A HIGH THROUGHPUT VLSI ARCHITURE DESIGN OF CANONICAL HUFFMAN ENCODER

Dr. S. Vijayalakshmi
Department of Electronics and Communication Engineering
Vel Tech Rangarajan Dr. Sagunthala R&D Institute of Science and Technology, Chennai, India.

E. Harinath Reddy Yadav
Department of Electronics and Communication Engineering
Vel Tech Rangarajan Dr. Sagunthala R&D Institute of Science and Technology, Chennai, India.

V. Padmasri Kavitha
Department of Electronics and Communication Engineering
Vel Tech Rangarajan Dr. Sagunthala R&D Institute of Science and Technology, Chennai, India.

U. Ramadevi
Department of Electronics and Communication Engineering
Vel Tech Rangarajan Dr. Sagunthala R&D Institute of Science and Technology, Chennai, India.

Abstract—

The Huffman encoding technique is a method of compressing data that takes input in the form of strings, which are typically 1 byte or 8 bits in length, and reduces them to a much smaller size, often just a few bits. This is accomplished by encoding the string data as binary bits using a Huffman encoder, which is typically implemented using a VLSI architecture. The encoding process involves two key steps: frequency sorting and Huffman coding. The frequency sorting block sorts the strings based on their frequency, typically in terms of ASCII values. This sorted data is then passed to the Huffman encoder, which assigns a variable-length code to each string based on its frequency. The resulting encoded data is typically much smaller than the original data, and can be decoded back to the original data with negligible loss of information. In this article, we propose a modified version of the Huffman encoding technique that encodes each byte into less than one byte. Specifically, we take 64 bits of data, consisting of eight string values, and sort them based on their ASCII values. The resulting data is then passed through a Huffman encoder to further reduce its size. In the worst case, the encoded data will still be 8 bits (i.e., 1 byte), but in most cases it will be much smaller.

I. INTRODUCTION

Huffman encoding is a widely used compression technique that minimizes the amount of input data by assigning shorter codes to frequently occurring symbols. In order to achieve high compression ratios, the input data must be sorted by frequency before being passed to the Huffman block. To address the limitations of current devices, a high-throughput Huffman encoder VLSI design based on the typical/canonical Huffman encoder was proposed in this study. In this architecture, each input symbol is treated as a separate node in a tree, and the two lowest frequency nodes are merged to form a new node. To optimize memory allocation for Huffman coding, a pre-defined code word table was suggested to reduce the computational workload with minimal performance degradation[10]. However, searching for Huffman codes in the table requires many clock cycles, which can limit the efficiency of Huffman coding. To improve the efficiency, a novel data structure was developed, although the properties of this data structure required challenging calculations, resulting in a low clock frequency. Although a PLA approach can achieve fast Huffman coding, it often requires significant hardware to store the code word table[12].

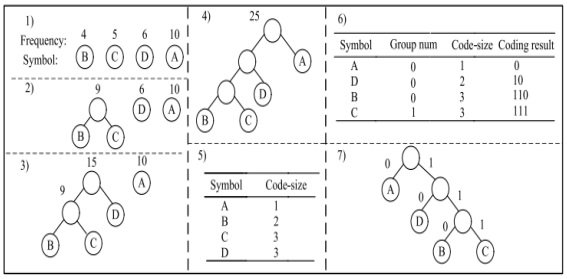

Fig 1:- process of canonical Huffman coding

The diagram presented depicts the process of Canonical Huffman encoding. This paper is divided into several sections. Firstly, in Section II, an overview of the Canonical Huffman encoder algorithm is presented. Then, Section III describes

the proposed serial architecture in detail. Section IV provides experimental results, which are compared to other architectures. Finally, Section V provides brief conclusions.

