Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186–5140
Volume 5, Number 1, pages 12–19, January 2016

An Efficient Implementation of LZW Decompression Using Block RAMs in the FPGA (Preliminary Version)

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

***Abstract*—LZW algorithm is one of the most important compression and decompression algorithms. The main contribution of this paper is to present an efficient hardware architecture of LZW decompression algorithm and to implement it in an FPGA. In our implementation, the codes of a compressed file is read one by one, and the dictionary table is continuously updated until the table is full. For each code of the compressed file, an inverse of string corresponding to this code is sequentially written to an output buffer. The length of this string and the address of the forefront of this string is stored. The inverse of string can be output reversely from the output buffer using the stored length and forefront address. Since output buffer uses dual-port block RAMs, input of the inverse strings and output of the original strings are performed in parallel. The experimental results show that our FPGA module of LZW decompression on Virtex-7 family FPGA uses 287 slice registers, 282 slice LUTs and 7 block RAMs with 36k-bit. One LZW decompression module is more than 2 times faster than sequential LZW decompression on a single CPU. Since the proposed FPGA module uses a few resources of the FPGA, we implement 34 LZW decompression modules which works in parallel in the FPGA. In other words, our implementation runs up to 64 times faster than sequential LZW decompression on a single CPU.**

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

I. Introduction

Data compression is a method of encoding rules that substantially reduces the total memory space to store or transmit a file in digital communications and data processing. Two kinds of data compression are used in different areas. One of these is lossy data compression that is commonly used to compress images. Some details of the image is losed and can never be recovered. Therefore, decompression of lossy compressed images does not recovers the same images. The other is lossless data compression that preserves all information of the original file. We can obtain the intact files by the decompression of lossless compressed files. The most famous compression algorithms are LZ-based algorithms such as LZ77 [1], LZ78 [2] and LZW [3]. LZ77 algorithm is to find the longest string of pending part of a file, that matches to the string in processed part of the file. Hence, it is not well for large volumes of arbitrary data. LZ78 algorithm creates a dictionary table that stores unmatched strings. LZ78 algorithm outperforms LZ77 algorithm for compression of large volumes of arbitrary data [4]. The speed of LZ78 algorithm depends on finding the longest matching string from the dictionary table. However, not all of the strings stored in the table have the same length, it wastes a lots of memory spaces in hardware implementation. LZW algorithm is proposed to reduce the large memory required in hardware implementation of LZ78 algorithm. The dictionary of LZW algorithm is initialized with the underlying character set and built up from top to buttom. Each entry of dictionary of LZW algorithm represents a string. Instead of storing entire string in the dictionary, each entry includes one character and one pointer . The character is the last character of corresponding string. The pointer points to an entry which represents the string excluding the last character. We can obtain the inverse of string by recursively accessing the entries of the dictionary depending on stored pointers. In this paper, we focus on LZW compression which is used in Unix unility $compress$ and in GIF image format. LZW compression is included in TIFF file format standard [5], 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. The main goal of this paper is to develop an efficient hardware architecture to maintain the dictionary 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 [6]. Recent FPGAs have embedded DSP slices and block RAMs. The Xilinx Virtex-7 family FPGAs have DSP slices, each of which is equipped with a multiplier, adders/subtracters, logic operators, registers, etc [7]. A DSP slice also has pipeline registers between operators to reduce the propagation time. 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 [8]. 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 [9], [10], [11], [12], [13], [14].

There is some research for accelerating LZW decompression which are implemented on hardware device as FPGAs or VLSI. Various accelerators have been proposed by different manufacturers. LZWR3 Core by Helion [15] provides a decompression rate of 180.75MBytes/s clocked at 226MHz in Xilinx Virtex-5 FPGA. Lin [16] gives a comparative decompression rate within the range 25∼83MBytes/s clocked at 100MHz, where parallel fixed-length dictionary tables are used. Navqi [17] provides a decompression rate of 140∼160MBytes/s clocked at 50MHz in Xilinx Virtex-2 FPGA. For each compressed image which is stored in a file, it is decompressed while the original image is used Hence. decompression is performed more frequently than compression, we say that LZW decompression is more important than the compression.

