LOSSLESS DATA COMPRESSION AND DECOMPRESSION ALGORITHM AND ITS HARDWARE ARCHITECTURE

V V V SAGAR[1]

[1]*JTO MPLS NOC BSNL BANGALORE*

Abstract - *LZW (Lempel Ziv Welch) and AH (Adaptive Huffman) algorithms were most widely used for lossless data compression. But both of these algorithms take more memory for hardware implementation. We discuss about the design of the two-stage hardware architecture with Parallel dictionary LZW algorithm first and Adaptive Huffman algorithm in the next stage. In this architecture, an ordered list instead of the tree based structure is used in the AH algorithm for speeding up the compression data rate. The resulting architecture shows that it not only outperforms the AH algorithm at the cost of only one-fourth the hardware resource but it is also competitive to the performance of LZW algorithm (compress). In addition, both compression and decompression rates of the proposed architecture are greater than those of the AH algorithm even in the case realized by software. The performance of the PDLZW algorithm is enhanced by incorporating it with the AH algorithm. The two stage algorithm is discussed to increase compression ratio with PDLZW algorithm in first stage and AHDB in second stage.*

Key Words: **PDLZW, AHDB, Verilog HDL language, Xilinx ISE 9.1, Synopsys**

1. INTRODUCTION

Data transmission and storage cost money. The more information being dealt with, the more it costs. In spite of this, most digital data are not stored in the most compact form. Rather, they are stored in whatever way makes them easiest to use, such as: ASCII text from word processors, binary code that can be executed on a computer, individual samples from a data acquisition system, etc. Typically, these easy-to-use encoding methods require data files about twice as large as actually needed to represent the information. Data compression is the general term for the various algorithms and programs developed to address this problem. A *compression program* is used to convert data from an easy-to-use format to one optimized for compactness. Likewise, an *uncompression program* returns the information to its original form.

A new two-stage hardware architecture is proposed that combines the features of both parallel dictionary LZW (PDLZW) and an approximated adaptive Huffman (AH) algorithms. In the proposed architecture, an ordered list instead of the tree based structure is used in the AH algorithm for speeding up the compression data rate. The resulting architecture shows that it outperforms the AH algorithm at the cost of only one-fourth the hardware resource, is only about 7% inferior to UNIX *compress* on the average cases, and outperforms the *compress* utility in some cases. The compress utility is an implementation of LZW algorithm.

2. PDLZW Algorithm

The major feature of conventional implementations of the LZW data compression algorithms is that they usually use only one fixed-word-width dictionary. Hence, a quite lot of compression time is wasted in searching the large-address-space dictionary instead of using a unique fixed-word-width dictionary a hierarchical variable-word-width dictionary set containing several small address space dictionaries with increasing word widths is used for the compression algorithm. The results show that the new architecture not only can be easily implemented in VLSI technology due to its high regularity but also has faster compression rate since it no longer needs to search the dictionary recursively as the conventional implementations do.

Lossless data compression algorithms include mainly LZ codes **[5, 6]. A** most popular version of LZ algorithm is called LZW algorithm **[4].** However, it requires quite a lot of time to adjust the dictionary. To improve this, two alternative versions of LZW were proposed. These are DLZW (dynamic LZW) and WDLZW (word-based DLZW) **[5].** Both improve LZW algorithm in the following ways. First, it initializes the dictionary with different combinations of characters instead of single character of the underlying character set. Second, it uses a hierarchy of dictionaries with successively increasing word widths. Third, each entry associates a frequency counter. That is, it implements LRU policy. It was shown that both algorithms outperform LZW **[4].** However, it also complicates the hardware control logic.