II. OVERVIEW OF CANONICAL HUFFMAN ENCODING

Huffman Coding is a lossless data compression technique that assigns a variable length prefix code to each character in the data. The code assigned to each character is based on its frequency in the data, with the least frequent character getting the largest code and the most frequent character getting the smallest code. While encoding the data is straightforward and efficient, decoding the bitstream generated by this method is not very efficient. To decode the data back to its original characters, the decoder needs to know the encoding mechanism used. Therefore, information about the encoding process is passed to the decoder as a table of characters and their corresponding codes. In regular Huffman coding of large data, the table takes up a lot of memory space, and if there are many unique characters in the data, the compressed data size increases due to the presence of the codebook. To make the decoding process computationally efficient while maintaining a good compression ratio, Canonical Huffman codes were introduced. In Canonical Huffman coding, the bit lengths of the standard Huffman codes generated for each symbol are used. The symbols are sorted according to their bit lengths in non-decreasing order and then sorted lexicographically for each bit length. The first symbol gets a code containing all zeros and of the same length as its original bit length. For subsequent symbols, if a symbol has the same bit length as the previous symbol, then the code of the previous symbol is incremented by one and assigned to the present symbol. Otherwise, if the symbol has a bit length greater than that of the previous symbol, the code of the previous symbol is incremented, zeros are appended until the length becomes equal to the bit length of the current symbol, and then the code is assigned to the current symbol. This process continues for the rest of the symbols.

III. PROPOSED SYSTEM

A. Overview

The proposed design consists of three stages: Frequency-Generation, Code-Size Computing & Sorting, and Code-Size-Limiting, as shown in Figure 2. To improve the encoding efficiency and overcome the limitations of the Canonical Huffman encoder, we propose two types of real-time frequency-sorting architectures in the first two stages that process the input symbol in a series scheme. Details of these architectures are provided in Sections III-B and III-C. The code-size-sorting module generates a temporary sorted result of the code-size data queue at each clock cycle, based on the hardware architecture. Once the code-size calculation process is complete, the HUFFMANVAL results are simultaneously generated.

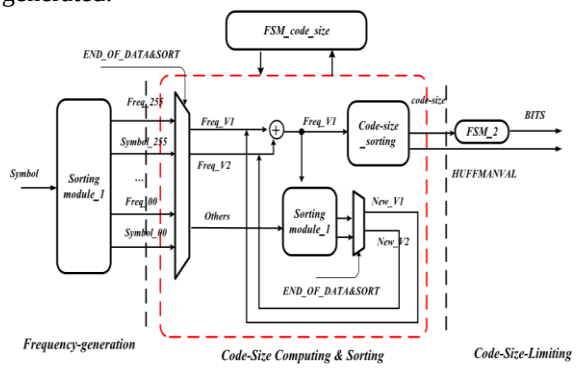

Fig 2:- System circuit block diagram.

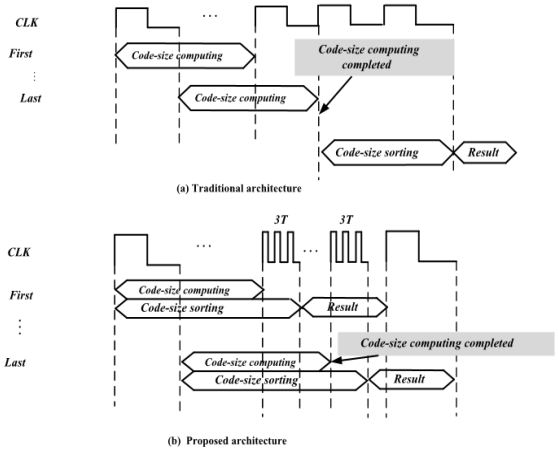

Fig 3:- Timing diagram comparison between the proposed architecture and the traditional designs.

This brief also presents an efficient VLSI architecture for the Code-Size-Limiting stage, which limits the bit length to enhance the encoding speed. The last stage is based on standard algorithms, and this brief optimizes the nesting of the algorithms to reduce the circuit area and power consumption effectively.

As depicted in Figure 3, the proposed architecture reduces the required clock cycle significantly compared to traditional Huffman encoder designs.

B. Architecture of Frequency-Generation Stage

The proposed stage consists of two parallel parts - the frequency-statistics process and the frequency-sorting process. It "eats" one symbol per cycle and generates 256 sets of ordered frequencies. This can be done almost simultaneously when the last symbol is entered due to the insertion of only two pipeline stages, leading to significant improvement in throughput and reduction in encoding time. This is in contrast to traditional designs where the sorting module is initiated only after completion of frequency statistics of all input symbols.

