International Journal of Engineering Research-Online
A Peer Reviewed International Journal
Articles available online http://www.ijoer.in

Vol.3., Issue.6., 2015
(Nov.-Dec.,)

RESEARCH ARTICLE

ISSN: 2321-7758

ENHANCED SECOND GENERATION 2^n PATTERN RUN LENGTH ENCODING (RLE) SCHEME FOR TEST DATA COMPRESSION

M .D. SATYA PRIYA[1], G.S.S.PRASAD[2], NGN PRASAD[3]
[1]PG Scholar, [2,3]Assist Professor,
Kakinada institute of Engineering and Technology, A.P.

ABSTRACT
Now a day's communication plays a vital role and it is the main aspect in the present world. Due to this rapid changes are occurred and to transmit the data effectively and efficiently it takes large time and power consumption to transmit it. However recent technological breakthrough in high speed processing units and communication devices has enabled the development of high data compression schemes. This paper presents a modified scheme for second generation Run length encoding (RLE). Second Generation Run length encoding algorithm performs compression of input data based on sequences of identical values. But 2nd Generation RLE is having some limitations and they have been highlighted and discussed in detail in this paper. In 2nd Generation RLE largest number of sequences may increase the number of bits to represents the length of each run, which may increase the size of memory stack which may results in performance degradation. For 2n-bit run it requires 22n memory stack. If run length is greater than 2n bits we require 22n+1 memory stack to store the run value. An efficient coding technique, Bit stuffing has been suggested in this paper. A new bit different from the original sequence is added in between reduces the repeat length, thereby with the same stack we can represent length as well. This technique is described using VHDL and is implemented on Saprtan3 FPGA.

Keywords: Bit stuffing, compression, memory stack, run, and Run length encoding (RLE).

1. INTRODUCTION

Data compression implies sending or storing a smaller number of bits. Data compression is a process that reduces the amount of data in order to reduce data transmitted and decreases transfer time because the size of the data is reduced [1]. Data compression is commonly used in modern database systems. Compression can be utilized for different reasons including:1) Reducing storage/archival costs, which is particularly important for large data warehouses.2) Improving query workload performance by reducing the I/O costs[2]. In database systems, the disk I/O is one of the important factor. Any Data Compression technique in database is for reducing memory space. Compression is best approach to improve the system performance [3]. Compression has been around nearly as long as there has been research in database and has been much worked in this field. [4][5][6].Data compression involves transforming a

string of characters in some representation (such as ASCII) into a new string which contains the same information but with smallest possible length [3]. Data compression has important application in the areas of data transmission and data storage. Compressing data reduces storage and communication costs. Similarly, compressing a file to half of its original size is equivalent to doubling the capacity of the storage medium. Data compression is rapidly becoming a standard component of communications hardware and data storage devices. Data compression implies sending or storing a smaller number of bits. Although many methods are used for this purpose, in general these methods can be divided into two broad categories: lossless and lossy methods.

In lossless data compression, the integrity of the data is preserved. The original data and the data after compression and decompression are exactly the same because, in these methods, the compression and decompression algorithms are exact inverses of each other: no part of the data is lost in the process. Redundant data is removed in compression and added during decompression. Lossless compression methods are normally used when we cannot afford to lose any data. Our eyes and ears cannot distinguish subtle changes. In such cases, we can use a lossy data compression method. These methods are cheaper they take less time and space when it comes to sending millions of bits per second for images and video. Several methods have been developed using lossy compression techniques. JPEG (Joint Photographic Experts Group) encoding is used to compress pictures and graphics, MPEG (Moving Picture Experts Group) encoding is used to compress video, and MP3 (MPEG audio layer 3) for audio compression.

2. Run Length Encoding

Run-length encoding (RLE) is probably very simplest form of data compression in which runs of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run[3][4]. This is most useful on data that contains many such runs: for example, simple graphic images such as icons, line drawings, and animations. It is not useful with files that don't have many runs as it could greatly increase the file size. It can be used to compress data made of any combination of symbols [6]. It does not need to know the frequency of occurrence of symbols and can be very efficient if data is represented as 0s and 1s.The general idea behind this method is to replace consecutive repeating occurrences of a symbol by one occurrence of the symbol followed by the number of occurrences.

