hash table whenever compression for each code segment is finished, that is, ClearCode is output. Since each entry represents the number of used entries in each bucket, we set each entry to zero without clearing the data tables.

In the proposed architecture, we perform LZW compression algorithm described in Algorithm 2. The main part of the architecture is the hash table as described in the above. There are three operations for the hash table, (i) *initialize operation*, (ii) *find operation*, and (iii) *add operation*. We show the details of these operations, as follows.

Initialize operation: As shown in the above, we clear only the number table to initialize the hash table. However, the next characters cannot be input during the initialization. Therefore, in the proposed architecture, we use two number tables and switch them in turn whenever ClearCode is output. Since the number table has 1024 entries, the initialize operation can be performed while another code segment is processed.

Find operation: This operation corresponds to "$q(j, x_i) \neq$ NULL", "$j \leftarrow q(j, x_i)$", and "Output(j)" in Algorithm 2. In the operation, first, we obtain the address of the hash table by computing $h(j, x)$. After that, we find whether a back-pointer $q(j, x)$ is stored in $B_{h(j,x)}$. As shown in the above, we can simultaneously read eight values in a bucket and the number of values in a bucket is read from the number table to read valid data. Since each entry in the hash table has the values of j and x, we can find it by comparing j and x read from the hash table with input values j and x. Therefore, we can check at most 8 entries in $B_{h(j,x)}$ at the same time. After comparing, if $q(j, x)$ is found, output it. Otherwise, we check whether $B_{h(j,x)}$ is full or not. If $|B_{h(j,x)}| < 8$, that is, $B_{h(j,x)}$ is not full, we can find $q(j, x)$ does not exist in the hash table and output NULL. If not, we perform the above operation for bucket $B_{(h(j,x)+i)\wedge 0\text{x}3\text{FF}}$ for $i = 1, 2, \ldots$ until we find whether $q(j, x)$ is stored or not.

Add operation: It is performed as operation AddTable in Algorithm 2. Indeed, it is performed after the find operation as described in Algorithm 2. The entry to be stored locates in the bucket which was referred last in the find operation. Therefore, according to the result of the find operation, we add j, x and $q(j, x)$ to the hash table and increment the corresponding number of stored values in the number table.

In order to implement the hash table, we use block RAMs configured as dual-port mode [13]. Each of the number table consists of one 18k-bit block RAMs. Also, two 18k-bit block RAMs are assigned to one of the 8 tables in the data table. Since we use two tables for the number table, eighteen 18k-bit block RAMs are used in total. For the number table, its dual-port is used as reading port and writing port. They are used to perform the find and add operations, respectively. On the other hand, for the data table, we also use the dual-port as reading port and writing port for each. To reduce the clock cycles, we always suppose that for input string of characters $x_0, x_1, \ldots, x_{n-1}$, the condition $q(j, x_i) =$ NULL is satisfied. Using this, we can continuously input characters unless the condition $q(j, x_i) =$ NULL is not satisfied. When the condition is not satisfied, we need to wait to input the next character.

4 Experimental Results

This section shows the implementation results of the proposed architecture for LZW compression algorithm in the FPGA. We have implemented the proposed circuit for LZW compression algorithm and evaluated it in VC707 board [14] equipped with the Xilinx Virtex-7 FPGA XC7VX485T-2. The experimental results of the implementation is shown in Table 3. We also use Intel Core i7-4790 (3.6GHz) to evaluate the running time of the sequential LZW compression. In the experiment, we have used three gray scale images with 4096×3072 pixels as shown in Fig. 2, which are converted from JIS X 9204-2004 standard color image data. The image "Graph" has high compression ratio since it has large areas with similar intensity levels. The image "Crafts" has low compression ratio since it has small details.