In order to reduce the hardware cost, a simplified DLZW architecture suited for VLSI realization called PDLZW (parallel dictionary LZW) architecture. This architecture improves and modifies the features of both LZW and DLZW algorithms in the following ways. First, instead of initializing the dictionary with single character or different combinations of characters a virtual dictionary with the initial $|\Sigma|$ address space is reserved. This dictionary only takes up a part of address space but costs no hardware. Second, a hierarchical parallel dictionary set with successively increasing word widths is used. Third, the simplest dictionary update policy called FIFO (first-in first-out) is used to simplify the hardware implementation. The resulting architecture shows that it outperforms Huffman algorithm in all cases and about only **5%** below UNIX *compress* on the average case but in some cases outperforms the *compress* utility.

2.1 Dictionary Design Considerations

The dictionary used in PDLZW compression algorithm is one that consists of *m* small variable-word width dictionaries, numbered from 0 to *m* - 1, with each of which increases its word width by one byte. That is to say, dictionary 0 has one byte word width, dictionary 1 two bytes, and so on. These dictionaries: constitute a dictionary set. In general, different address space distributions of the dictionary set will present significantly distinct performance of the PDLZW compression algorithm. However, the optimal distribution is strongly dependent on the actual input data files. Different data, profiles have their own optimal address space distributions. Therefore, in order to find a more general distribution, several different kinds of data samples are: run with various partitions of a given address space. Each partition corresponds to a dictionary set. For instance, the 1K address space is partitioned into ten different combinations and hence ten dictionary sets. An important consideration for hardware implementation is the required dictionary address space that dominates the chip cost for achieving an acceptable compression ratio.

2.2. Compression processor architecture

In the conventional dictionary implementations of LZW algorithm, they use a unique and large address space dictionary so that the search time of the dictionary is quite long even with CAM (content addressable memory). In our design the unique dictionary is replaced with a dictionary set consisting of several smaller dictionaries with different address spaces and word widths. As doing so the dictionary set not only has small lookup time but also can operate in parallel.

The architecture of PDLZW compression processor is depicted in Figure 1. It consists of CAMs, an 5- byte shift register, a shift and update control, and a codeword output circuit. The word widths of CAMs increase gradually from 2 bytes up to 5 bytes with 5 different address spaces: 256, 64, 32, 8 and 8 words. The input string is shifted into the 5-byte shift register. The shift operation can be implemented by barrel shifter for achieving a faster speed. Thus there are 5 bytes can be searched from all CAMs simultaneously. In general, it is possible that there are several dictionaries in the dictionary set matched with the incoming string at the same time with different string lengths. The matched address within a dictionary along with the dictionary number of the dictionary that has largest number of bytes matched is outputted as the output codeword, which is detected and combined by the priority encoder. The maximum length string matched along with the next character is then written into the next entry pointed by the update pointer (UP) of the next dictionary (CAM) enabled by the shift and dictionary update control circuit. Each dictionary has its own UP that always points to the word to be inserted next. Each update pointer counts from 0 up to its maximum value and then back to 0. Hence, the FIFO update policy is realized. The update operation is inhibited if the next dictionary number is greater than or equal to the maximum dictionary number.