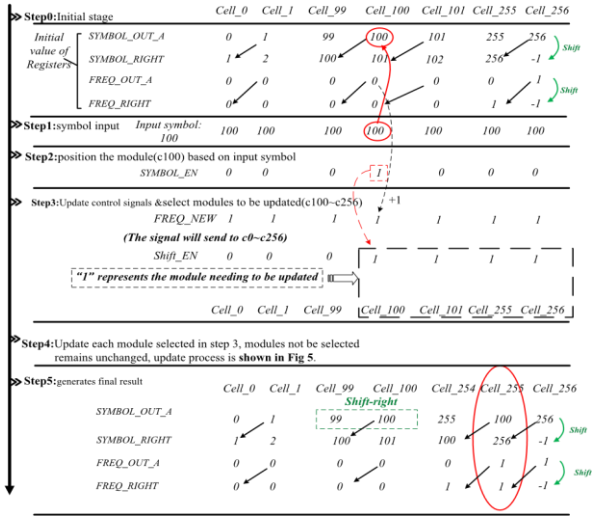

Fig 4:- working Mechanism of Frequency-Statistics & Sorting Stage

The operating mechanism of the Frequency-Statistics & Sorting stage is illustrated in Fig. 4. To optimize the performance of the sorting algorithm and improve efficiency by reducing the clock cycle, the sorting process is used to identify two nodes with the minimum and sub-minimum frequencies. To accelerate the encoding efficiency and realize the parallel characteristics, 257 cells (Cell_0~Cell_256) are employed to store and update each symbol and its frequency. Cell_256 holds the symbol with the largest frequency in the final result, Cell_255 holds the symbol with the second-largest frequency, and so on. Finally, Cell_0 stores the symbol with the smallest frequency, while Cell_256 provides an inverse code point.

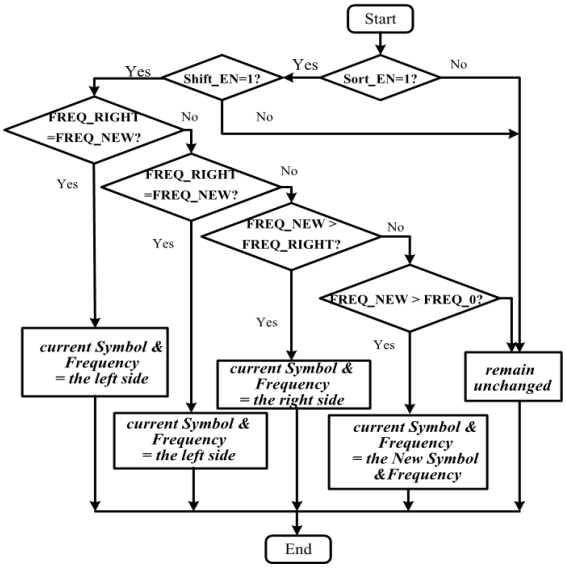

Fig 5:- Tree chat of Frequency-Statistics & Sorting Stage

The operating steps of the Frequency-Statistics & Sorting stage are shown in Fig. 4, which includes the following steps:

- Step-0: The initial stage.
- Step-1: The proposed design receives the input symbol. For example, assuming the current input symbol is "100," each cell's stored symbols are compared with "100."
- Step-2: The cell that stores the input symbol "100" will be positioned, and the corresponding signal SYMBOL_EN[100] will be set to "1." Additionally, the signal Shift_EN_100~Shift_EN_256 will be set to "1."
- Step-3: The Shift_EN signal that is set to 1'b1 indicates that the corresponding cell needs updating. Meanwhile, the new frequency FREQ_NEW, which is one more than the original frequency, is added. Additionally, FREQ_OUT_A in the cell that stores the symbol "100" is also increased by 1.
- Step-4: Each selected cell in Step-3 will be updated according to the method shown in Fig. 5.
- Step-5: The sorted frequency and its symbol are output. For example, Cell_255 has a frequency of "1," and the corresponding symbol is "100."

The updating mechanism of each cell is shown in Fig. 5. If the Shift_EN signal is 1'b1, the frequency and symbol of the corresponding modules will be updated according to the

relationship between FREQ_NEW, FREQ_RIGHT, and FREQ_0. In this way, the values stored in Cell_0~Cell_256 can be updated simultaneously in the same clock cycle.

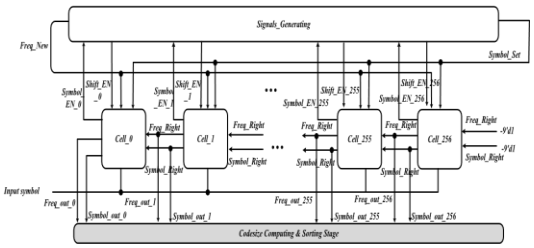

Fig. 6 shows the VLSI architecture of Sorting-Module-1.