The main contribution of this paper is to present an efficient hardware LZW decompression algorithm and to implement it in an FPGA. Our FPGA module of hardware LZW decompression on Virtex-7 family FPGA uses uses 287 slice registers, 282 slice LUTs and 14 block RAMs with 18k-bit. In our implementation, the updating of dictionary tables is continuously performed every 2 clock cycles, while the operation of writting out characters is performed in parallel. Since updating of dictionary tables only depends on the input compressed codes and faster than the operation of writting out characters, the original characters are output by traversing the dictionary tables without validating whether the coresponding entry of dictionary has been updated or not. The experimental resultls show that our hardware LZW decompression module runs 2.1 times faster than sequential LZW decompression on a single CPU. Since the decompression rate is data dependent, according to the experimental results, the decompression rate of our module is up to 279.84MBytes/s while the compression ratio of input file is high. Even if the compression ratio is low, our module still has a decompression rate of 183.38MBytes/s. Since the proposed FPGA module uses only a few resources of the FPGA, we implement 34 LZW decompression modules in FPGA, where all modules works in parallel. Our implementation of 34 paralleled modules runs up to 64 times faster than sequential LZW decompression on a single CPU.

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

II. LZW COMPRESSION AND DECOMPRESSION

The main purpose of this section is to review LZW compression and decompression algorithms. Please see Section 13 in [5] for details.

The LZW (Lempei-Ziv-Welch) [3] lossless data compression algorithm gives high compression efficiency. In this algorithm, an input string of characters is compressed into a series of codes using a dictionary table that maps string into fixed-length codes. The dictionary of LZW algorithm is initialized with the underlying character set and built up from top to buttom. If the input file is an image, the underlying character set may be 0∼255 represented by 8-bit. In general, the compression ratio is improved as the address space of dictionary table increases. However, the total number of entries of the dictionary is always 4096 since the improvement is of little significance beyond this number. The LZW compression algorithm reads characters of input file one by one and search for the longest matched string in the dictionary table. It writes the index of entry of the matched string as the output code. Subsequently, it concatenates the matched string and the next character to add it into the dictionary as a new entry. Let $X = x_0x_1 \cdots x_{n-1}$ be the characters of input file and $Y = y_0y_1 \cdots y_{m-1}$ be compressed codes. For simplicity, we assume that an input string includes 4 characters a, b, c and d. Let S be a string table which maps a string to a code, where the code corresponds to index of table. The string table S is initialized as $S(a) = 0$, $S(b) = 1$, $S(c) = 2$ and $S(d) = 3$. New code is assigned to a string by performing "AddTable" operation. For example, after initialization of S, if AddTable(cb) is performed, $S(cb) = 4$ holds. The LZW compression algorithm is described as follows:

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

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

Table I shows the compression flow of an input string "$cbcbcbcda$". First, "$x_0 = c$" is read from input file. Since $\Omega \parallel x_0 = c$ is in S, $\Omega \leftarrow c$ is performed. Subsequently, the next character "$x_1 = b$" is input. Since $\Omega \parallel x_1 = cb$ is not in S. $S(c) = 2$ is output as a code. Moreover, $\Omega \parallel x_1 = cb$ is mapped to 4, more specifically, $S(cb) = 4$ holds. $\Omega \leftarrow x_1 = b$ is performed. In the same way, we can confirm that series of codes 214630 is output as well as $S(cb) = 4, S(bc) = 5, S(cbc) = 6, S(cbcd) = 7, S(da) = 8$ are added to table.