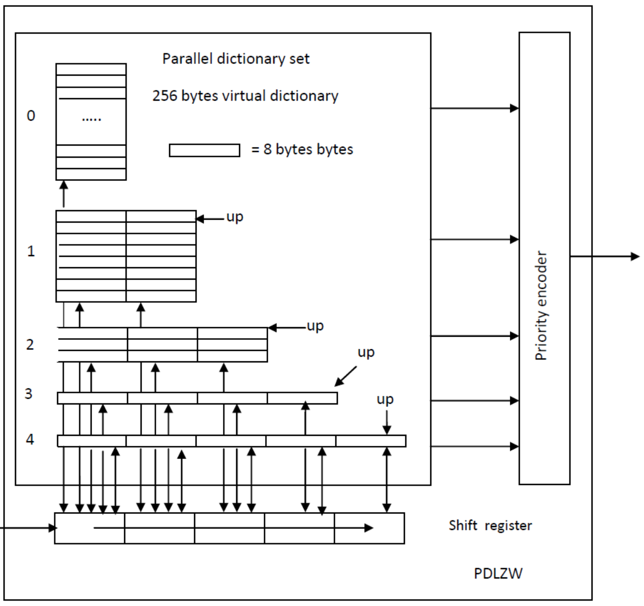

Fig- 1 PDLZW Architecture for compression

The data rate for the PDLZW compression processor is at least one byte per memory cycle. The memory cycle is mainly determined by the cycle time of CAMs but it is quite small since the maximum capacity of CAMs is only 256 words. Therefore, a very high data rate can be expected.

2.3 PDLZW Algorithms

Like the LZW algorithm proposed in [17], the PDLZW algorithm proposed in [9] also encounters the special case in the decompression end. In this paper, we remove the special case by deferring the update operation of the matched dictionary one step in the compression end so that the dictionaries in both compression and decompression ends can operate synchronously. The detailed operations of the PDLZW algorithm can be referred to in [9]. In the following, we consider only the new version of the PDLZW algorithm.

2.4 PDLZW Compression Algorithm:

As described in [9] and [12], the PDLZW compression algorithm is based on a parallel dictionary set that consists of m small variable-word-width dictionaries, numbered from 0 to m-1 , each of which increases its word width by one byte. More precisely, dictionary 0 has one byte word width, dictionary 1 two bytes, and so on. The actual size of the dictionary set used in a given application can be determined by the information correlation property of the application. To facilitate a general PDLZW architecture for a variety of applications, it is necessary to do a lot of simulations for exploring information correlation property of these applications so that an optimal dictionary set can be determined. The detailed operation of the proposed PDLZW compression algorithm is described as follows. In the algorithm, two variables and one constant are used. The constant max_*dict_no* denotes the maximum number of dictionaries, excluding the first single-character dictionary (i.e., dictionary 0), in the dictionary set. The variable *max_matched_dict_no* is the largest dictionary number of all matched dictionaries and the variable *matched_addr* identifies the matched address within the *max_matched_dict_no* dictionary. Each compressed codeword is a concatenation of *max_matched_dict_no* and *matched_addr*.

Algorithm: PDLZW Compression

Input: The string to be compressed.
Output: The compressed codewords with each having log2K bits. Each codeword consists of two components: *max_matched_dic_no* and *matched_addr*, where *K* is the total number of entries of the dictionary set.

Begin

```
1: Initialization.
1.1: string-1= null.
1.2: max_matched_dic_no =max_dict_no.
1.3: update_dict_no = max_matched_dict_no
        update_string= Ø {empty}.
2: while (the input buffer is not empty) do
2.1: Prepare next max_dict_no +1characters for searching.
2.1.1: string-2 = read next.
(max_matched_dict_no +1) characters from the input buffer.
2.1.2: string = string-1 || string -2.
{Where || is the concatenation operator}
2.2 Search string in all dictionaries in parallel and set the max_matched_dict_no and matched_addr.
2.3: Output the compressed codeword containing max_matched_dict_no || matched_addr.
2.4: if (max_matched_dict_no < max_dict_no and update_string ≠ Ø ) then
add the update_string to the entry pointed by UP [update_dict_no] of dictionary [update_dict_no].
 {UP [update_dict_no] is the update pointer associated with the dictionary}
2.5 Update the update pointer of the dictionary [max_matched_dict_no + 1].
2.5.1 UP [max_matched_dict_no + 1] = UP [max_matched_dict_no + 1] + 1
2.5.2 if UP[max_matched_dict_no + 1] reaches its upper bound then reset it to 0. {FIFO update rule.}
2.6: update_string =extract out the first (max_matched_dict_no + 2)
Bytes from string;
update_string_no= max_matched_dict_no + 1 .
2.7: string -1= shift string out the first (max_matched_dict_no + 1) bytes.
End {End of PDLZW Compression Algorithm.}
```