The Signal Generating block counts the frequencies of the input symbols according to Step-3 in Fig. 4, where the Symbol_Set's value is consistent with the input symbol. Cell_0~Cell_256 generate sorted results based on the methods shown in Step-4 and Step-5 of Fig. 4. The Signal Generating block and the 257 cells are triggered synchronously by the clock and operate in parallel. Therefore, a temporary frequency statistic and its sorted results are obtained at each clock cycle. As a result, when the last symbol is input, the accurate frequency statistics and the final sorted result will be produced simultaneously. Compared with the traditional sequential structure of statistics before sorting, this proposed architecture can save many clock cycles and improve the encoding efficiency.

C. Code-Size Computing & Sorting Stage

The primary operation of this stage is based on the Canonical Huffman encoding algorithm, which consists of the following steps:
1. Select the two symbols V1 and V2 with the minimum and second minimum frequency, respectively, and merge them into a new node.
2. Add the frequencies of V1 and V2 to create the frequency of the new node. Also, increase the code-sizes of V1 and V2 by one.
3. Sort the new node with other symbols and repeat steps 1 and 2 until all symbols have code-sizes.
4. Once all symbols have code-sizes, sort them from smallest to largest to group them.

The stage comprises three primary components - Sorting-module-1, Code-size_sorting, and FSM_code_size - as shown in Fig. 2. The FSM_code_size module controls the other two modules to work in parallel using a finite-state machine as its core.

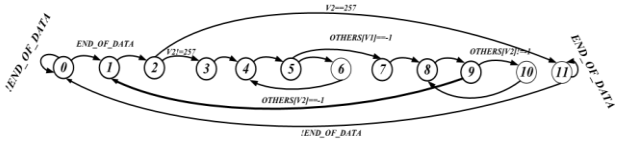

Fig. 7. State transition diagram.

If the signal END_OF_DATA is set to 1'b1, indicating that all 256 frequency sets have been ordered by the first stage, the FSM_code_size module will initiate the code-size calculation process, which involves re-ordering the updated frequency queue using Sorting module-1 and ordering the code-size queue executed by the Code-size_sorting module. The FSM_code_size module uses a finite-state machine as the kernel to schedule the other two modules operating in parallel. The state transition diagram of the FSM_code_size is presented in Fig. 7, where state "0" represents the initial state, state "1" is used to find the symbols V1 and V2, states "2" and "3" are used to sum the frequencies of V1 and V2, and states "4"-"10" are used to calculate the code-sizes of V1 and V2, update the new V1 and V2 simultaneously, and send the code-size to the Code-size_sorting module.

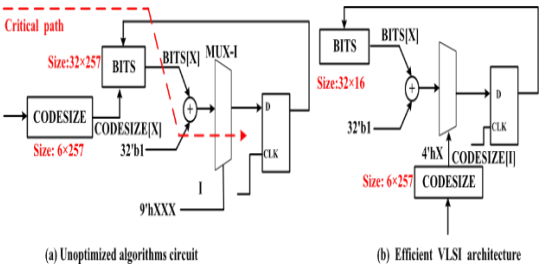

Fig 8:- Mechanism Of Code Size Sorting

It is important to note that the Sorting module-1 of the Frequency-Statistics & Sorting stage is reused in this stage to ensure that the new V1 and V2 can be found at every clock cycle, which can significantly reduce the area of the entire design. Additionally, in traditional methods, the sorting of the codesize is not started until the computation of all symbols' codesizes have been completed, leading to significant time waste. However, in this stage, the computing and sorting of the codesize are performed in parallel.

D. Code-Size-Limiting Stage

The module in question typically features three nested layers of lookup tables, according to standard algorithms. A VLSI design of this module is depicted in Figure 8(a), which includes three large hardware blocks: CODESIZE look-up tables, BITS look-up table, and MUX-I. Unfortunately, these blocks are known to cause issues such as long propagation delay, large area, and high power consumption. To

address these concerns, an optimized architecture has been designed as shown in Figure 8(b). Firstly, the sequential lookup table found in the standard algorithm has been optimized for parallel lookup, ensuring that only one lookup table is on the timing path. Secondly, simulation tests were conducted to identify the presence of redundant content in the BIT table, which is not used in practice. Therefore, the BIT size was reduced from 32*257 to 32*16, using only valid content in the table. This, in turn, reduces the size of the multiplexer. Many simulation experiments have verified the method's functional correctness, and it greatly reduces both the logic delay and the area.