We show LZW decompression algorithm on which we focus in this paper. Let C be the dictionary of decompression

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
9	–	a	–	0

which is the inverse of table S. For example, if $S(cb) = 4$, $C(4) = cb$ holds. Table C is initialized as $C(0) = a$, $C(1) = b$, $C(2) = c$ and $C(3) = d$. Let $C_f(i)$ denote the first character of corresponding string of code i. For example, $C_f(4) = c$ while $C(4) = cb$ holds. The LZW decompression algorithm reads a series Y of codes one by one and adds an entry of the table. It writes a string X at the same time. The LZW decompression algorithm is described as follows:

[LZW decompression algorithm]
Output($C(y_0)$);
for $i \leftarrow 1$ to $m-1$ do
  if(y_i is in C)
    begin
      Output($C(y_i)$);
      AddTable($C(y_{i-1}) \parallel C_f(y_i)$);
    end
  else
    begin
      Output($C(y_{i-1}) \parallel C_f(y_{i-1})$);
      AddTable($C(y_{i-1}) \parallel C_f(y_{i-1})$);
    end

Table II shows the decompression flow for a series of codes 214630. The character($C(2) = c$) of the first code "2" is output. Next, the code $y_1 = 1$ is input. Since $y_1 = 1$ is in C, $C(1) = b$ is output and string $C(y_0) \parallel C_f(y_1) = cb$ is added to $C(4)$. Then, next code $y_2 = 4$ is input. Since $y_2 = 4$ is in C, $C(4) = cb$ is output and string $C(y_1) \parallel C_f(y_2) = bc$ is added to $C(5)$. Next, since $y_3 = 6$ is not in C, $C(y_2) \parallel C_f(y_2) = cbc$ is output and string cbc is added to $C(6)$. It is simple to confirm that string $cbcbcbcda$ is output as well as $cb, bc, cbc, cbcd, da$ are added to table.

We will modify LZW decompression to implement it in the FPGA. We define several notations before showing the hardware LZW decompression algorithm. We assume that $X = x_0x_1 \cdots x_{n-1}$ is a string of characters that are selected from a underlying alphabet set, where the alphabet set consists of k characters $\alpha(0)$, $\alpha(1)$, …, $\alpha(k-1)$. The same as the example above, we assume that $k = 4$ holds as

Table II
LZW DECOMPRESSION FLOW FOR INPUT CODES $Y = 214630$

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

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

i	$p(i)$	$C_f(i)$	L(i)	C
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	-	-

4 characters $\alpha(0) = a$, $\alpha(1) = b$, $\alpha(2) = c$ and $\alpha(3) = d$. Let $Y = y_0y_1 \cdots y_{m-1}$ denote the compressed series of codes. Each of the first $m-1$ codes y_0, y_1, $\cdots$, y_{m-2} has a corresponding AddTable operation such that the number of entries of table C is $k+m-1$. Let p denote the pointer table using input code Y:

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

We traverse pointer table p until reaching NULL.

Let $p^0(i) = i$ and $p^{j+1}(i) = p(p^j(i))$ for all $j \ge 0$ and i. More specifically, $p^j(i)$ is the code where we reach from code i in jth pointer traversing operation. Let $L(i)$ be an integer that satisfies $p^{L(i)}(i)$ =NULL and $p^{L(i)-1}(i) \neq$NULL. Let C_f be a character table defined as follows:

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

We note that $C_f(i)$ represents the first character of $C(i)$. Also, since each code of $y_i(k \le i \le k+m-1)$ has a corresponding AddTable operation as shown in LZW decompression before, we must notice that if $k \le i \le k+m-2$, the last character of $C(i)$ equals to $C_f(i+1)$. Hence, we can define string $C(i)$ as follows:

$$C(i) = \begin{cases} C_f(i), & \text{if}(0 \le i \le k-1) \\ T(i), & \text{if}(k \le i \le k+m-2) \end{cases} \quad (3)$$

where $T(i)$ denotes $C_f(p^{L(i)-1}(i)) \parallel C_f(p^{L(i)-2}(i)+1) \cdots C_f(p^0(i)+1)$. Table III shows the values of p, C_f, L and C if $Y = 214630$.