2.5 PDLZW Decompression Algorithm:

To recover the original string from the compressed one, we reverse the operation of the PDLZW compression algorithm. This operation is called the PDLZW decompression algorithm. By decompressing the original substrings from the input compressed codewords, each input compressed codeword is used to read out the original substring from the dictionary set. To do this without loss of any information, it is necessary to keep the dictionary sets used in both algorithms, the same contents. Hence, the substring concatenated of the last output substring with its first character is used as the current output substring and is the next entry to be inserted into the dictionary set. The detailed operation of the PDLZW decompression algorithm is described as follows. In the algorithm, three variables and one constant are used. As in the PDLZW compression algorithm, the constant max_*dict_no* denotes the maximum number of dictionaries in the dictionary set. The variable *last_dict_no* memorizes the dictionary address part of the previous codeword. The variable *last_output* keeps the decompressed substring of the previous codeword, while the variable *current_output* records the current decompressed substring. The output substring always takes from the *last_output* that is updated by *current_output* in turn.

Algorithm: PDLZW Decompression

Input: The compressed codewords with each containing *log 2 K* bits, where is the total number of entries of the dictionary set.
Output: The original string.

Begin

```
1: Initialization.
1.1: if ( input buffer is not empty) then
current_output= empty; last_output= empty;
addr= read next log2 k codeword from input buffer.
{where codeword = dict_no || dict_addr and || is the concatenation operator.}
1.2 if (dictionary[addr] is defined ) then
current_output = dictionary[addr];
last_output= current_output;
output = last_output;
update_dict_no= dict_no[addr] + 1
2: while (the input buffer is not empty) do
```

```
2.1: addr= read next log2k bit codeword from input buffer.
2.2{output decompressed string and update the
associated dictionary.}
2.2.1: current_output= dictionary[addr];
2.2.2: if(max_dict_no update_dict_no) then
add (last_output || the first character of current_output) to
the entry pointed by
UP[update_dict_no] of dicitionary [update_dict_no];
2.2.3: UP[update_dict_no] =UP[update_dict_no] + 1 .
2.2.4: if UP[update_dict_no] reaches its upper bound then
reset it to 0.
2.2.5: last_output =current_output;
Output= last_output;
update_dict_no= dict_no[addr] + 1
End {End of PDLZW Decompression Algorithm. }
```

3. Two Stage Architecture

The output code words from the PDLZW algorithm are not uniformly distributed but each codeword has its own occurrence frequency, depending on the input data statistics. Hence, it is reasonable to use another algorithm to encode statistically the fixed-length code output from the PDLZW algorithm into a variable-length one. As seen in figure 4.3 because of using only PDLZW algorithm for different dictionary size sometimes the compression ratio may decrease as dictionary size increase for particular address space. This irregularity can also be removed by using AH in the second stage. Up to now, one of the most commonly used algorithms for converting a fixed-length code into its corresponding variable-length one is the AH algorithm. However, it is not easily realized in VLSI technology since the frequency count associated with each symbol requires a lot of hardware and needs much time to maintain. Consequently, in what follows, we will discuss some approximated schemes and detail their features.

```
Algorithm: AHDB
 Input: The compressed codewords from PDLZW
algorithm.
Output: The compressed codewords.

Begin

1: Input pdlzw_output;
2: while (pdlzw_output!= null)
2.1: matched_index =search_ordered_list(pdlzw_output);
2.2: swapped_block
=determine_which_block_to_be_swapped(matched_index);
2.3: if (swapped_block!=k) then
2.3.1:swap(ordered_list[matched_index],ordered_list[point
er_of_swapped_block]);
2.3.2: pointer_of_swapped_block=
pointer_of_swapped_block + 1;
2.3.3: reset_check(pointer_of_swapped_block); {Divide the
pointer_of_swapped_block by two (or reset ) when it
reaches a threshold.}
 else
2.3.4: if( matched_index!=0 ) then
Swap(list[matched_index],list[matched_index - 1] ) ;
2.4: Input pdlzw_output;
End {End of AHDB Algorithm. }
```