IV. RESULTS AND DISCUSSION

The Verilog HDL was used to describe the proposed VLSI architecture, which was then synthesized using the Synopsys Design Compiler and the SMIC 0.18μm standard CMOS cell library. The synthesis results indicated that the proposed architecture was able to operate at a frequency of 400 MHz. The area of the VLSI architecture was measured at 2,008,766μm2, and the power consumption was 850.84mW. In comparison, the design presented in reference [11] had an area of 340,114μm2 and operated at a frequency of 50 MHz.

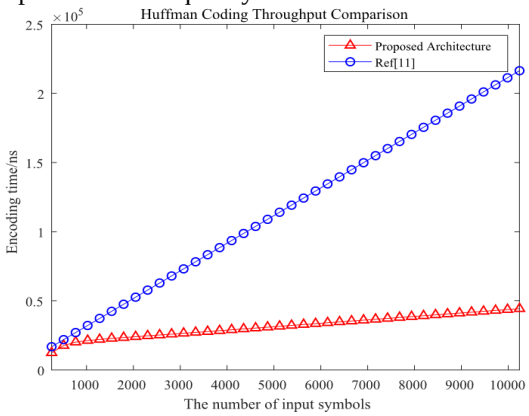

Fig 9:- Throughput comparison of the VLSI architecture.

To evaluate the throughput of the proposed Huffman encoder, reference [11] used 256 8-bit symbols. The encoding time was 840 × 20 ns = 16800 ns. To demonstrate the performance of the proposed encoder, 200 groups of 256 8-bit symbols were tested. The proposed VLSI architecture took 4,952 × 2.5 ns = 12,380 ns for encoding, which was 26.30% faster than the design in reference [11].

To test the high throughput of the VLSI architecture further, the number of 8-bit symbols was increased by 256 at a time, as shown in Fig. 9. When encoding 10,240 8-bit symbols, the proposed VLSI architecture reduced the encoding time by 79.52%, taking 17,726 × 2.5 ns = 44,315 ns. In comparison, the design in reference [11] required 10,824 × 20 ns = 216,480 ns.

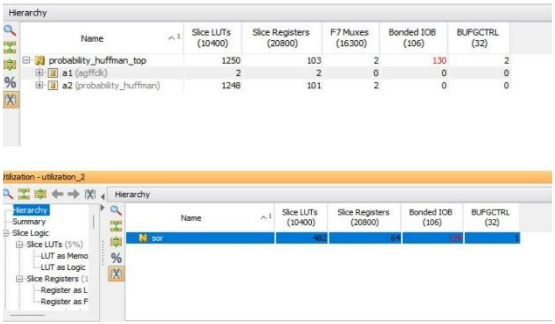

Fig 10:- Area Comparison

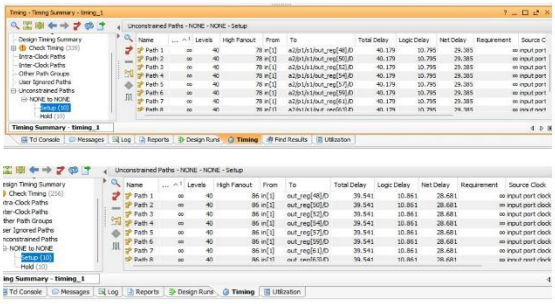

Fig 11:- Delay Comparison

The VLSI architecture proposed in this study was implemented using the Synopsys Design Compiler and described using Verilog HDL. The standard CMOS cell library used for synthesis was SMIC 0.18μm micron. Synthesis results showed that the proposed architecture had an operating frequency of 400 MHz, an area of 2,008,766μm2, and a power consumption of 850.84mW. In comparison, the design presented in [11] had an operating frequency of 50 MHz and an area of 340,114μm2. To test the throughput of the proposed Huffman encoder, the authors used 256 8-bit symbols in [11], while 200 groups of 256 8-bit symbols were used in this study. The proposed architecture reduced the encoding time by 26.30% and 79.52% when encoding 256 8-bit symbols and 10,240 8-bit symbols, respectively, in comparison with the design in [11]. The Kodak24 data set was used to evaluate the performance of the proposed architecture. The average encoding time for the AC coefficients of the Y channel was 537,338.4 ns for the proposed architecture and 4,264,517.5 ns for the design in [11]. The proposed architecture reduced the encoding time by 87.40%. When applied to 100 testing pictures downloaded from Taobao.com with Q value of 100, the proposed architecture improved the compression rate by 12.24% on average. The efficiency of the proposed architecture was evaluated using a ratio, which was

calculated to be 0.653. A smaller value of this ratio indicates better performance of the circuit designed in this study.