Next, we show the flow of the hardware LZW decompression which can be implemented in a FPGA. This algorithm is performed with 3 parts as follows:

Part 1 Update dictionary table
Part 2 write inverse of corresponding string to memory
Part 3 read string reversely from memory to output

Part 1 of hardware LZW decompression algorithm updates pointer table p and character table C_f sequentially. For i($0 \leq i \leq k-1$), p and C_f are initialized which are configured as virtual memory space occpuying any memories resources. For i($k \leq i \leq k+m-1$), p is updated with the values of input codes. The first characters of $C(i)$ are obtained to update table C_f. We show Part 1 of hardware LZW decompression algorithm as follows:

[Part 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
  $p(i) \leftarrow y_{i-k}$; $C_f(i) \leftarrow C_f(p^1(i))$;

In Part 2 of hardware LZW decompression algorithm, for each compressed code $y_i(0 \leq i \leq m-1)$ of Y, characters of corresponding string $C(i)$ is reversely read out from table C_f by traversing pointer table p. At the same time with traversing operation, the length of string is also computed and stored in $L(i)$. For example, if $C(4) = cb$ holds, cb is read out in the order as $b \rightarrow c$. The inverse of strings are sequentially written to an output memory as a buffer. Let b denote this output memory and $addr$ denote the current writting address of it. The writting operation of b is from top to bottom. While the last character of inverse of string for one code is written to b, the length $L(i)$ and current address $addr$ are stored in another table defined as t, where $addr$ now represents the address of the first character of $C(i)$. The details of Part 2 of hardware LZW decompression algorithm are shown as follows:

[Part 2 of hardware LZW decompression algorithm]
for $i \leftarrow 0$ to $m-1$ do
  $j \leftarrow y_i$;
  while($p(j) \neq$ NULL)
    $b[addr] \leftarrow C_f(j+1)$; $j \leftarrow p(j)$;
    $L(i) \leftarrow L(i)+1$; $addr \leftarrow addr+1$;
  $b[addr] \leftarrow C_f(j)$; $L(i) \leftarrow L(i)+1$;
  $t[i] \leftarrow \{L(i), addr\}$; $addr \leftarrow addr+1$;

In Part 3 of hardware LZW decompression algorithm, the length $L(i)$ and address $addr$ of the first of string are read from table t one by one. Then, for each pair of $addr$ and $L(i)$, we read string reversely from $addr$ until $L(i)$ characters are all read out from output memory b. The reading operation of b is bottom to top which is opposite to writting operation of it. Part 3 is shown as follows:

[Part 3 of hardware LZW decompression algorithm]
for $i \leftarrow 0$ to $m-1$ do
  $\{L(i), addr\} \leftarrow t[i]$;
  while($L(i) \neq 0$)
    Output($b[addr]$);
    $L(i) \leftarrow L(i)-1$; $addr \leftarrow addr-1$;

In our implementation, we use dual-port mode block RAMs of FPGA to implement the tables p, C_f, b. Hence, the writting and reading operations of these tables can be performed concurrently. In other words, the operation of 3 Parts of hardware LZW decompression algorithm are implemented in parallel that we show this in the next section.

III. Our FPGA architecture for LZW decompression

This section describes our FPGA architecture for 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 [18].

In this paper, we focus on the decompression of a gray-scale image with 8-bit intensity level. First, we show the compression of the image. The input file contains a string of integers within the range $[0, 255]$. Therefore, the dictionary table of LZW compression is initialized with the underlying integers set $[0, 255]$. Also, code 256 is reserved as "ClearCode" which denotes to clear the table. Code 257 is reserved as "EndOfInformation" which represents the end of the input file. Then, dictionary table is built up from 258 by performing AddTable operation. We configure the size of table to be 4096 in this paper. Hence, "ClearCode" is output after the entry 4095 is added to the table. Subsequently, the table is initialized and bulit up from 258 again. The same operation is repeated until all pixels of input image are converted to codes. "EndOfInformation" is output after the last pixel of input image is compressed to a code. We note that a string of codes is seperated by "ClearCode" which is called a code segment. Each code segment has $4096-258 = 3838$ codes except the last one which has less codes.