"Crafts" "Flowers" "Graph"

Fig. 2. Three gray scale images with 4096×3072 pixels used for experiments

Table 3. Implementation results of the proposed hardware algorithm

number of circuits	1	24	available
slice registers	104 (0.02%)	3120 (0.51%)	607200
slice LUTs	346 (0.11%)	7782 (2.56%)	303600
18K-bit block RAMs	18 (0.87%)	432 (20.97%)	2060
clock frequency [MHz]	179.99	163.35	—

Table 4 shows the time of compression on CPU and FPGA and the compression ratio (original image size : compressed image size). In our implementation on the FPGA, to save the usage of block RAMs of FPGA, As shown in Table 4, for only one proposed circuit of LZW compression, the results show that implementation on FPGA is not faster than the implementation on the CPU. However, since the proposed circuit uses very few FPGA resources, we have succeeded in implementing 24 identical LZW compression circuits in an FPGA, where the frequency is 163.35MHz. Simply calculated, for image "Crafts", our implementation with 24 circuits runs up to 23.51 times faster than sequential LZW compression on a single CPU.

Table 4. Computing time for three images

images	compression ratio	CPU [ms]	FPGA [ms]	Speed-up
"Crafts"	1.43:1	109.10	101.07	1.08:1
"Flowers"	1.72:1	93.60	107.93	0.87:1
"Graph"	36.72:1	46.79	138.26	0.34:1

For gray scale image "Graph" which has high compression ratio with 4096×3072 pixels, the proposed circuit of LZW compression compresses $4096 \times 3072 \times 1$Byte original data in $138.26ms$, that is, the throughput of the proposed circuit is 86.79MBytes/s. On the other hand, for gray scale image "Crafts" which has low compression ratio, the throughput is 118.73MBytes/s.

5 Conclusions

We have presented a hardware architecture for LZW compression algorithm of compressing images. In the proposed architecture, we efficiently use dual-port block RAMs embedded in the FPGA to implement a hash table that is used as

the dictionary. It was implemented in a Virtex-7 family FPGA XC7VX485T-2. The experimental results show that our module provides a throughput up to 118.73MBytes/s. Since the proposed circuit uses a few resources of the FPGA, we have succeeded in implementing 24 identical LZW compression circuits in an FPGA. The implementation of 24 LZW compression circuits attains a speed up factor of 23.51 over the sequential implementation on the CPU.

References

1. Adobe Developers Association: TIFF Revision 6.0 (June 1992), `http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf`
2. Funasaka, S., Nakano, K., Ito, Y.: Fast LZW compression using a GPU. In: in Proc. of International Symposium on Computing and Networking. pp. 303–308 (2015)
3. Helion Technology: LZRW3 Data Compression Core for Xilinx FPGA (October 2008)
4. Klein, S.T., Wiseman, Y.: Parallel Lempel Ziv coding. Discrete Applied Mathematics 146(2), 180–191 (2005)
5. Lin, M.: A hardware architecture for the LZW compression and decompression algorithms based on parallel dictionaries. Journal of VLSI signal processing systems for signal, image and video technology 26(3), 369–381 (2000)
6. Lin, M., Lee, J., Jan, G.E.: A Lossless Data Compression and Decompression Algorithm and Its Hardware Architecture. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 14(9), 925–936 (2006)
7. Mishra, M.K., Mishra, T.K., Pani, A.K.: Parallel Lempel-Ziv-Welch (PLZW) technique for data compression. International Journal of Computer Science and Information Technologies 3(3), 4038–4040 (2012)
8. Navqi, S., Naqvi, R., Riaz, R.A., Siddiqui, F.: Optimized RTL design and implementation of LZW algorithm for high bandwidth applications. Electrical Review 4, 279–285 (2011)
9. Prakash, S., Purohit, M., Raizada, A.: A novel approach of speedy-highly secured data transmission using cascading of PDLZW and arithmetic coding with cryptography. International Journal of Computer Applications 57(19) (2012)
10. Shyni, K., Kumar, K.V.M.: Lossless LZW data compression algorithm on CUDA. IOSR Journal of Computer Engineering pp. 122–127 (2013)
11. Welch, T.A.: A technique for high-performance data compression. IEEE Computer 17(6), 8–19 (June 1984)
12. Xilinx Inc.: 7 Series FPGAs Configuration User Guide (2013)
13. Xilinx Inc.: 7 Series FPGAs Memory Resources User Guide (Nov 2014)
14. Xilinx Inc.: VC707 Evaluation Board for the Virtex-7 FPGA User Guide (2014)