The method can be even more efficient if the data uses only two symbols (for example 0 and 1) in its bit pattern and one symbol is more frequent than the other RLE may also be used to refer to an early graphics file format supported by CompuServe for compressing black and white images, but was widely supplanted by their later Graphics Interchange Format. RLE also refers to a little-used image format in Windows 3.x, with the extension rle, which is a Run Length Encoded Bitmap, used to compress the Windows 3.x startup screen. Typical applications of this encoding are when the source information comprises long substrings of the same character or binary digit [7].

The RLE algorithm performs a lossless compression of input data based on sequences of identical values (runs). It is a historical technique, originally exploited by fax machine and later adopted in image processing. The algorithm is quite easy: each run, instead of being represented explicitly, is translated by the encoding algorithm in a pair (l,v) where l is the length of the run and v is the value of the run elements[3]. The longer the run in the Sequence to be compressed, the better is the compression ratio [3][7]. Run-length encoding (RLE) packs consecutive same values into a (value, length) pair to compress the data. For example, sequence ‘52, 52, 52, 4, 4’ will be encoded into ‘(52, 3), (4, 2)’. When there exists many runs of the same values, RLE can lead to a very high compression ratio. At the same time, RLE is very light-weighted in terms of both the compression and decompression performance [4][8-12].

3. Second generation 2n pattern Run Length Encoding Scheme: Our compression method, 2n -PRL, relies on encoding compatible patterns within the test data. Two patterns of the same length are compatible (inversely compatible) if every bit pair at the same position has the same (opposite) value after properly filling the values of don't-cares. 2n -PRL first partitions test data into fixed-length (L-bit) segments where L has a power of 2. Compression is then conducted on 2|n| compatible patterns where n is a signed integer. Depending on whether encoding is performed within a segment or across several adjacent segments, test data can be encoded in either of the following two ways: internal 2n -PRL (n<0) and external 2n PRL (n≥0), characterized by the sign of the exponent n. Internal 2n -PRL compresses 2|n| runs of compatible (inversely compatible) sub segments inside a single segment. For example, consider a 12-bit segment "100011011011." If it is divided into four sub segments, each of which has a length of 12 × 2−2 = 3 bits, the last three sub segments are found inversely compatible with the first sub segment "100." Hence ,the segment can be encoded by −2−2-PRL, where the negative sign before the radix represents that the sub segments following the first one are inversely compatible with the first one and the exponent value −2 represents that this segment is divided into 1/2−2 = 4 sub segments. A codeword used for internal 2n -PRL has three components as shown in Fig. 1(a), where sign (S) is the sign before the radix ("1" for negative and "0" for positive), exponent (E) is the K-bit exponent field value (including the sign of the exponent), and pattern (P) is the first sub segment data whose don't-cares are properly filled to enable compatibility with the subsequent sub segments. A negative sign bit before the radix represents inversely compatible. As an example, the above 12-bit segment can be encoded into codeword 1110 100 as shown in Fig. 1(b). External 2n -PRL compresses 2n consecutive compatible segments into a short one [13-16].

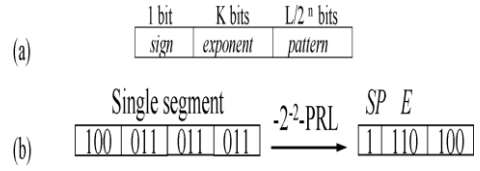

Fig. 1. (a) Codeword format for internal 2^n -PRL. (b) Encoding example.

