An Efficient Implementation of LZW Decompression in the FPGA

Xin Zhou, Yasuaki Ito, and Koji Nakano
Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, Hiroshima, 739-8527 Japan

***Abstract*—LZW algorithm is one of the most famous dictionary-based compression and decompression algorithms. The main contribution of this paper is to present a hardware LZW decompression algorithm and to implement it in an FPGA. The experimental results show that one proposed module on Virtex-7 family FPGA XC7VX485T-2 runs up to 2.16 times faster than sequential LZW decompression on a single CPU, where the frequency of FPGA is 301.02MHz. Since the proposed module is compactly designed and uses a few resources of the FPGA, we have succeeded to implement 150 identical modules which works in parallel on the FPGA, where the frequency of FPGA is 245.4MHz. In other words, our implementation runs up to 264 times faster than a sequential implementation on a single CPU.**

***Keywords*-LZW decompression, FPGA, block RAMs**

I. 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. Data compression includes two basic methods, lossy compression and lossless compression. Lossy compression uses the fact that human are not sensitive to some frequency ingredients of image or sound. Some information of the original data are discarded in lossy compression. Thus, the decompressed data are not identical to the original data. On the other hand, lossless compression preserves all information of the original data. In other words, the decompression of lossless compression creates exactly the same data with the original data.

Some famous compression algorithm are proposed such as LZ77 [1], LZ78 [2] and LZW [3]. LZ77 algorithm uses two buffers such as dictionary buffer and preview buffer. Dictionary buffer includes the processed data and preview buffer stores the pending data. In LZ77 algorithm, the longest string of preview buffer matching to the string of dictionary buffer is converted to a code that corresponds to the index of dictionary buffer. However, it is not suitable to hardware implementation since it needs a large dictionary buffer and preview buffer. LZ78 algorithm creates a dictionary table and finds the longest matched string in the dictionary table. If there is no matched string in the dictionary table, it outputs the index of dictionary table and the last character of the unmatched string. LZW algorithm is a variant of LZ78 algorithm that outputs only the index of matched string of dictionary table. In this paper, we focus on LZW compression which is used in Unix utility "compress" and in GIF image format. LZW compression is included in TIFF standard [4], which is widely used in the area of commercial digital printing. Since dictionary tables are created by reading input data one by one, LZW compression and decompression are hard to parallelize. A high-definition image or video is compressed to a file once, and stored to the server of a commerical organization to be acessed by users in different regions or countries. The compressed file is transferred to users through the network, and then is decompressed locally. Hence, decompression is performed more frequently than compression. The main goal of this paper is to develop an efficient hardware architecture of LZW decompression and implement it in an FPGA.

An FPGA (Field Programmable Gate Array) 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 [5]. Recent FPGAs have embedded block RAMs. As illustrated in Figure 1, the Xilinx Virtex-7 family FPGAs have block RAMs, each of which 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 [6]. 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. They are widely used in consumer and industrial products for accelerating processor intensive algorithm [7], [8], [9], [10], [11], [12].

Numerous implementations of variety of LZW decompression on FPGAs or VLSIs have been proposed to accelerate the computation. LZRW3 data compression core [13] is designed by Helion technology. This data compression core uses LZRW3 algorithm that is a variant of LZ77 algorithm, and provides a maximum decompression rate of 180.75MBytes/s clocked at 226MHz in Xilinx Virtex-5 FPGA. Navqi *et al.* [14] implemented a variant of LZW algorithm in Xilinx Virtex-2 FPGA, where only one fixed-length dictionary table is used. This implementation provides a maximum decompression rate of 160MBytes/s clocked at 50MHz in Xilinx Virtex-2 FPGA. Several implementations of data decompression are proposed based on PDLZW(Parallel Dictionary LZW) algorithm that is a variant of LZW algorithm [15], [16], [17]. Instead of one variable-length

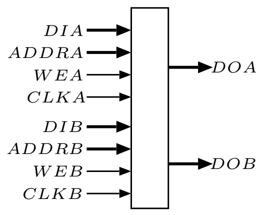

Figure 1. The dual-port block RAM in Xilinx FPGAs

table used in LZW algorithm, multiple fixed-length tables are used in PDLZW algorithm to accelerate the speed of data compression and decompression. Lin implemented the PDLZW algorithm in a VLSI that provides a maximum decompression rate of 45.5MBytes/s clocked at 62.5MHz [15]. Lin *et al.* also proposed a two stage hardware architecture that combines PDLZW and AH(Adaptive Huffman) algorithm and implement it in a VLSI [16]. By decreasing the number of parallel dictionaries, this implementation provides a maximum decompression rate of 83MBytes/s clocked at 100MHz. On the other hand, there is some research for accelerating the computation of LZW decompression using GPUs (Graphics Processing Units) [18], [19], multiprocessor [20] and cluster systems [21]. However, as far as we know, there is no hardware implementation of the original LZW decompression algorithm since it is not easy to implement it.