In our implementation, our LZW-based module decompresses all code segments one by one. This module contains pointer table p and character table C_f. First, a virtual memory space with the range $[0, 257]$ is reserved. The tables take up the address space but do not actually occupy any part of table hardware. Since table p and C_f both have 3838 entries, the size of table p is $12 \times 3838 = 44.97$K-bit and the size of table C_f is $8 \times 3838 = 29.98$K-bit. Hence, we use 3 and 2 18K-bit block RAMs to implement pointer table p and character table C_f, respectively. Moreover, we use these block RAMs as dual-port mode memory [8] as shown in Figure 1. A dual-port block RAM has two set of ports

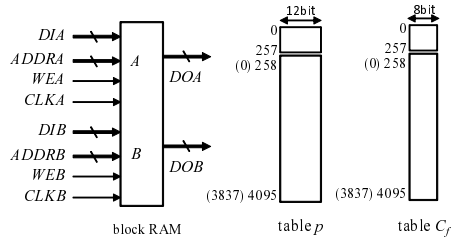

Figure 1. Dual-port block RAM and memory configurations of table p and C_f

such as Port A and Port B which work independently. The architecture of our LZW decompression module is shown in Figure 2. Since the updating of table p and C_f is input code dependent, we show it first. For table p, Port B of it is used to write all codes of Y into the block RAM one by one, where the codes are used as pointers during traversing operation. We notice that codes of Y should have been written into the block RAM from entry 258. Since a virtual memory space with the range $[0, 257]$ is considered, the table is actually built up from entry 0 to bottom. More specifically, y_0 is written into entry 0 of table p. For table C_f, Port A of it is used to update this table. The same as the table p, $C_f(i)(0 \leq i \leq 257)$ is reserved as virtual memory space. C_f is also built up from entry 0. More specifically, $C_f(258)$ is written into entry 0 of table C_f. As shown in previous section, if $0 \leq i \leq 257$, $C_f(i) = i$ holds. Also, if $258 \leq i \leq 4095$, $C_f(i) = C_f(p(i)) = C_f(y_{i-258})$ holds. For example, we want to update $C_f(i)(i \geq 258)$ which corresponds to y_{i-258}. If $y_{i-258} < 258$, $C_f(y_{i-258}) = y_{i-258}$ holds, then we update this table with y_{i-258}. Otherwise, if $y_{i-258} \geq 258$, we need to read out the value of entry y_{i-258} from table C_f. Actually, since memory space $[0, 257]$ is reserved, we read out $C_f(y_{i-258} - 258)$ to update this table. As shown in Figure 2, we write the new entry through port *DIA* of table C_f using a multiplexer. The multiplexer select the value between the input code y_{i-258} and output $C_f(y_{i-258} - 258)$ of port *DOA*. Since updating table p and C_f only depends on the input code without traversing the tables, all entries of two tables are concurrently updated one by one. It takes 2 clock cycles to update each entry until the last code of one code segment is input. In other words, all input codes are input once every 2 clock cycles.

Next, we will show how to write an inverse of string to output buffer by traversing table p and C_f. Since traversing operation of table p and C_f is performed for each code, it takes more than 2 clock cycles to obtain the inverse of string for one code. Hence, we use a code buffer to store the input codes and to process them one by one afterwards, where the code buffer is configured as a FIFO(First-In-First-Out) memory using block RAMs. We use an output buffer to write the inverse of strings in it. The output buffer is configured as dual-port block RAM as shown in Figure 1. After the inverse of string is written into the output buffer through Port B of it, we can obtain the decompressed characters by reading the inverse reversely through the Port A of it. If the code buffer is not empty, we read the first code from code buffer to output the corresponding string of this code. For example, for the code $y_i(0 \leq i \leq n-1)$ from code buffer, we can obtain an inverse of $C(y_i)$ which is the string corresponding to code y_i by traversing the table p and C_f. First, if $y_i < 258$, $C(y_i) = C_f(y_i) = y_i$ holds, and the length of string $C(y_i)$ must be 1. The string $C(y_i)$ with length 1 is written into output buffer through port *DIB* of the output buffer. Otherwise, if $y_i \geq 258$, we read string $C(y_i)$ reversely by traversing the table p and C_f and write it to output buffer. As shown in Figure 2, Port A of table p is used for traversing operation. If $y_i \geq 258$, $y_i - 258$ is computed and connected to port *ADDRA* of table p. The pointer $p^1(y_i - 258) = p(y_i - 258)$ will be read out and feed back to port *ADDRA* through a multiplexer. Concurrently, character $C_f(y_i - 258 + 1)$ is read out from table C_f through port *ADDRB* of table C_f which depends on port *ADDRA* of table p as shown in Figure. This character will be written into output buffer in the next clock cycle. Next, we see the table p again, if the pointer $p^1(y_i - 258) \geq 258$, it is clearly that the traversing operation has not reached the dead end. The pointer $p^1(y_i - 258)$ is fed back to port *ADDRA* through a multiplexer. Then, the next pointer $p^2(y_i - 258) = p(p^1(y_i - 258))$ will be read out from table p. Also, at the same time, depending on the pointer $p^1(y_i - 258)$, character $C_f(p^1(y_i - 258) + 1)$ is read out from table C_f. The traversing operation will not stop until the pointer $p^{L(i)-1}(y_i - 258)$ is read out, where $p(p^{L(i)-1}(y_i - 258))$ =NULL holds and $L(i)$ denotes the length of corresponding string. We use a counter to accumulate $L(i)$ during traversing operation. For traversing operation of code yi, it is clearly that characters $C_f(y_i - 258 + 1) \parallel C_f(p^1(y_i - 258) + 1) \parallel \cdots C_f(p^{L(i)-1}(y_i - 258) + 1)$ are written into the output buffer one by one. In other words, inverse of string $C(y_i)$ are store in output buffer from top to bottom. As soon as the last character of inverse of string $C(y_i)$ is written into output buffer, the length $L(i)$ and current writting address of output buffer are store in table t, where table t is configured as a FIFO memory using block RAMs.