3.1 Performance of PDLZW + AHDB

As described previously, the performance of the PDLZW algorithm can be enhanced by incorporating it with the AH algorithm, as verified from Fig. 4.3. The percentage of data reduction increases more than 5% in all address spaces from 272 to 4096. This implies that one can use a smaller dictionary size in the PDLZW algorithm if the memory size is limited and then use the AH algorithm as the second stage to compensate the loss of the percentage of data reduction. From both Figs. 4.3 and 4.4 , we can conclude that incorporating the AH algorithm as the second stage not only increases the performance of the PDLZW algorithm but also compensates the percentage of data reduction loss due to the anomaly phenomenon occurred in the PDLZW algorithm. In addition, the proposed scheme is actually a parameterized compression algorithm because its performance varies with different dictionary- set sizes but the architecture remains the same. Furthermore, our design has an attractive feature: although simple and, hence, fast but still very efficient, which makes this architecture very suitable for VLSI technology. The performance in percentage of data reduction of various partitions using the 368- address dictionary of the PDLZW algorithm followed by the AHDB algorithm is shown in Tables VI and VII. The percentage of data reduction and memory cost of various partitions using a 368-address dictionary PDLZW algorithm followed by the AHDB algorithm is depicted in Table VIII. To illustrate our design, in what follows, we will use the PDLZW compression algorithm with the 368-address dictionary set as the first stage and the AHDB as the second stage to constitute the two-stage compression processor. The decompression processor is conceptually the reverse of the compression. Counter part and uses the same data path. As a consequence, we will not address its operation in detail in the rest of the paper.

3.2 PROPOSED DATA COMPRESSION ARCHITECTURE

In this section, we will show an example to illustrate the hardware architecture of the proposed two-stage compression scheme. The proposed two-stage architecture consists of two major components: a PDLZW processor and an AHDB processor, as shown in Fig. 6. The former is composed of a dictionary set with partition {256, 64, 32, 8, and 8}. Thus, the total memory required in the processor is 296 B (= 64×2 + 32×3 + 8×4 + 8×5) only. The latter is centered around an ordered list and requires a content addressable memory (CAM) of 414 B (=368 × 9B).Therefore, the total memory used is a 710-B CAM.

3.3 PDLZW Processor

The major components of the PDLZW processor are CAMs, a 5-B shift register, and a priority encoder. The word widths of CAMs increase gradually from 2 to 5 B with four different address spaces: 64, 32, 8, and 8 words, as portrayed in Fig. 6. The input string is shifted into the 5-B shift register. Once in the shift register the search operation can be carried out in parallel on the dictionary set. The address along with a matched signal within a dictionary containing the prefix substring of the incoming string is output to the priority encoder for encoding the output codeword *pdlzw_output,* , which is the compressed codeword output from the PDLZW processor. This codeword is then encoded into canonical Huffman code by the AHDB processor. In general, it is not impossible that many (up to five) dictionaries in the dictionary set containing prefix substrings of different lengths of the incoming string simultaneously. In this case, the prefix substring of maximum length is picked out and the matched address within its dictionary along with the matched signal of the dictionary is encoded and output to the AHDB processor.

In order to realize the update operation of the dictionary set, each dictionary in the dictionary set except the dictionary 0 has its own update pointer (UP) that always points to the word to be inserted next. The update operation of the dictionary set is carried out as follows. The maximum- length prefix substring matched in the dictionary set is written to the next entry pointed by UP the of next dictionary along with the next character in the shift register. The update operation is inhibited if the next dictionary number is greater than or equal to the maximum dictionary number.