The main contribution of this paper is to present an efficient hardware LZW decompression algorithm and to implement it in an FPGA. In general, LZW decompression uses a dictionary table which stores variable-length strings. However, in our hardware algorithm we use two tables, pointer table p and character table C_f which store a single value in each entry. The algorithm consists of three steps and these steps are concurrently executed efficiently using the dual-port block RAMs. The proposed module of hardware LZW decompression algorithm in Virtex-7 family FPGA uses 278 slice registers, 307 slice LUTs and 13 block RAMs with 18k-bit, where the frequency of FPGA is 301.02MHz. The running time of proposed module attains a speed up factor that surpasses 2.16 times over a sequential algorithm on a single CPU. Since the decompression rate is data dependent, according to the experimental results, the decompression rate of our module is about 280.17MBytes/s while the compression ratio of input file is extremely high. Even in the worst condition, the decompression rate of proposed module is about 143.54MBytes/s. Since the proposed FPGA module is compactly designed, we have succeeded to implement 150 same modules in an FPGA, where all modules works in parallel clocked at 245.4MHz. Calculated simply, the implementation with 150 paralleled modules can run up to 264 times faster than the sequential algorithm on a single CPU.

This paper is organized as follows. Section II reviews the LZW compression and decompression algorithms. We show a new hardware LZW decompression algorithm which is suitable to be implemented in an FPGA in Section III . In Section IV, we show an efficient FPGA implementation of the hardware LZW decompression algorithm. Section V shows the experimental results of the performance of the hardware LZW decompression algorithm. Finally, we conclude this paper in Section VI.

II. LZW COMPRESSION AND DECOMPRESSION

The main purpose of this section is to review LZW compression and decompression algorithms. For details of the algorithms, the interested reader may refer to Section 13 in [4].

The LZW (Lempei-Ziv-Welch) [3] lossless data compression algorithm always gives competitive compression efficiency. In this algorithm, an input string of characters is compressed 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 integers. It reads characters in an input image string one by one and adds an entry in a string table (or a dictionary). In the same time, it writes an output string of codes by looking up the string 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 of handling the boundary case, we assume that an input string is a string of 4 characters a, b, c and d. Let S be a string table, which determines a mapping of a string to a code, where codes are non-negative integers. Initially, $S(a) = 0$, $S(b) = 1$, $S(c) = 2$ and $S(d) = 3$. By procedure AddTable, new code is assigned to a string. For example, if AddTable(cb) is executed after initialization of S, we have $S(cb) = 4$. The LZW compression algorithm is described as follows:

[LZW compression algorithm]
```
$\Omega \leftarrow \phi$
for $i \leftarrow 0$ to $n-1$ do
    if($\Omega \cdot x_i$ is in $S$)
        $\Omega \leftarrow \Omega \cdot x_i$;
    else   Output($S(\Omega)$); AddTable($\Omega \cdot x_i$); $\Omega \leftarrow x_i$;
Output($S(\Omega)$);
```

where "$\cdot$" denotes the concatenation of characters and Ω denotes a string.

Table I shows the compression flow of an input string "*cbcbcbcda*". First, since $\Omega \cdot x_0 = c$ in S, $\Omega \leftarrow c$ is performed. Next, since $\Omega \cdot x_1 = cb$ is not in S, $S(c) = 2$ is output and we have $S(cb) = 4$. Also, $\Omega \leftarrow x_1 = b$ is performed. It should have no difficult to confirm that 214630 is output by this algorithm.

Table I
LZW COMPRESSION FLOW FOR INPUT STRING $X = cbcbcbcda$

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

Table II
CODE TABLE C AND THE OUTPUT STRING FOR 214630

i	y_i	C	X
0	2	-	c
1	1	$cb(4)$	b
2	4	$bc(5)$	cb
3	6	$cbc(6)$	cbc
4	3	$cbcd(7)$	d
5	0	$da(8)$	a

Next, let us show LZW decompression algorithm. Let C be *the code table*, the inverse of string table S. For example if $S(cb) = 4$ then $C(4) = cb$. Initially, $C(0) = a$, $C(1) = b$, $C(2) = c$, and $C(3) = d$. Also, let $C_1(i)$ denote the first character of code i. For example $C_1(4) = c$ if $C(4) = cb$. Similarly to LZW compression, the LZW decompression algorithm reads a string Y of codes one by one and adds an entry of a code table. In the same time, it writes a string X of characters. The LZW decompression algorithm is described as follows:

```
[LZW decompression algorithm]
Output($C(y_0)$);
for $i \leftarrow 1$ to $n-1$ do
  if($y_i$ is in $C$)
    Output($C(y_i)$); AddTable($C(y_{i-1}) \cdot C_1(y_i)$);
  else
    Output($C(y_{i-1}) \cdot C_1(y_{i-1})$); AddTable($C(y_{i-1}) \cdot C_1(y_{i-1})$);
```

Table II shows the decompression process for a code string 214630. First, $C(2) = c$ is output. Since $y_1 = 1$ is in C, $C(1) = b$ is output and AddTable(cb) is performed. Hence, $C(4) = cb$ holds. Next, since $y_2 = 4$ is in C, $C(4) = cb$ is output and AddTable(bc) is performed. Thus, $C(5) = bc$ holds. Since $y_3 = 6$ is not in C, $C(y_2) \cdot C_1(y_2) = cbc$ is output and AddTable(cbc) is performed. The reader should have no difficulty to confirm that $cbcbcbcda$ is output by this algorithm.

Since the length of strings in C are variable, it is difficult to implement this algorithm on hardware as it is. Therefore, we introduce a new LZW decompression algorithm without such dictionary table in the next section.

III. LZW DECOMPRESSION ALGORITHM FOR HARDWARE IMPLEMENTATION

This section is to propose a new LZW decompression algorithm for hardware implementation. The hardware algorithm does not use any variable length dictionary table. In following, we explain the details of the algorithm.