For each code stored in code buffer, the traversing operation is performed to written inverse of the corresponding string to output buffer through port *DIB* of output buffer. Moreover, the length and address of first character of the string is stored in table t. Next, we show how to otuput strings reversely from output buffer using the other port of output buffer. First, if table t is not empty, we obtain $L(i)$ and $addr(i)$ from this table, where $L(i)$ is the length of string $C(y_i)$ of code y_i. Then, reading operation of output buffer starts at address $addr(i)$ through port *ADDRA*. $L(i)$

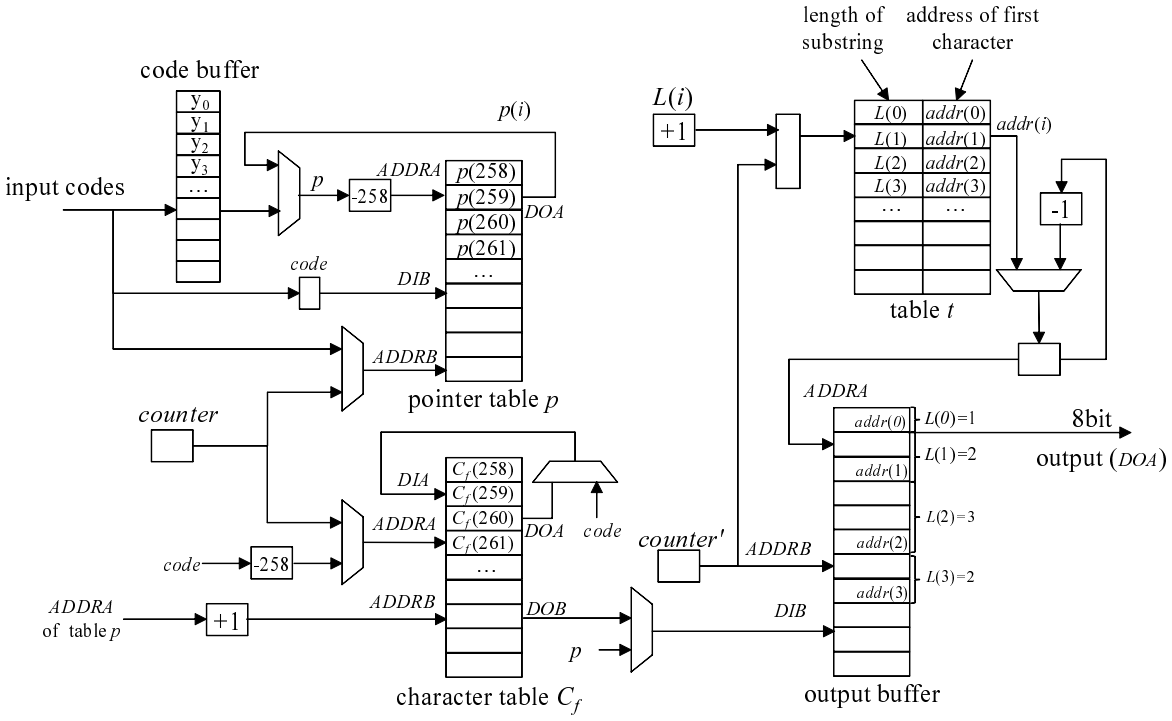

Figure 2. The outline of our FPGA architecture for hardware LZW algorithm

is decreased during the reading operation. The value of $addr(i)$ is decreased every clock cycle until $L(i)$ reaches 0. In other words, inverse of $C(y_i)$ is output reversely by the reading operation. Since output buffer is configured as dual-port block RAM memory, the writing and reading operation of output buffer are performed in parallel. Moreover, the updating of table p and C_f are also performed in parallel. If all codes of this code segment are decompressed, the codes of new code segment will input into the LZW decompression module. The same procedure is repeated without re-initializing all the tables. The module terminates as soon as all codes of input file are decompressed and then wait for the next input file.

In our implementation, each entry of table p and C_f is updated every 2 clock cycles. For a certain code stored in code buffer, an inverse of the corresponding string is written into output buffer continuously. In other words, if string contains L characters, it takes L clock cycles to write it into output buffer. If the string $C(y_i)$ of a certain code y_i is long, it takes many clock cycles to write the string into output buffer. And during the writting operation of code y_i, the next y_{i+1} can not be performed. However, the input codes are input every clock cycles. Therefore, the number of codes which are stored in code buffer will increase. To avoid the overflow of the code buffer, the size of code buffer is $12 \times 3838 = 44.97$K-bit which occupies 3 18K-bit block RAMs. We use 3 and 2 18K-bit block RAMs to implement the table p and C_f. The size of output buffer is $8 \times 3838 \times 2 = 59.97$K-bit that we use 4 18K-bit block RAMs to implement it. The depth of output buffer is described with 13-bit. Hence, the size of table t is $(12 + 13) \times 1280 = 31.25$K-bit. We use 2 18K-bit block RAMs to implement it. Totally, 14 18K-bit block RAMs are used in our implementation of hardware LZW decompression algorithm.

