International Journal of Advance Research and Innovation
Vol. 7(2), Apr-Jun 2019, pp. 77-81
Doi: 10.51976/ijari.721912
www.gla.ac.in/journals/ijari
© 2019 IJARI, GLA University

Article Info
Received: 01 May 2019 | Revised Submission: 20 May 2019 | Accepted: 28 May 2019 | Available Online: 15 Jun 2019

Improvement of Test Data Compression Using Huffman and Golomb Coding Techniques

Thilagavathi and Karthick***

ABSTRACT

Test vector compression is an emerging trend in the field of VLSI testing. According to these trends, increasing test data volume is one of the biggest challenges in the testing industry. The overall throughput of automatic test equipment (ATE) is sensitive to the download time of test data. An effective approach to the reduction of the download time is to compress test data before the download. But this tester has limited speed, memory and I/O channels. The test data bandwidth between the tester and the chip is small which is the bottleneck in determining how fast the testing process. To overcome these limitations of the Automatic Test Equipment (ATE), a new hybrid test vector compression technique is proposed. the large volume of test data input is compressed in a hybrid fashion before being downloaded into the processor and the test compression ratio is increased and is experimentally verified with the benchmark circuits.

Keywords: *Automatic Test Equipment; Data Compression; Golomb Coding.*

1.0 Introduction

Compression is used just about everywhere. All the images you get on the web are compressed, typically in the JPEG or GIF formats, most modems use compression, HDTV will be compressed using MPEG-2, and several file systems automatically compress files when stored, and the rest of us do it by hand. The neat thing about compression, as with the other topics we will cover in this course, is that the algorithms used in the real world make heavy use of a wide set of algorithmic tools, including sorting, hash tables, tries, and FFTs. Furthermore, algorithms with strong theoretical foundations play a critical role in real-world applications.Lossless compression algorithms usually exploit statistical redundancy in such a way as to represent the sender's data more concisely, but nevertheless perfectly. Lossless compression is possible because most real-world data has statistical redundancy. For example, in English text, the letter 'e' is much more common than the letter 'z', and the probability that the letter 'q' will be followed by the letter 'z' is very small.Another kind of compression, called lossy data compression, is possible if some loss of fidelity is acceptable. For example, a person viewing a picture or television video scene might not notice if some of its finest details are removed or not represented perfectly (i.e. may not even notice compression artifacts). Similarly, two clips of audio may be perceived as the same to a listener even though one is missing details found in the other. Lossy data compression algorithms introduce relatively minor differences and represent the picture, video, or audio using fewer bits.

2.0 Literature Review

As mentioned in the introduction, coding is the job of taking probabilities for messages and generating bit strings based on these probabilities. In practice we typically use probabilities for parts of a larger message rather than for the complete message, e.g., each character or word in a text. To be consistent with the terminology in the previous section, we will consider each of these components a message on its own, and we will use the term message sequence for the larger message made up of these components. In general each little message can be of a different type and come from its own probability distribution. For example, when sending an image we might send a message specifying a color followed by messages specifying a frequency component of that color. Even the messages specifying the color might come from different probability distributions since the probability of particular colors might depend on the context. We distinguish between algorithms that

**Corresponding Author: Department of Electronics and Communication Engineering, Gnanamani College of Technology. Tamil Nadu, India (E-mail: thilagaece2013@gmail.com)*
***Department of Electronics and Communication Engineering, Gnanamani College of Technology. Tamil Nadu, India*

assign a unique code (bit-string) for each message, and ones that "blend" the codes together from more than one message in a row. In the first class we will consider Huffman codes, which are a type of prefix code.

In the later category we consider arithmetic codes. The arithmetic codes can achieve better compression, but can require the encoder to delay sending messages since the messages need to be combined before they can be sent.

3.0 Dictionary-Based Code Compression

Dictionary based compression techniques are extremely popular in embedded systems domain since they provide a dual advantage of good compression ratio as well as a fast decompression mechanism.

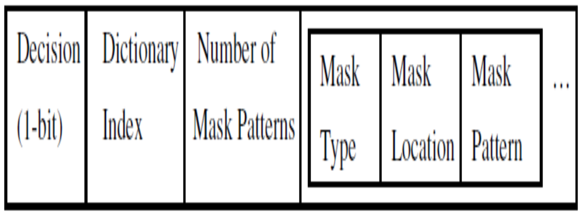
The basic idea is to take advantage of commonly occurring instruction sequences by using a dictionary.