Again let $X = x_0 x_1 \cdots x_{n-1}$ be a string of characters. We assume that characters are selected from an alphabet (or a set) with k characters $\alpha(0), \alpha(1), \ldots, \alpha(k-1)$. We use $k = 4$ characters $\alpha(0) = a$, $\alpha(1) = b$, $\alpha(2) = c$, and $\alpha(3) = d$, when we show examples as before. Let $Y = y_0 y_1 \cdots y_{m-1}$ denote the compressed string of codes obtained by the LZW compression algorithm.

Before showing the LZW decompression for hardware implementation, we define several notations. We define pointer table p using code table Y as follows:

$$p(i) = \begin{cases} \text{NULL} & \text{if } 0 \le i \le k-1 \\ y_{i-k} & \text{if } k \le i \le k+m-1 \end{cases} \qquad (1)$$

We can traverse pointer table p until we reach NULL. Let $p^0(i) = i$ and $p^{j+1}(i) = p(p^j(i))$ for all $j \ge 0$. In other words, $p^j(i)$ is the code where we reach from code i in j pointer traversing operations. Let $L(i)$ be an integer satisfying $p^{L(i)}(i) = \text{NULL}$ and Let C_f be the character table defined as follows:

$$C_f(i) = \begin{cases} \alpha(i) & \text{if } 0 \le i \le k-1 \\ C_f(p(i)) & \text{if } k \le i \le k+m-1 \end{cases} \qquad (2)$$

It should have no difficulty to confirm that $C_f(i)$ represents the first character of $C(i)$, and $L(i)$ is the length of $C(i)$. Using C_f and p, we can define the value of $C(i)$ in following. If $0 \le i \le k-1$, $C(i) = C_f(i)$. On the other hand, if $k \le i \le k+m-1$, $C(i) = C_f(p^{L(i)-1}(i)) \cdot C_f(p^{L(i)-2}(i)+1) \cdot C_f(p^{L(i)-3}(i)+1) \cdots C_f(p^0(i)+1)$.

Table III shows the value of p, C_f, L, and C for $Y = 214630$. According to the table, we can obtain the decompressed string. Figure 2 shows an example of obtaining the decompression string of code $y_3 = 6$, that is $C(6)$, from the table. For code $y_3 = 6$, we first read $p(6) = 4$ from table p. Also, we read $C_f(6+1) = c$ from table C_f that corresponds to the last character of $C(6)$. Since the obtained pointer 4 is not NULL, we continue the traversing of table. Next, $p(4) = 2$ and $C_f(4+1) = b$ are read from tables p and C_f, respectively. Finally, pointer $p(2)$ is read out, and we stop the traversing operation for code y_3 because $p(2)$ is NULL. Also, $C_f(2) = c$ is read out as the first character of the string corresponding to code y_3. We note that each character of string corresponding to a code is obtained in reverse order.

We are now in position to show hardware LZW decompression. This algorithm can be done in three steps as follows:

Step 1: Update tables p and C_f.

Table III
THE VALUES OF p, L, C_f AND C IF $Y = 214630$

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

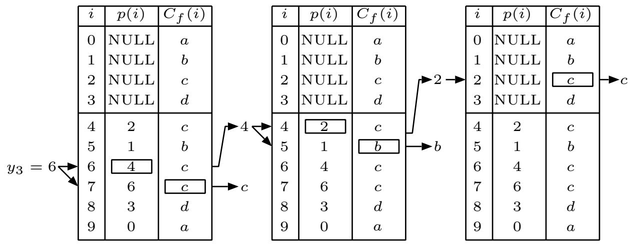

Figure 2. An example of traversing tables p and C_f

Step 2: Compute partially-reversal strings of X and L using p and C_f.
Step 3: Reorder decompression string X.

In Step 1, we initialize the values of $p(i)$, $C_f(i)$ for each i $(0 \leq i \leq k-1)$. After that, we compute the values of $p(i)$ and $C_f(i)$ for each i $(k \leq i \leq k+m-1)$. The details of Step 1 are spelled out as follows:

[Step 1 of hardware LZW decompression algorithm]
```
for $i \leftarrow 0$ to $k-1$ do
    $p(i) \leftarrow$ NULL; $C_f(i) \leftarrow \alpha(i)$;
for $i \leftarrow k$ to $k+m-1$ do
    $p(i) \leftarrow y_{i-k}$; $C_f(i) \leftarrow C_f(y_{i-k})$;
```

In Step 2 of hardware LZW decompression algorithm, for each compressed code y_i $(0 \leq i \leq m-1)$ of Y, $C^r(y_i)$ is read from table C_f by traversing pointer table p, where $C^r(i)$ denotes a string obtained by reversing $C(i)$. At the same time, the length of string $L(i)$ is also computed. By traversing table C_f with table p, the reversed string is read and temporally stored to an output buffer for each character. Let o denote a table for storing characters of concatenation of strings $C^r(y_0) \cdot C^r(y_1) \cdots C^r(y_{m-1})$. For example, if $C(7) = abc$, in Step 2, we have $C^r(7) = cba$ and $L(7) = 2$. The details of Step 2 of hardware LZW decompression algorithm are shown as follows:

[Step 2 of hardware LZW decompression algorithm]
```
$addr \leftarrow 0$
for $i \leftarrow 0$ to $m-1$ do
    $j \leftarrow y_i$; $L(i) \leftarrow 0$;
    while($p(j) \neq$ NULL)
        $o(addr) \leftarrow C_f(j+1)$; $j \leftarrow p(j)$;
        $L(i) \leftarrow L(i)+1$; $addr \leftarrow addr+1$;
    $o(addr) \leftarrow C_f(j)$; $L(i) \leftarrow L(i)+1$; $addr \leftarrow addr+1$;
```

In Step 3 of hardware LZW decompression algorithm, a concatenated string of $C^r(y_0) \cdot C^r(y_1) \cdots C^r(y_{m-1})$ stored in output buffer o is arranged in corrected order, that is, $C(y_0) \cdot C(y_1) \cdots C(y_{m-1})$. Each ordered character is output one by one. The algorithm code of Step 3 is shown as follows:

[Step 3 of hardware LZW decompression algorithm]
```
$addr \leftarrow 0$;
for $i \leftarrow 0$ to $m-1$ do
    $l \leftarrow L(i)$;
    while($l > 0$)
        Output($o(addr+l-1)$); $l \leftarrow l-1$;
    $addr \leftarrow addr + L(i)$;
```

By sequentially executing Step 1, Step 2, and Step 3, LZW decompression can be performed. In addition, the execution of these steps can be overlapped. More specifically, after an execution for an input code in each step is completed, the execution for the code in the next step can be started. Figure 3 illustrates a process of the above execution for an input compressed code $Y = y_0 y_1 \cdots y_{m-1}$. In our FPGA implementation described in the next section, we use block RAMs of FPGA to implement the pointer table p, character table C_f, and output buffer o. In the utilized FPGA, the block RAMs can be configured as a dual-port block RAM. Since dual-port block RAM has two set of ports that work independently, the writing and reading operations of these tables can be performed concurrently. Using the block RAMs efficiently, we realizes the overlapped execution of the three steps.

IV. OUR FPGA ARCHITECTURE FOR LZW DECOMPRESSION

This section describes our FPGA architecture of the hardware LZW decompression algorithm using block RAMs in Xilinx Virtex-7 FPGA. We use Xilinx Virtex-7 Family FPGA XC7VX485T-2 as the target device [22].

In this paper, we focus on the decompression of LZW-compressed data in a TIFF image file. We assume that a TIFF image file contains a gray-scale image with 8-bit depth, that is, each pixel has intensity represented by an 8-bit unsigned integer. Since each of RGB or CMYK color planes can be handled as a gray scale image, it is obvious to modify gray scale image TIFF decompression for color image decompression. For further details on a TIFF image file, the interested reader may refer to [4].

First, we show how image data in a TIFF image file is compressed. Since every pixel has an 8-bit intensity level,

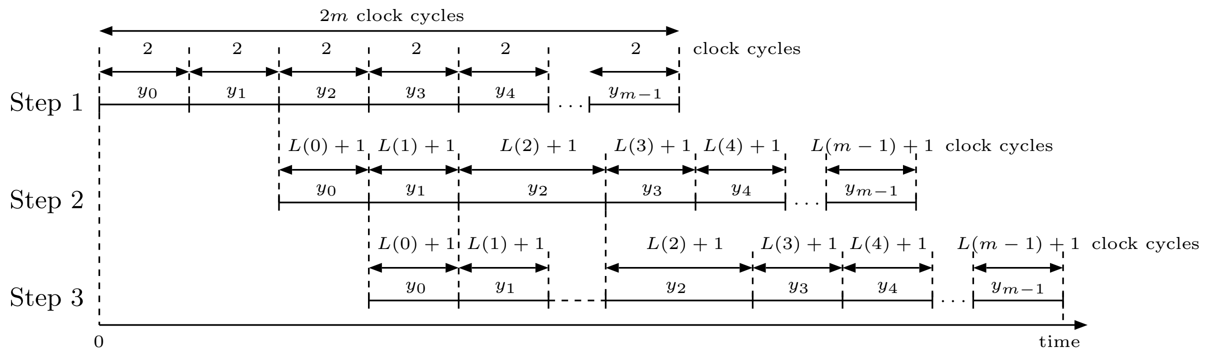

Figure 3. Process of our LZW decompression hardware for an input compressed code $Y = y_0y_1 \cdots y_{m-1}$

we can think that an input string of an integer in the range $[0, 255]$. Hence, codes from 0 to 255 are assigned to these integers. Code 256 (ClearCode) is reserved to clear the code table. Code 257 (EndOfInformation) is used to specify the end of the data. Thus, AddTable operations assign codes to strings from code 258. While the entry of the code table is less than 512, codes are represented as 9-bit integer. After adding code table entry 511, we switch to 10-bit codes. Similarly, after adding code table entry 1023 and 2037, 11-bit codes and 12-bit codes are used, respectively. As soon as code table entry 4094 is added, ClearCode is output. After that, the code table is re-initialized and AddTable operations use codes from 258 again. The same procedure is repeated until all pixels are converted into codes. After the code for the last pixel is output, EndOfInformation is written out. We can think that a code string is separated by ClearCode, We call each of them *a code segment*. Except the last one, each code segment has $4094 - 257 + 1 = 3838$ codes. The last code segment may have codes less than that.