3.4 AHDB Processor

The AHDB processor encodes the output codewords from the PDLZW processor. As described previously, its purpose is to recode the fixed-length codewords into variable-length ones for taking the advantage of statistical property of the codewords from the PDLZW processor and, thus, to remove the information redundancy contained in the codewords. The encoding process is carried out as follows. The *pdlzw_output,* which is the output from the PDLZW processor and is the "symbol" for the AHDB algorithm, is input into *swap* unit for searching and deciding the matched index, , from the ordered list. Then the *swap* unit exchanges the item located in *n* with the item pointed by the pointer of the swapped block. That is, the more frequently used symbol bubbles up to the top of the ordered list. The index *ahdb_addr* of the "symbol" pdlzw_*output* of the ordered list is then encoded into a variable-length codeword (i.e., canonical Huffman codeword) and output as the compressed data for the entire processor. The operation of canonical Huffman encoder is as follows. The *ahdb_addr* is compared with all *codeword_offset :* 1, 9, 18, 31, 101, 154, 171, and 186 simultaneously, as shown in Table IV and Fig. 6, for deciding the length of the codeword to be encoded. Once the length is determined, the output codeword can be encoded as *ahdb_addr- code_offset + first_codeword.* For example, if *ahdb_addr*=38 from Table IV, the length is 8 b since 38 is greater than 31 and smaller than 101. The output codeword is: 38-31+44=001100112 As described above, the compression rate is between 1–5 B per memory cycle.

Test File	Text Files				
	Compress	AH	PDLZW +		
			AHAT	AHFB	AHDB
1.alice28.txt(152,089)	59.52	42.33	35.70	39.59	42.64
2.Asyoulik.txt(125,179)	56.07	39.38	32.53	37.12	40.07
3.book1(768,771)	56.81	43.00	39.93	37.78	40.98
4.book2(610,856)	58.95	40.02	40.18	38.88	41.85
5.cp.htm(24,603)	54.00	33.70	30.67	36.14	39.42
6.fields.c(11,150)	55.48	35.96	31.94	43.71	46.27
7.paper1(53,161)	52.83	37.29	30.65	37.47	40.40
8.paper2(82,199)	56.01	42.03	32.26	38.64	41.83
9.paper3(46,526)	52.36	41.14	30.04	37.45	40.63
10.paper4(13,286)	47.64	40.07	26.73	37.60	40.77
11.paper5(11,954)	44.96	36.86	25.93	37.13	40.25
12.paper6(38,105)	50.94	36.97	30.75	38.73	41.56
Avg	53.80	39.06	32.27	38.35	41.38

Table 1 Performance Comparison in Percentage of Data Reduction for Text file between Compress, PDLZW + AH, PDLZW + AHAT, PDLZW + AHFB, AND PDLZW + AHDB

Test File	Executable Files				
	Compress	AH	PDLZW +		
			AHAT	AHFB	AHDB
1.acrord32.exe(2,318,848)	34.10	21.77	29.93	30.15	31.41
2.cutftp32.exe(813,568)	40.72	27.74	34.74	37.47	38.41
3. fdisk.exe(64,124)	23.75	11.10	19.23	25.01	26.05
4.tc.exe(292,248)	25.45	14.72	23.09	25.79	26.90
5.waterfal.exe(186,880)	35.28	28.58	32.85	37.96	38.77
6.winamp.exe(892,928)	52.25	28.59	48.14	51.80	52.39
7.winzip32.exe(983,040)	41.35	21.26	36.47	39.95	40.52
8.xmplayer.exe(398,848)	36.28	26.70	31.00	34.87	35.65
Avg	36.14	24.51	31.93	35.37	36.26

Table 2 Performance Comparison in Percentage of Data Reduction for Executable file between Compress, PDLZW + AH, PDLZW + AHAT, PDLZW + AHFB, AND PDLZW + AHDB

4. Performance

Table 1 and 2 shows the compression ratio of the LZW (compress), the AH algorithm, PDLZW+AHAT, PDLZW+AHFB , and PDLZW+AHDB. The dictionary set used in PDLZW is only 368 addresses (words) and partitioned as {256,64,32,8,8}.From the table, the compression ratio of PDLZW + AHDB is competitive to that of the LZW (i.e., compress) algorithm in the case of executable files but is superior to that of the AH algorithm in both cases of text and executable files.