In general a dictionary contains 256 or more entries. As a result, a code pattern will have fewer than 32 bit changes.

If a code pattern is different from a dictionary entry in 8 bit positions, it requires only one 8-bit mask and its position i.e., it requires 13 (8+5) extra bits.

This can be improved further if we consider bit changes only in byte boundaries.

Fig 1: Generic Encoding Format



This leads to a tradeoff - requires fewer bits (8+2) but may miss few mismatches that spread across two bytes. Our study uses the latter approach that uses fewer bits to store a mask position.

If we choose two distinct bit-mask patterns, 2-bit fixed (2f) and 4-bit sliding (4s), we can generate six combinations: (2f), (4f), (2f, 2f), (2f, 4f), (4f, 2f), (4f, 4f). Similarly, three distinct mask patterns can create up to 39 combinations.

Now we can try to answer the two questions posed at the beginning of this section.

Table 1: Various Bit-Mask Patterns

Bit-Mask	Fixed	Sliding
1 bit		X
2 bits	X	X
3 bits		X
4 bits	X	X
5 bits		X
6 bit		X
7 bit		X
8 bit	X	X

It is easy to answer the first question: up to two mask patterns are profitable. The reason is obvious based on the cost consideration. The smallest cost to store the three bitmask information (position and pattern) is 15 bits (if three 1-bit sliding patterns are used). In addition, we need 1-5 bits to indicate the mask combination and 8-14 bits for a codeword (dictionary index). Therefore, we require approximately 29 bits (on average) to encode a 32-bit vector. In other words, we save only 3 bits to match 3 bit differences (on a 32-bit vector). Clearly, it is not very profitable to use three or more bitmask patterns. Applying a larger bitmask can generate more matching patterns. However, it may not improve the compression ratio. Similarly, using a sliding mask where a fixed one is sufficient is wasteful since a fixed mask require fewer number of bits (compared to its sliding counterpart) to store the position information. For example, if a 4-bit sliding mask (cost of 9 bits) is used where a 4-bit fixed (cost of 7 bits) is sufficient, two additional bits are wasted.

4.0 Proposed System

Data compression is known for reducing storage and communication costs. It involves transforming data of a given format, called source message, to data of a smaller sized format, called code word. Data encryption is known for protecting information from eavesdropping. It transforms data of a given format, called plaintext, to another format, called cipher text, using an encryption key.

Our decompression hardware for variable-length coding is capable of operating at the speed closest to the best known field- programmable gate array-based decoder for fixed-length coding. On Compressed Bit streams, more configuration information can be stored using the same memory.

The access delay is also reduced, because less bits need to be transferred through the memory interface. To measure the efficiency of bit stream, is meant by Compression Ratio (CR). It is defined as the ratio between the Compressed Bit stream Size (CS) and the Original Bit stream Size (OS).

4.1 Golomb coding

Golomb Coding is a lossless data compression method using a family of data compression codes invented by Solomon W. Golomb in the 1960s. Golomb coding for data compression is a well-known technique due to its lower complexity. Thus, it has become one of the favorite choices for lossless data compression technique in many applications especially in mobile multimedia communication. In this paper, the development of Golomb Coding compression and decompression algorithms in the Field Programmable Gate Array (FPGA) is presented. The coding scheme development in FPGA utilises the Verilog HDL. Alphabets following a geometric distribution will have a Golomb code as an optimal prefix code, making Golomb coding highly suitable for situations in which the occurrence of small values in the input stream is significantly more likely than large values.

Rice coding (invented by Robert F. Rice) denotes using a subset of the family of Golomb codes t/o produce a simpler (but possibly suboptimal) prefix code; Rice used this in an adaptive coding scheme, although "Rice coding" can refer to either that scheme or merely using that subset of Golomb codes. Whereas a Golomb code has a tunable parameter that can be any positive value, Rice codes are those in which the tunable parameter is a power of two. This makes Rice codes convenient for use on a computer, since multiplication and division by 2 can be implemented more efficiently in binary arithmetic. Rice coding is used as the entropy encoding stage in a number of lossless image compression and audio data compression methods.