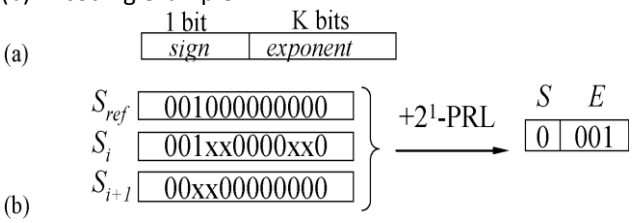

Fig. 2. (a) Codeword format for external 2^n-PRL. (b) Encoding example.

Table 1. Simple Encoding Table For L = 8.

Segments		S_{ref}	Codewords	Types
S_1	11X11XX1	11111111	01011	$+2^{-3}$-PRL
S_2	11XXXXX1	11111111	0001	$+2^{1}$-PRL
S_3	X1XXXXX1			
S_4	0XXXXX0X	00000000	1001	-2^{1}-PRL
S_5	X0XXXXXX			
S_6	X01XXXXX	10101010	011010	$+2^{-2}$-PRL
S_7	X10XXXX1	01010101	1010	-2^{2}-PRL
S_8	0XXXXXXX			
S_9	XXXXXXX			
S_{10}	X1XXXXX1			
S_{11}	X0X010XX	10101010	1001	-2^{1}-PRL
S_{12}	1XXX1XXX			
S_{13}	100XXXX1	10010101	111010	-2^{-2}-PRL
S_{14}	1011XX10	10111110	010010111110	$+2^{4}$-PRL

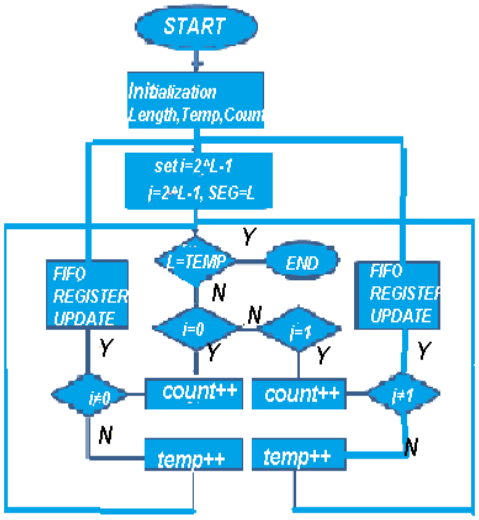

Fig.3. Flowchart of modified RLE compressor

Compression algorithm can be explained as follows:

1) First initialize the length, temp and count values.
2) Length denote length of data sequence, temp denote temporary register which holds the present data value of the length, count indicates the count of 0's and 1's indicated by i and j, but initially count =0.
3) Initialize i=2^L-1, AND j= 2^L-1, where i is used to

count no of 0's and j for no of 1's.

4) If L=TEMP, then compression stops, since data is not available.
5) If step 4 is false , then check whether i =0 or not.
6) In this step if i=0 then count is incremented i.e.,count = count +1, temporary register is incremted for next data value, FIFO register flag is also updated to hold the count for i.
7) If i=0 is false, then it cheks for j=1,if it is true then count is incremented i.e.,count = count +1, temporary register is incremted for next data value, FIFO register flag is also updated to hold the count for j.
8) The steps 6 and 7 repeat until length= tem, if equal then compression stops and FIFO register holds the run length count of i and j values.
9) Print last data.

4. Decompression architecture for 2^n PRL

Fig. 4 shows the general architecture of our decompression method. Conceptually, the decompression architecture of 2^n PRL comprises a finite-state machine (FSM) and a control and generator unit (CGU). The FSM is responsible for codeword identification. The CGU is responsible for controlling data transmission between the ATE and the FSM, generating test patterns, and controlling the scan chain of the CUT. Fig. 4 presents the decompressor for 2^n -PRL with 8-bit segments (L = 8) and a 2-bit exponent (K = 2). The decompressor is delineated as follows.

1) ***Decoder:*** It identifies code words and generates control signals such as *Inverse*, *Load*, and *Shift*. *Inverse* decides whether the buffer content should be inverted due to decompressing inversely compatible segments. *Load* enables loading a decoded test pattern from the multiplexer array into the buffer when a segment is encoded by internal 2^n -PRL or an exception type. *Shift* enables sending test data in the buffer into the scan chain.