Figure 4 shows the proposed architecture of LZW decompression. In our implementation, the LZW-based module decompresses all codes in a segment that are given one by one. In order to implement pointer table p, character table C_f, code buffer and output buffer o, the block RAMs are used. The block RAMs are configured as dual-port mode [6] as shown in Figure 1. A dual-port block RAM has two sets of ports which work independently. We use these two port to perform executions in three steps described in Section III in parallel. For table p, as shown in the previous section, since the values of $p(i)$ $(0 \leq i \leq 255)$ are initialized to NULL and codes 256 and 257 are reserved codes, we do not actually use the block RAM in that range to reduce the memory resources as illustrated in Figure 5. Instead of use of the block RAM, the circuit checks the value of the address. Namely, if the address is in $[0, 255]$, NULL is output. Otherwise the value of $p(i)$ is read from the block RAM. For the same reason, table C_f do not use the block RAM in the range. Each value of $C_f(i)$ $(0 \leq i \leq 255)$ is initialized to an alphabet $\alpha(i)$. From the target application, we can assume that $\alpha(i) = i$ $(0 \leq i \leq 255)$ since the alphabets correspond to pixel values. If the address is in $[0, 255]$, the value of the address is just output.

We can obtain a string of each code by traversing tables p and C_f. To store the characters, an output buffer o is used. Output buffer o is also configured as dual-port block RAMs. Since the characters of corresponding string of a code is reversely read out from table C_f and then written to the output buffer for each character in reversed order, we use table L to store the length of the string to reverse it in the following step. Finally, we read the characters from output buffer and reverse it with the length. Indeed, it is not necessary to store all the values of L since the executions of three steps described in Section III. Therefore, table L is configured as a FIFO (First-In-First-Out). As the same reason, we use a FIFO, which is also composed of the block RAMs, to temporally store input codes called code buffer. For the reader's benefit, the behavior of the proposed architecture for each step is described next.

Step 1: In Step 1, for tables p and C_f, one port set of the dual-port block RAMs is used to perform the updating operation as described in the algorithm in Section III, respectively. The table update is executed for given compressed codes y_i $(0 \leq i \leq m-1)$ one by one. If an input code $y_i \leq 257$ holds, it is unnecessary to update both tables since the elements in p and C_f are constant values for $i \leq 257$. Otherwise, if $y_i \geq 258$, table p is updated by writing y_i to $p(i + 258)$. The update for table p can be easily done since y_i is stored to an element at address i in the block RAM as illustrated in Figure 5. Also, the update operation for table C_f is performed. It takes 2 clock cycles to read a value stored at $C_f(y_i - 258)$ and write it to $C_f(i)$. The above operations are repeatedly executed for each input code. Since the update operations for both tables can be executed at the same time, it takes two clock cycles for each input code. Since m codes are input, in total, $2m$ clock cycles are necessary to perform Step 1. Recall that all each

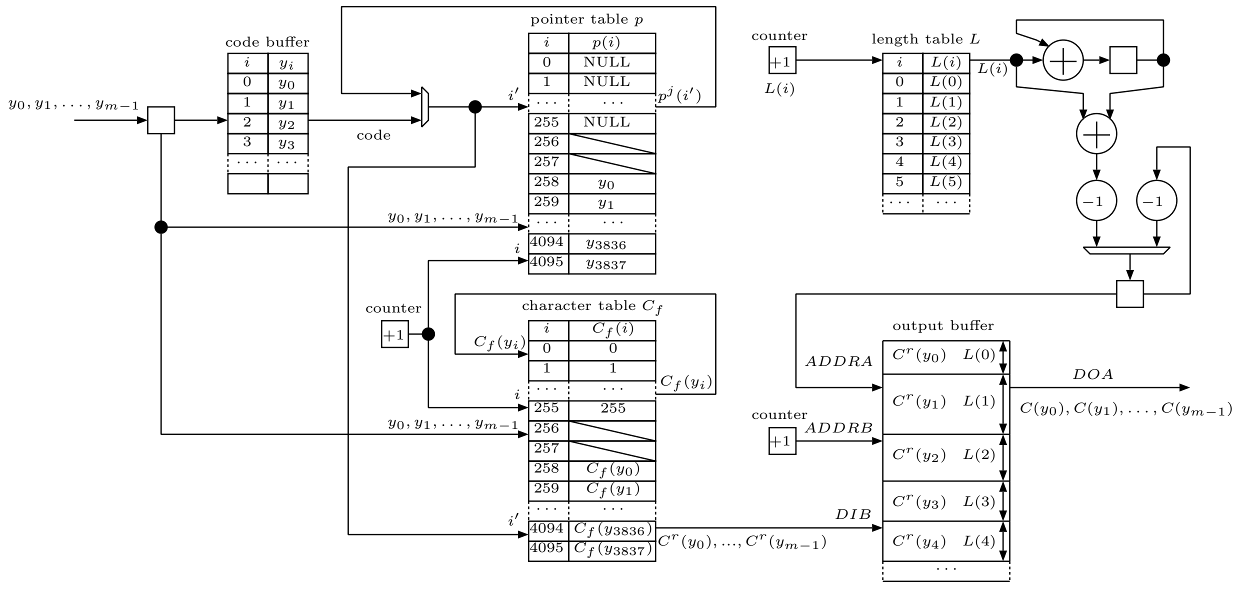

Figure 4. The outline of our FPGA architecture for hardware LZW algorithm

code segment has 3838 codes except the last one. For each code segment that has 3838 codes, table p and C_f are full if the update operations for all codes of one code segment are performed. The update operation is terminated until all codes of this segment are decompressed. For the last code segment, if code 257 (EndOfInformation) is reached, the update operation is terminated until all codes of the last code segment are decompressed.