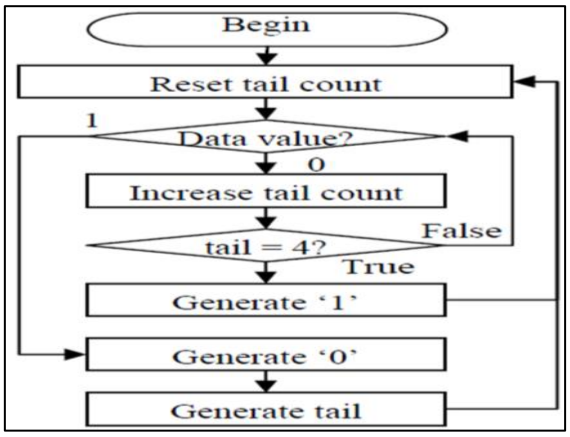
Note below that this is the Rice-Golomb encoding, where the remainder code uses simple truncated binary encoding, also named "Rice coding" (other varying-length binary encodings, like arithmetic or Huffman encodings, are possible for the remainder codes, if the statistic distribution of remainder codes is not flat, and notably when not all possible remainders after the division are used). In this algorithm, if the M parameter is a power of 2, it becomes equivalent to the simpler Rice encoding.

4.2 Huffman coding

Huffman coding is a variable-length encoding scheme. The number of bits required to store a coded character varies according to the relative frequency or weight of the character. A significant space savings is achieved for frequently used characters (requiring only one, two or three bits). Little space saving is achieved for infrequent characters. A Huffman Coding Tree is built from the observed frequencies of characters in a document. The document is scanned and the occurrence of each character is recorded. Next, a Binary Tree is built in which the external nodes store the character and the corresponding character frequency observed in the document.

The Huffman encoding algorithm starts by constructing a list of all the alphabet symbols in descending order of their probabilities. It then constructs, from the bottom up, binary tree with a symbol at every leaf. This is done in steps, where at each step two symbols with the smallest probabilities are selected, added to the top of the partial tree, deleted from the list, and replaced with an auxiliary symbol representing the two original symbols. When the list is reduced to just one auxiliary symbol (representing the entire alphabet), the tree is complete. The tree is then traversed to determine the codewords of the symbols.

Fig 2: Golomb Encoding Flowchart



A symbol is any 8-bit combination as well as an End Of File (EOF) marker. This means that there are 257 possible symbols in any code. As an entropy encoding scheme, Huffman encoding assigns short code words to frequently occurring words and longer code words to infrequently occurring words. Huffman encoding is based on a Huffman tree, which in turn is created upon a probability distribution of words.

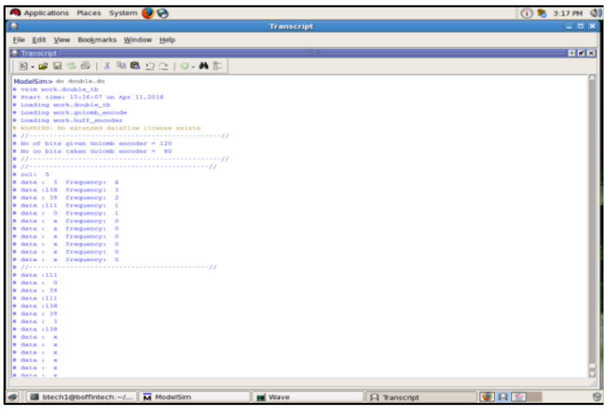
Creating an individual Huffman tree for each bitstream file is called dynamic Huffman encoding. Such a Huffman tree is part of the compressed data and has to be reconstructed before decompression. Such decompressors turn out to be costly in terms of hardware usage.

Applying the same Huffman tree for all bitstreams is called static Huffman encoding. The static Huffman tree is based on the probability distribution of all words in our benchmark corpus and will never change. The compression rate for the overall corpus is hence still optimal, whereas we could achieve better compression rates for single bitstream files using dynamic Huffman encoding.