2) ***Multiplexer Array:*** It employs an array of multiplexers to produce test patterns. The number of multiplexers equals

$L - L/2K-1$. There are four multiplexers for K = 2 and L = 8. These multiplexers are connected to the decoder and buffer in a manner shown in Fig.5.

3) ***Buffer:*** It stores the decompressed test data and sends them to the scan chain.

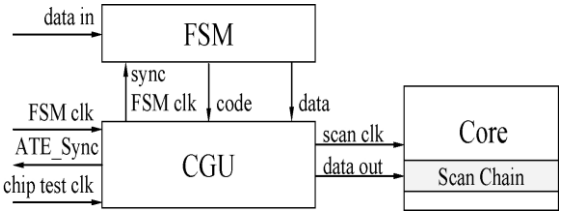

Fig.4. General decompression architecture of 2^n -PRL.

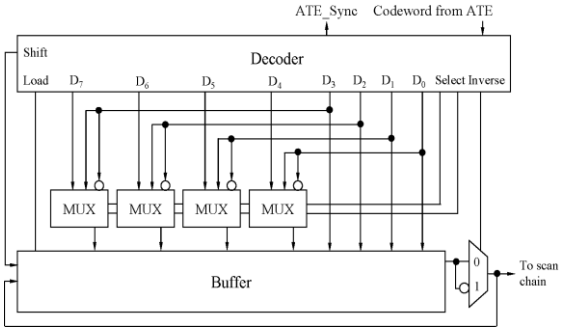

Fig.5. Decompressor for 2^n -PRL with L = 8 and K = 2

5. Simulation Results

HDL design has the ability to simulate HDL programs. Simulation allows an HDL description of a design (called a model) to pass design verification, an important milestone that validates the design's intended function (specification) against the code implementation in the HDL description. It also permits architectural exploration. The engineer can experiment with design choices by writing multiple variations of a base design, then comparing their behavior in simulation. Thus, simulation is critical for successful HDL design. An HDL simulator — the program that executes the testbench — maintains the simulator clock, which is the master reference for all events in the testbench simulation. Events occur only at the instants dictated by the testbench HDL (such as a reset-toggle coded into the testbench), or in reaction (by the model) to stimulus and triggering events. Modern HDL simulators have full-featured graphical user interfaces, complete with a suite of debug tools. These allow the user to stop and restart the simulation at any time, insert simulator breakpoints (independent of the HDL code), and monitor or modify any element in the HDL model hierarchy

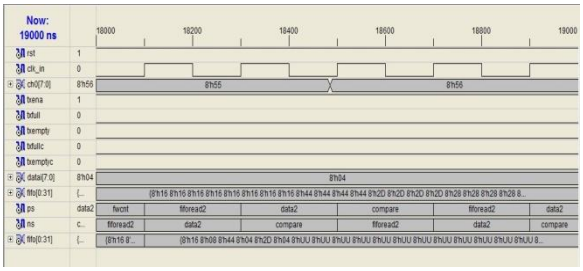

Fig.6. Compressed output

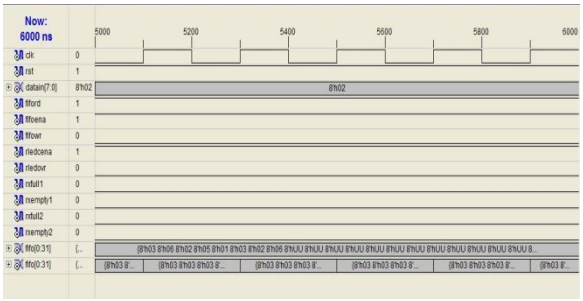

Fig.7. Decompressed output

Table 2.Compression Ratios Obtained by Our Method and Previous Works.