V. CONCLUSION

The aim of this paper is to implement a canonical Huffman encoder that includes a frequency sorter, which sorts the input string in ascending order with respect to ASCII values, and the Huffman encoder encodes the string to bits using trees, nodes, and leafs concepts. This proposed architecture can be applied in various applications where data compression is necessary, such as IoT and many other applications. The Frequency-Statistics process and the sorting process operate almost in parallel in the Frequency-Statistics & Sorting Stage, which reduces the time consumption of the pre-scan. The Code-Size Computing & Sorting Stage also performs code-size computing and sorting almost in parallel, reducing the required clock cycle significantly. Compared with the traditional Huffman encoder, the proposed Canonical Huffman encoder circuit in this brief achieves improved coding efficiency.

REFERENCES

[1] Huffman, D. A. “A method for the construction of minimum-redundancy codes,” Proceedings of the IRE, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[2] Sarkar, S. J., Sarkar, N. K., and Banerjee, A. “A novel Huffman coding based approach to reduce the size of large data array,” in Proceedings of the International Conference on Circuit Power and Computing Technologies (ICCPCT), Nagercoil, India, 2016, pp. 1–5.

[3] Liu, Y. and Luo, L. “Lossless compression of full-surface solar magnetic field image based on Huffman coding,” in Proceedings of the IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), Chengdu, China, 2017, pp. 899–903.

[4] Markandeya, N. and Patil, S. “Improve information rate in Thien and Lin’s image secret sharing scheme using Huffman coding technique,” in Proceedings of the International Conference on Computing, Communication, Control and Automation (ICCUBEA), Pune, India, 2017, pp. 1–5.

[5] Patil, R. B. and Kulat, K. D. “Audio compression using dynamic Huffman and RLE coding,” in Proceedings of the 2nd International Conference on Communication and Electronics Systems (ICCES), Coimbatore, India, 2017, pp. 160–162.

[6] Kumar, N. H., Patil, R. M., Deepak, G., and Murthy, B. M. “A novel approach for securing data in IoTcloud using DNA cryptography and Huffman coding algorithm,” in Proceedings of the International Conference on Innovative Information and Embedded Systems (ICIIECS), Coimbatore, India, 2017, pp. 1–4.

[7] Keerthy, S. V., Kishore, T. K. C. R., Karthikeyan, B., Vaithiyanathan, V., and Raj, M. M. A. “A hybrid technique for quadrant based data hiding using Huffman coding,” in Proceedings of the International Conference on Innovative Information and Embedded Communication Systems (ICIIECS), Coimbatore, India, 2015, pp. 1–6.

[8] ISO/IEC 10918-1:1994. Information technology – Digital compression and coding of continuous-tone still images: Requirements and guidelines, 2017.

[9] ISO/IEC 13818-2:2013. Information technology – Generic coding of moving pictures and associated audio information – Part 2: Video, 2019.

[10] Lee, S. J., Yang, K. H., Song, J. S., and Lee, C. W. “An efficient memory allocation scheme for Huffman coding of multiple sources,” Signal Processing: Image Communication, vol. 14, pp. 311–323, Jan. 1999.

[11] Wei, R., and Zhang, X. “Efficient VLSI Huffman encoder implementation and its application in high rate serial data encoding,” IEICE Electronics Express, vol. 14, no. 21, pp. 1–11, Oct. 2017.

[12] Lei, S.-M. and Sun, M.-T. “An entropy coding system for digital HDTV applications,” IEEE Transactions on Circuits and Systems for Video Technology, vol. 1, no. 1, pp. 147–155, Mar. 1991.

[13] [13] T. Kumaki et al. presented a paper on "CAM-based VLSI architecture for Huffman coding with real-time optimization of the code word table" at the IEEE International Symposium on Circuits and Systems in 2005 (vol. 5, pp. 5202–5205).