5.0 Results

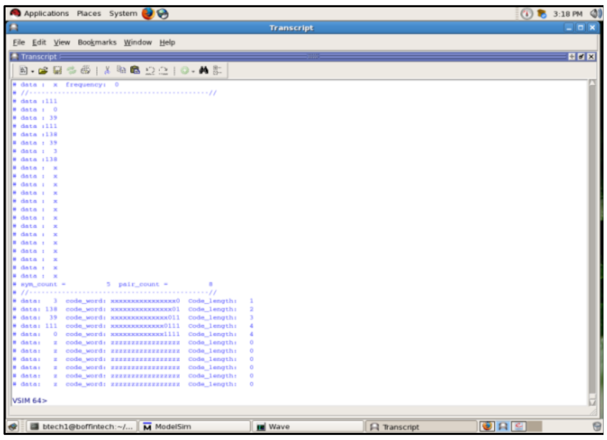
Simulation Results are shown by figure 3 and figure 4. Synthesis results shown in figure 5 and figure 6.

Fig 3: Golomb Code Output Simulation Waveform



```
ModelSim> do double.do
# vsim work.double_tb
# Start time: 15:16:07 on Apr 11,2018
# Loading work.double_tb
# Loading work.golomb_encode
# Loading work.huff_encoder
# WARNING: No extended dataflow license exists
# //-----------------------------------------------//
# No of bits given Golomb encoder = 120
# No oo bits taken Golomb encoder =  80
# //-----------------------------------------------//
# //--------------------------------------------//
# col:  5
# data :  3  frequency:  4
# data :138  frequency:  3
# data : 39  frequency:  2
# data :111  frequency:  1
# data :  0  frequency:  1
# data :  x  frequency:  0
# data :  x  frequency:  0
# data :  x  frequency:  0
# data :  x  frequency:  0
# data :  x  frequency:  0
# data :  x  frequency:  0
# //--------------------------------------------//
# data :111
# data :  0
# data : 39
# data :111
# data :138
# data : 39
# data :  3
# data :138
# data :  x
# data :  x
# data :  x
# data :  x
# data :  x
# data :  x
```

Fig 4: Golomb Code to Huffman Code Output



```
# data :  x  frequency:  0
# //--------------------------------------------//
# data :111
# data :  0
# data : 39
# data :111
# data :138
# data : 39
# data :  3
# data :138
# data :  x
# data :  x
# data :  x
# data :  x
# data :  x
# data :  x
# data :  x
# data :  x
# data :  x
# data :  x
# data :  x
# data :  x
# data :  x
# data :  x
# sym_count =          5  pair_count =          8
# //--------------------------------------------//
# data:   3  code_word: xxxxxxxxxxxxxxx0  Code_length:   1
# data: 138  code_word: xxxxxxxxxxxxxx01  Code_length:   2
# data:  39  code_word: xxxxxxxxxxxxx011  Code_length:   3
# data: 111  code_word: xxxxxxxxxxxx0111  Code_length:   4
# data:   0  code_word: xxxxxxxxxxxx1111  Code_length:   4
# data:   z  code_word: zzzzzzzzzzzzzzzz  Code_length:   0
# data:   z  code_word: zzzzzzzzzzzzzzzz  Code_length:   0
# data:   z  code_word: zzzzzzzzzzzzzzzz  Code_length:   0
# data:   z  code_word: zzzzzzzzzzzzzzzz  Code_length:   0
# data:   z  code_word: zzzzzzzzzzzzzzzz  Code_length:   0
# data:   z  code_word: zzzzzzzzzzzzzzzz  Code_length:   0

VSIM 64>
```