Circuits	SHC [7]	VIHC [20]	RL-HC [10]	9C [10]	BM [16]	MD-PRC [18]	2^n-PRL
s5378	**55.10**	51.78	53.75	51.64	54.98	54.63	54.94
s9234	54.20	47.25	47.59	50.91	51.19	53.20	**57.72**
s13207	77.00	83.51	82.51	82.31	84.89	86.01	**88.10**
s15850	66.00	67.94	67.34	66.38	69.49	69.99	**74.29**
s38417	59.00	53.36	**64.17**	60.63	59.39	55.38	58.33
s38584	64.10	62.28	62.40	65.53	66.86	67.73	**72.44**
Average	62.57	61.02	62.96	62.90	64.47	64.49	67.64

6. Conclusion

This paper has presented a run-length-based compression method called 2^n -PRL. 2^n -PRL is very effective in compressing $2|n|$ successively compatible (or inversely compatible) patterns either inside a segment or across multiple segments. The decompressor was small and easy to implement. The experimental results showed that 2^n -PRL can achieve an average compression ratio of up to 67.64%.

References

[1]. S. Mitra and K. S. Kim, "X-Compact: An efficient response compaction technique," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol.23, no. 3, pp. 421–432, Mar. 2004.

[2]. J. Rajski, J. Tyszer, M. Kassab, and N. Mukherjee, "Embedded deterministic test," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 23, no. 5, pp. 776–792, May 2004.

[3]. S. Mitra and K. S. Kim, "XMAX: X-tolerant architecture for maximal test compression," in *Proc. IEEE Int. Conf. Comput. Des.*, Oct. 2003,pp. 326–330.

[4]. B. Koenemann, C. Banhart, B. Keller, T. Snethen, O. Farnsworth, and D. Wheater, "A SmartBIST variant with guaranteed encoding," in *Proc.Asia Test Symp.*, 2001, pp. 325–330.

[5]. N. A. Touba, "Survey of test vector compression techniques," *IEEE Des. Test Comput.*, vol. 23, no. 4, pp. 294–303, Apr. 2006.

[6]. D. A. Huffman, "A method for the construction of minimum redundancy codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[7]. A. Jas, J. Ghosh-Dastidar, N. Mom-Eng, and N. A. Touba, "An efficient test vector compression scheme using selective Huffman coding," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 22, no. 6, pp. 797–806, Jun. 2003.

[8]. X. Kavousianos, E. Kalligeros, and D. Nikolos, "Optimal selective Huffman coding for test-data compression," *IEEE Trans. Comput.*, vol. 56, no. 8, pp. 1146–1152, Aug. 2007.

[9]. P. T. Gonciari, B. Al-Hashimi, and N. Nicolici, "Improving compression ratio, area overhead, and test application time for system-on-a-chip test data compression/decompression," in *Proc. Des. Automat. Test Eur.*, Mar. 2002, pp. 604–611.

[10]. M. Tehranipoor, M. Nourani, and K. Chakrabarty, "Nine-coded compression technique for testing embedded cores in SoCs," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 13, no. 6, pp. 719–731, June.

[11]. Efficient coding schemes for the hardsquare model, IEEE Trans. Inform. Theory, vol. 47, pp. 1166–1176, Mar. 2001.

[12]. B. Ramamurthi and A. Gersho. Classified Vector Quantization of Images. IEEE Transactions on Communications, COM-34:1105–1115, November 1986.

[13].] M. Hans and R.W. Schafer. AudioPak—An Integer Arithmetic Lossless Audio Code. In Proceedings of the Data Compression Conference, DCC '98. IEEE, 1998.

[14]. G. Langdon and J.J. Rissanen. Compression of black-white images with arithmetic coding. IEEE Transactions on Communications, 29(6):858–867, 1981.

[15]. J. Ziv and A. Lempel. A universal algorithm for data compression. IEEE Transactions on Information Theory, IT-23(3):337–343, May 1977.

[16]. M. Nelson and J.-L. Gailly. The Data Compression Book. M&T Books, CA, 1996.