Step 2: We will show how to obtain partially-reversed strings $C^r(y_i)$ $(0 \leq i \leq m-1)$ with table p and C_f updated in Step 1. In the following, we use another port set of the dual-port block RAMs of tables p and C_f, respectively. As shown in the algorithm of Step 2 in the previous section. for each input code $y_i(0 \leq i \leq 3837)$, we traverse tables p and output characters of $C^r(y_i)$ in table C_f. Since it takes one clock cycle to read an element in tables p and $C^r(y_i)$, respectively, two clock cycles are necessary to output a character in $C^r(y_i)$. However, the access to tables p and C_f can be performed currently. We can overlap the access for an input code y_i with that for the next code y_{i+1}. Therefore, when the length of $C^r(y_i)$ is $L(i)$, we can output a string $C^r(y_i)$ in $L(i) + 1$ clock cycles. The characters of $C^r(y_i)$ are stored into an output buffer o one by one. Also, $L(i)$ is counted at the same time. After outputting the characters of $C^r(y_i)$, and $L(i)$ is stored to table L which is composed of a dual-port block RAM. Since it takes $L(i) + 1$ clock cycles to output for each input code y_i, Step 2 is performed in $\Sigma_{i=0}^{m-1}(L(i) + 1)$ clock cycles in total.

Step 3: In Step 3, partially-reversed strings $C^r(y_0)$, $C^r(y_1)$,..., $C^r(y_{m-1})$, stored in output buffer o in Step 2

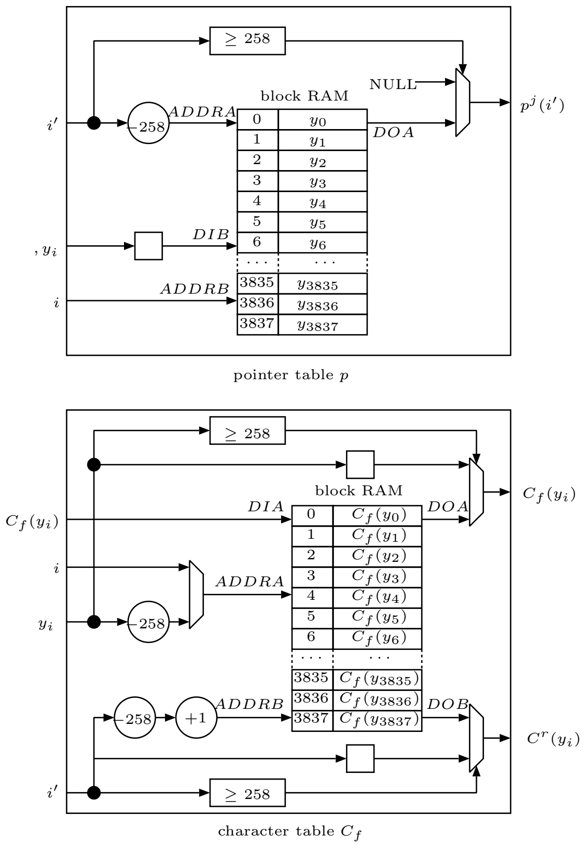

Figure 5. Dual-port block RAM and memory configurations of tables p and C_f

is reordered to the uncompressed strings $C(y_0)$, $C(y_1)$,..., $C(y_{m-1})$. Since the length of each string is known from $L(i)$, each character can be read reversely from output buffer o one by one. Each operation for an input code y_i can be started after $C^r(y_i)$ is stored to output buffer o, that is, $L(i)$ is stored into table L. It takes $L(i) + 1$ clock cycles to perform the operation for a code y_i since one clock cycle for reading $L(i)$ and L(i) clock cycles for reversely reading $C(y_i)$ are necessary.

Let us consider the overlapped execution among the three steps as illustrated in Figure 3. Recall that Step 1 can be performed in 2 clock cycles for each input code. The operation for an input code y_i $(0 \leq i \leq m-1)$ in Step 2 can be performed after the operation for the next code y_{i+1} in Step 1 is finished. Also, the execution time for each y_i is at least 2 clock cycles since $L(i) + 1 \geq 2$. Therefore, the execution of Step 2 can be started 4 clock cycles later after the first code y_0 is given in Step 1 and performed continuously. In Step 3, the operation for an input code y_i can be performed after the operation for y_i in Step 2. Namely, the operation for y_i in Step 3 can be executed when the operation for y_j $(y_j \geq i+1)$ in Step 2. Therefore, in Step 3, the execution sometimes waits for the execution in Step 2. Let us consider the longest case for computing time that an input data obtained by compressing data whose elements are the same value is given. For example, when a string $X = 0, 0, 0, \ldots$ is compressed, the compressed data is $Y = 0, 258, 259, \ldots$. The length $L(i)$ of each uncompressed string $C(y_i)$ can be represented as $L(i) = i + 1$ $(0 \leq i \leq m-1)$ since the lengths are incremented by one for each code. Since $L(i+1) = L(i)+1$ in this case, $L(i) < L(i+1)$ always holds. Therefore, the execution for y_i in Step 3 can be performed when that for y_{i+1} is performed concurrently. Moreover, the execution for each y_i in Step 3 waits for one clock cycle. In such case, it takes $\Sigma_{i=0}^{m-1}(L(i) + 2) = m(m+5)/2$ clock cycles. Since the execution of Step 2 can be started 4 clock cycles later after the first code y_0 is given in Step 1 and Step 3 can be started $2(= L(0) + 1)$ clock cycles later after the execution of Step 2 is started, Step 3 can be started 6 clock cycles later after the first code y_0 is given in Step 1. Therefore, in such case, it takes $m(m+5)/2 + 6$ clock cycles to perform the LZW decompression in total.