IV. 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 [19] equipped with the Xilinx Virtex-7 FPGA XC7VX485T-2. According to the implementation results, one LZW decompression module uses 287 slice registers, 283 slice LUTs and 14 18K-bit block RAMs. We implement 34 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 3, 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 ($\frac{original\ image\ size}{compressed\ image\ size}$).

"Crafts" "Flowers" "Graph"

Figure 3. Three gray scale image with 4096×3072 pixels

The image "Graph" has high compression ratio since it has large areas with simliar intensity levels. The image "Crafts" has small compression ratio since it has small details. Both CPU and FPGA decompression take more time to create dicitionary tables if the image has small compression ratio. In LZW decompression on CPU, the operation of creating dictionary tables occpuies most of the computing time. In our implementation on FPGA, the operation of creating tables is performed independently, and writting characters to output buffer and output characters from output buffer are paralleled, hence, the operation of outputing characters occpuies most of the time. As shown in Table V, even only one module of hardware LZW decompression algorithm is implemented for the time evaluation, the implementation on FPGA is still faster than on the CPU. For example, it takes 19674631 clock cycles to decompress image "Crafts", i.e., $\frac{19674631}{300.661MHz} = 65.438ms$. It takes 18339574 clock cycles to decompress image "Flowers", i.e., $\frac{18339574}{300.661MHz} = 60.998ms$. To decompress image "Graph", it only takes 12892927 clock cycles, i.e., $\frac{12892927}{300.661MHz} = 42.882ms$. Hence, for gray scale image "Graph" which has high compression ratio with 4096×3072 pixels, the LZW decompression module output 4096×3072×1Byte original data in $42.882ms$, we say the decompression rate of module is $\frac{4096\times3072\times1\text{Byte}}{42.882ms} = 279.84$MBytes/s. Since the proposed FPGA module uses a few resources of the FPGA, we implement 34 modules of hardware LZW decompression in a FPGA, where all modules work in parallel. In other words, our implementation runs up to 64 times faster than sequential LZW decompression on a single CPU.