Because the cost of memory is a major part of any dictionary- based data compression processor discussed in the paper, we will use this as the basis for comparing the hardware cost of different architectures. According to the usual implementation of the AH algorithm, the memory requirement of an N- symbol alphabet set is (N + 1) + 4 (2N -1) integer variables [18], which is equivalent to 2 × {(N +1) + 4(2N-1)} = 4.5kB where N=256. The memory required in the AHDB algorithm is only a 256-B CAM, which corresponds to the 384-B static random-access memory (SRAM). Here, we assume the complexity of one CAM cell is 1.5 times that of a SRAM cell [21]. However, as seen from Tables I and II, the average performance of the AHDB algorithm is only 1.65%= ((39.50-36.86) + (26.89-26.23)/2)% worse than that of the AH algorithm.

After cascading with the PDLZW algorithm, the total memory cost is increased to 710-B CAM equivalently, which corresponds to 1065 B of RAM and is only one-fourth of that of the AH algorithm. However, the performance is improved by 8.11%=(39.66%-31.55%) where numbers 39.66% and 31.55% are from Tables VIII and III, respectively.

5. Results

The proposed two-stage compression/decompression processor given in Fig 5.3 has been synthesized and simulated using Verilog HDL. The resulting chip has a die area of 4.3× 4.3mm and a core area of 3.3 ×3.3 mm . The simulated power dissipation is between 632 and 700 mW at the operating frequency of 100 MHz. The compression rate is between 16.7 and 125 MB/s; the decompression rate is between 25 and 83 MB/s. Since we use D-type flip-flops associated with Two Stage Architecture needed gates as the basic memory cells of CAMs (the dictionary set in the PDLZW processor) and of ordered list (in the AHDB processor), these two parts occupy most of the chip area. The remainder only consumes about 20% of the chip area. To reduce the chip area and increase performance, the full-custom approach can be used. A flip-flop may take between 10 to 20 times the area of a six-transistor static RAM cell , a basic CAM cell may take up to 1.5 times the area (nine transistors) of a static RAM cell. Thus, the area of the chip will be reduced dramatically when full-custom technology is used. However, our HDL-based approach can be easily adapted to any technology, such as FPGA, CPLD, or cell library