V. Experimental Results

This section shows the implementation results of the hardware LZW decompression algorithm in the FPGA.

We have implemented the proposed architecture for hardware LZW decompression algorithm and evaluated it in VC707 board [23] equipped with the Xilinx Virtex-7 FPGA XC7VX485T-2. According to the implementation results, one LZW decompression module uses 278 slice registers, 307 slice LUTs and 13 18K-bit block RAMs. We have succeeded to implement 150 proposed LZW decompression modules which work in parallel in the FPGA. The experimental results of the implementation is shown in Table IV. We also use Intel Xeon CPU E5-2430 (2.2GHz) to evaluate the running time of sequential LZW decompression. We have used three gray scale images with 4096×3072 pixels as shown in Figure 6, which are converted from JIS X 9204-2004 standard color image data. Table V shows the time of decompression on CPU and FPGA and the compression ratio $(\frac{original\ image\ size}{compressed\ image\ size})$. The image "Graph" has high compression ratio since it has large areas with similar intensity levels. The image "Crafts" has small compression ratio since it has small details. Both CPU and FPGA decompression take more time to create dictionary tables if the image has small compression ratio. In LZW decompression on CPU, the operation of creating dictionary tables occupies most of the computing time. In our implementation on FPGA, the operation of creating tables is performed independently, and writing characters to output buffer and reading characters from output buffer are paralleled, hence, the operation of outputting characters occupies most of the time. As shown in Table V, for only one proposed module, the results show that implementation on FPGA is 2.16 times faster than the implementation on the CPU. For example, in our FPGA implementation of one proposed module, it takes 19674631 clock cycles to decompress image "Crafts", i.e., $\frac{19674631}{301.02\text{MHz}} = 65.36ms$. It takes 18339574 clock cycles to decompress image "Flowers", i.e., $\frac{18339574}{301.02\text{MHz}} = 60.924ms$. To decompress image "Graph", it only takes 12892927 clock cycles, i.e., $\frac{12892927}{301.02\text{MHz}} = 42.831ms$. Hence, for gray scale image "Graph" which has high compression ratio with 4096×3072 pixels, the LZW decompression module outputs 4096×3072×1 Bytes of original data in $42.83ms$. Therefore, the decompression rate of module is $\frac{4096\times3072\times1\text{Bytes}}{42.831ms} = 280.17$MBytes/s. Since the decompression rate depends on input data, the decompression rate can be even better if the compression rate of input file is higher. Suppose that in the worst case for computing time, 4096×3072 input codes are given, all of which corresponding strings include 1 character. Since it takes 2 clock cycles to decompress each code that includes only 1 character, all the codes can be decompressed in $4096 \times 3072 \times 2 = 25165824$ clock cycles, i.e., $\frac{25165824}{301.02\text{MHz}} = 83.602ms$. More specifically, the minimum decompression rate of proposed module is $\frac{4096\times3072\times1\text{Byte}}{83.602ms} = 143.54$MByte/s. Since the proposed FPGA module uses a few resources of the FPGA, we have succeeded to implement 150 modules in a FPGA, where all modules work in parallel. Since the number of modules increases, the frequency of FPGA decreases to 245.4MHz. Simply calculated, our implementation with 150 modules runs up to 264 times faster than sequential LZW decompression on a single CPU.

There are some literatures reported to implement data decompression using the FPGA shown in Section I. Performances such as method, slices, block RAMs, frequency and

“Crafts” “Flowers” “Graph”

Figure 6. Three gray scale image with 4096×3072 pixels used for experiments

Table IV
IMPLEMENTATION RESULT OF ONE MODULE OF HARDWARE LZW DECOMPRESSION ALGORITHM

number of modules	1	150	Available
Slice Registers	278 (0.05%)	40642 (6.69%)	607200
Slice LUTs	307 (0.1%)	45194 (14.89%)	303600
18K-bit block RAMs	13 (0.63%)	1950 (94.66%)	2060
Clock frequency [MHz]	301.02	245.4	—

decompression rate are compared in Table VI. It is difficult to directly compare to other works because used FPGAs, algorithms and size of dictionary differ. Our implementation provides a competitive decompression rate with other works. For a file compressed by the original LZW compression algorithm, only our implementation can directly decompress it.