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

number of modules	1	34	Available
Slice Registers	287 (0.05%)	9894 (1.63%)	607200
Slice LUTs	283 (0.09%)	9302 (3.06%)	303600
18K-bit block RAMs	14 (0.67%)	476 (23.1%)	2060
I/O	25 (3.57%)	564 (80.6%)	700
Clock frequency [MHz]	300.661	264.13	—

V. CONCLUSIONS

We have presented a hardware LZW decompression algorithm of decompressing images. It was implemented in an Virtex-7 family FPGA XC7VX485T-2. According to the implementation results, one LZW decompression module uses 287 slice registers, 283 slice LUTs and 14 18K-bit block RAMs, where these are very few of FPGA resources. By simulation, one FPGA module of LZW decompression is more than 2 times faster than sequential LZW decompression on a single CPU. We also implemented 34 LZW decompression modules in parallel which attains a speed up factor of 64 over the sequential implementation on the CPU. Our module provides a decompression rate up to 279.84MBytes/s which is higher than other research. Since the decompression rate is data dependent, the decompression rate can be even better if the compression rate of input file is higher.

Table V
EXPERIMENTAL RESULTS (MILLISECONDS) FOR THREE IMAGES

images	compression ratio	time of CPU	time of FPGA	Speedup ratio
"Crafts"	1.43:1	141.534	65.438	2.16:1
"Flowers"	1.72:1	127.136	60.998	2.08:1
"Graph"	36.72:1	75.9	42.882	1.77:1

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," *Information Theory, IEEE Transactions on*, vol. 24, no. 5, pp. 530–536, 1978.

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

[4] D. Craft, "Aldc and pre-processor extension, bldc, provide ultra fast compression for general-purpose and bit-mapped image data," in *dcc*. IEEE, 1995, p. 440.

[5] A. D. Association *et al.*, "Tiff revision 6.0," *Mountain View, Cal.: Adobe Systems*, 1992.

[6] K. Nakano and Y. Yamagishi, "Hardware n choose k counters with applications to the partial exhaustive search," *IEICE Trans. on Information & Systems*, 2005.

[7] Xilinx Inc., *7 Series DSP48E1 Slice User Guide*, Nov. 2014.

[8] ——, *7 Series FPGAs Memory Resources User Guide*, Nov. 2014.

[9] 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.

[10] X. Zhou, Y. Ito, and K. Nakano, "An efficient implementation of the hough transform using dsp slices and block rams on the fpga," in *Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium on*. IEEE, 2013, pp. 85–90.

[11] 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 *Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium on*. IEEE, 2013, pp. 91–96.

[12] 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 *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014, pp. 762–770.

[13] K. Hashimoto, Y. Ito, and K. Nakano, "Template matching using dsp slices on the fpga," in *Computing and Networking (CANDAR), 2013 First International Symposium on*. IEEE, 2013, pp. 338–344.

[14] X. Zhou, Y. Ito, and K. Nakano, "An efficient implementation of the one-dimensional hough transform algorithm for circle detection on the fpga," in *Computing and Networking (CANDAR), 2014 Second International Symposium on*. IEEE, 2014, pp. 447–452.

[15] Helion Technology, *LZRW3, Data Compression Core for Xilinx FPGA*, 2006.

[16] M.-B. Lin, J.-F. Lee, and G. E. Jan, "A lossless data compression and decompression algorithm and its hardware architecture," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 9, pp. 925–936, 2006.

[17] S. Navqi, R. Naqvi, R. A. Riaz, and F. Siddiqui, "Optimized rtl design and implementation of lzw algorithm for high bandwidth applications," *Electrical Review*, vol. 4, pp. 279–285, 2011.

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

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