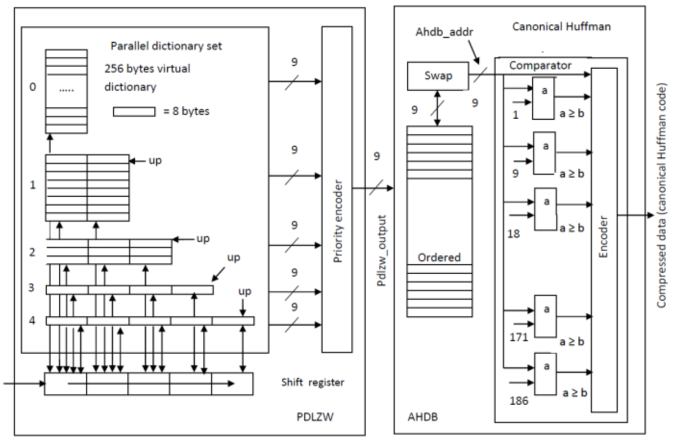

Fig- 2 Two-stage Architecture for compression

6. CONCLUSION

A new two-stage architecture for lossless data compression applications, which uses only a small-size dictionary, is proposed. This VLSI data compression architecture combines the PDLZW compression algorithm and the AH algorithm with dynamic-block exchange. The PDLZW processor is based on a hierarchical parallel dictionary set that has successively increasing word widths from 1 to 5 B with the capability of parallel search. The total memory used is only a 296-B CAM. The second processor is built around an ordered list constructed with a CAM of 414B (= 368 × 9B) and a canonical Huffman encoder. The resulting architecture shows that it is not only to reduce the hardware cost significantly but also easy to be realized in VLSI technology since the entire architecture is around the parallel dictionary set and an ordered list such that the control logic is essentially trivial. In addition, in the case of executable files, the performance of the proposed architecture is competitive with that of the LZW algorithm (compress). The data rate for the compression processor is at least 1 and up to 5 B per memory cycle. The memory cycle is mainly determined by the cycle time of CAMs but it is quite small since the maximum capacity of CAMs is only 64 × 2 B for the PDLZW processor and 414 B for the AHDB processor. Therefore, a very high data rate can be achieved.

REFERENCES

[1] Ming-Bo Lin, Jang-Feng Lee and Gene Eu Jan "LZW data compression and decompression algorithm and its hardware architecture," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.14, no.9, pp.925-936, Sep. 2006..

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 2nd ed. New York: McGraw-Hill, 2001.

[3] R. C. Gonzalez and R. E.Woods, Digital Image Processing. Reading, MA: Addison-Welsley, 1992.

[4] S. Henriques and N. Ranganathan, "A parallel architecture for data compression," in Proc. 2nd IEEE Symp. Parall. Distrib. Process., 1990, pp. 260–266.

[5] S.-A. Hwang and C.-W. Wu, "Unified VLSI systolic array design for LZ data compression," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 9, no. 4, pp. 489–499, Aug. 2001.

[6] S.-A. Hwang and C.-W. Wu, "Unified VLSI systolic array design for LZ data compression," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 9, no. 4, pp. 489–499, Aug. 2001.

[7] B. Jung and W. P. Burleson, "Efficient VLSI for Lempel-Ziv compression in wireless data communication networks," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 6, no. 3, pp. 475–483, Sep. 1998.

[8] D. E. Knuth, "Dynamic Huffman coding," J. Algorithms, vol. 6, pp. 163–180, 1985.

[9] M.-B. Lin, "A parallel VLSI architecture for the LZW data compression algorithm," in Proc. Int. Symp. VLSI Technol., Syst., Appl., 1997, pp. 98–101.

[10] J. L. Núñez and S. Jones, “Gbit/s lossless data compression hardware,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 11, no. 3, pp. 499–510, Jun. 2003.

[11] H. Park and V. K. Prasanna, “Area efficient VLSI architectures for Huffman coding,” IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process., vol. 40, no. 9, pp. 568–575, Sep. 1993.

[12] N. Ranganathan and S. Henriques, “High-speed vlsi designs for lempel-ziv-based data compression,” IEEE Trans. Circuits Syst. II. Analog Digit. Signal Process., vol. 40, no. 2, pp. 96–106, Feb. 1993.

[13] S. Khalid, Introduction to Data Compression, 2nd ed. San Mateo, CA: Morgan Kaufmann, 2000.

[14] B. W. Y.Wei, J. L. Chang, and V. D. Leongk, “Single-chip lossless data compressor,” in Proc. Int. Symp. VLSI Technol., Syst., Appl., 1995, pp. 211–213.

[15] T. A. Welch, “A technique for high-performance data compression,” IEEE Comput., vol. 17, no. 6, pp. 8–19, Jun. 1984.

[16] I. H. Witten, Alistair, and T. C. Bell, Managing Compressing and Indexing Documents and Images, 2nd ed. New York: Academic, 1999, pp. 36–51.

[17] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” IEEE Trans. Inf. Theory, vol. IT-23, no. 3, pp. 337–343, Mar. 1977.

[18] I. H. Witten, Alistair, and T. C. Bell, Managing Compressing and Indexing Documents and Images, 2nd ed. New York: Academic, 1999, pp. 36–51.