Table V
COMPUTING TIME (MILLISECONDS) FOR THREE IMAGES

images	compression ratio	time of CPU	time of FPGA	Speedup ratio
“Crafts”	1.43:1	141.534	65.36	2.16:1
“Flowers”	1.72:1	127.136	60.924	2.08:1
“Graph”	36.72:1	75.901	42.831	1.77:1

Table VI
COMPARISON WITH RELATED WORKS FOR DATA DECOMPRESSION

	Helion [13]	Navqi [14]	This work
Method	LZRW3	Variant of LZW	LZW
Device	Xilinx Virtex-5	Xilinx XC2V250	Xilinx XC7VX485T
Slices	166	247	139
18K-bit block RAMs	4	8	13
Frequency [MHz]	226	50	301.02
Decompression rate [MBytes/s]	180.75	160	280.17

VI. CONCLUSIONS

We have presented a hardware LZW decompression algorithm of decompressing images. It was implemented in a Virtex-7 family FPGA XC7VX485T-2. According to the implementation results, one LZW decompression module uses 278 slice registers, 307 slice LUTs and 13 block RAMs with 18K-bit. According to simulating results, one FPGA module of LZW decompression is more than 2.16 times faster than sequential LZW decompression on a single CPU. Our module provides a decompression rate up to 280.17MBytes/s which is higher than other research. Since the proposed module uses a few resources of the FPGA, we have succeeded to implemented 150 LZW decompression modules in parallel which attains a speed up factor of 264 over the sequential implementation on the CPU.

REFERENCES

[1] J. Ziv and A. Lempel, “A Universal Algorithm for Sequential Data Compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[2] ——, “Compression of Individual Sequences via Variable-Rate Coding,” *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[3] T. A. Welch, “A technique for high-performance data compression,” *IEEE Computer*, vol. 17, no. 6, pp. 8–19, June 1984.

[4] Adobe Developers Association, *TIFF Revision 6.0*, June 1992. [Online]. Available: http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf

[5] K. Nakano and Y. Yamagishi, “Hardware n Choose k Counters with Applications to the Partial Exhaustive Search,” *IEICE Transactions on Information & Systems*, 2005.

[6] Xilinx Inc., *7 Series FPGAs Memory Resources User Guide*, Nov. 2014.

[7] K. Nakano and E. Takamichi, “An Image Retrieval System using FPGAs,” *IEICE Transactions on Information and Systems*, vol. 86, no. 5, pp. 811–818, 2003.

[8] X. Zhou, Y. Ito, and K. Nakano, "An Efficient Implementation of the Hough Transform using DSP slices and block RAMs on the FPGA," in *Proceedings of IEEE 7th International Symposium on Embedded Multicore Socs (MCSoC)*, 2013, pp. 85–90.

[9] Y. Ago, K. Nakano, and Y. Ito, "A Classification Processor for a Support Vector Machine with embedded DSP slices and block RAMs in the FPGA," in *Proceedings of IEEE 7th International Symposium on Embedded Multicore Socs (MCSoC)*, 2013, pp. 91–96.

[10] X. Zhou, Y. Ito, and K. Nakano, "An Efficient Implementation of the Gradient-based Hough Transform using DSP slices and block RAMs on the FPGA," in *Proceedings of International Parallel and Distributed Processing Symposium Workshops*, 2014, pp. 762–770.

[11] K. Hashimoto, Y. Ito, and K. Nakano, "Template Matching using DSP slices on the FPGA," in *Proceedings of International Symposium on Computing and Networking (CANDAR)*, 2013, pp. 338–344.

[12] X. Zhou, Y. Ito, and K. Nakano, "An Efficient Implementation of the One-Dimensional Hough Transform Algorithm for Circle Detection on the FPGA," in *Proceedings of International Symposium on Computing and Networking (CANDAR)*, 2014, pp. 447–452.

[13] Helion Technology, *LZRW3 Data Compression Core for Xilinx FPGA*, October 2008.

[14] S. Navqi, R. Naqvi, R. A. Riaz, and F. Siddiqui, "Optimized RTL design and implementation of LZW algorithm for high bandwidth applications," *PRZEGLAD ELEKTROTECHNICZNY (Electrical Review)*, vol. 4, pp. 279–285, 2011.

[15] M. Lin, "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*, vol. 26, no. 3, pp. 369–381, 2000.

[16] M. Lin, J. Lee, and G. E. Jan, "A Lossless Data Compression and Decompression Algorithm and Its Hardware Architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 9, pp. 925–936, 2006.

[17] S. Prakash, M. Purohit, and A. Raizada, "A novel approach of speedy-highly secured data transmission using cascading of PDLZW and arithmetic coding with cryptography," *International Journal of Computer Applications*, vol. 57, no. 19, 2012.

[18] S. Funasaka, K. Nakano, and Y. Ito, "A parallel algorithm for LZW decompression, with GPU implementation," in *to appear in Proc. of International Conference on Parallel Processing and Applied Mathematics*, 2015.

[19] K. Shyni and K. V. M. Kumar, "Lossless LZW data compression algorithm on CUDA," *IOSR Journal of Computer Engineering*, pp. 122–127, 2013.

[20] S. T. Klein and Y. Wiseman, "Parallel Lempel Ziv coding," *Discrete Applied Mathematics*, vol. 146, no. 2, pp. 180–191, 2005.

[21] M. K. Mishra, T. K. Mishra, and A. K. Pani, "Parallel Lempel-Ziv-Welch (PLZW) technique for data compression," *International Journal of Computer Science and Information Technologies*, vol. 3, no. 3, pp. 4038–4040, 2012.

[22] Xilinx Inc., *7 Series FPGAs Configuration User Guide*, 2013.

[23] ——, *VC707 Evaluation Board for the Virtex-7 FPGA User Guide*, 2014.