Fig 5: Golomb Simulation Waveform

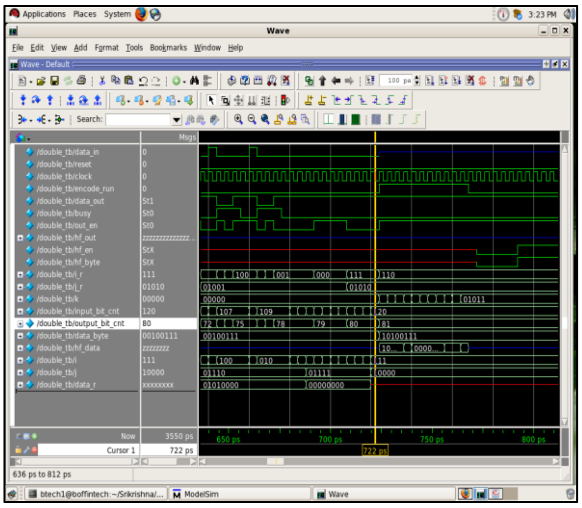
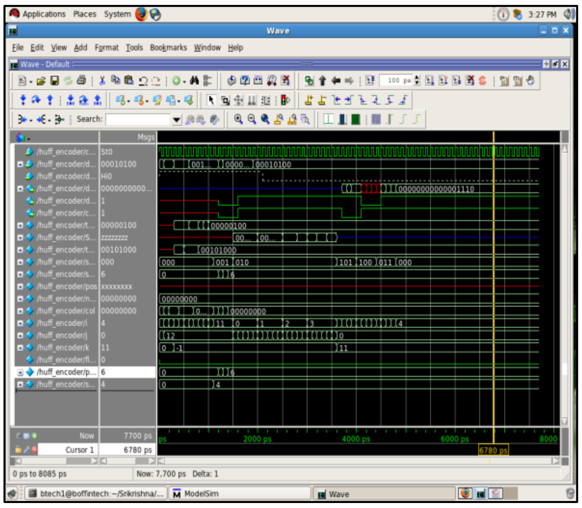


Fig 6: Huiffman Simulation Waveform



6.0 Conclusions

The test compression technique which combines both Huffman and Golomb coding is proposed. Thus, it reduces both the amount of test storage and testing time, thereby reducing the tester memory and channel capacity requirements. As the proposed method is mainly software based, the hardware requirements and cost of ATE are minimized. The technique is completely lossless and time and space efficient because of its higher compression ratio and rapid decompression process. Currently, work is underway on implementing the decompression procedure in the embedded processor along with automatic application of test vectors for analyzing test fault coverage. The test vector compression is implemented through ModelSim 6.4a version and the functionality of each test compression technique was verified. Among these three, compression produces more reduction in test vector than the normal individual coding schemes.

References

[1] R Merkle. Secure communication over an insecure channel, Communications of the ACM.

[2] D Kahn. The Codebreakers, The Story of Secret Writing. New York: Macmillan, 1967.

[3] CE Shannon. Communication theory of secrecy systems, Bell Syst. Tech. J., 28(8), 1949, 656–715.

[4] ME Hellman. An extension of the Shannon theory approach to cryptography. IEEE Trans..

[5] HF Gaines. Cryptanalysis, 1939, Dover. ISBN 0-486-20097-3 .

[6] Asinkov. Elementary Cryptanalysis: A Mathematical Approach, Mathematical Association of America, ISBN 0-88385-622-0,1966.

[7] S Kwong, YF Ho. A Statistical Lempel- Ziv Compression Algorithm for Personal Digital Assistant (PDA), IEEE Transactions on Consumer Electronics, 47(1), 2001.

[8] L Li, K Chakrabarty. Test Data Compression Using Dictionaries with Fixed-Length Indices, Proceedings of the 21st IEEE VLSI Test Symposium (VTS.03).

[9] PG Howard, JS Vitter. Practical Implementations of Arithmetic Coding, Providence, R.I. 02912-1910.

[10] S Yang. P Qiu. Efficient Integer Coding for Arbitrary Probability Distributions, IEEE Transactions On Information Theory, 52(8), 2006.

[11] P Elias. Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory, 21(2), 1975, 194-203.

[12] J Walder, M Kratky, J Platos. Fast Fibonacci Encoding Algorithm, Dateso, 72-83, ISBN 978-80-7378-116-3, 2010.

[13] ST Klein, MKB Nissan. On the Usefulness of Fibonacci Compression Codes, Computer Journal, 2005.

[14] X Kavousianos, E Kalligeros, D Nikolos. Multilevel-Huffman test-data compression for IP cores with multiple scan chains, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 16(7), 2008.

[15] Jas, NA Touba. Test Vector Compression via Cyclical Scan Chains and Its Application to Testing Core-Based Designs, Proc. Int’l Test Conf. (ITC 98), IEEE CS Press, 1998, 458-464.

[16] Chandra, K Chakrabarty. Test Data Compression for System-on-Chip Using Golomb Codes, VTS '00: Proceedings of the 18th IEEE VLSI Test Symposium, 2000.