Efficient VLSI Huffman encoder implementation and its application in high rate serial data encoding

Rongshan Wei\textsuperscript{a)} and Xingang Zhang
College of Physics and Information Engineering, Fuzhou University, Fuzhou, Fujian, 350116, China
a) wrs08@fzu.edu.cn

Abstract: In this paper, we present a new data structure element for constructing a Huffman tree, and a new algorithm is developed to improve the efficiency of Huffman coding by constructing the Huffman tree synchronously with the generation of codewords. The improved algorithm is adopted in a VLSI architecture for a Huffman encoder. The VLSI implementation is realized using the Verilog hardware description language and simulated by Modelsim. The proposed scheme achieves rapid coding speed with a gate count of 9.962 K using SMIC 0.18 micron standard library cells.

Keywords: Huffman coding, VLSI, serial data encoding

Classification: Integrated circuits

References

[1] D. A. Huffman: “A method for the construction of minimum-redundancy codes,” Proc. IRE 40 (1952) 1098 (DOI: 10.1109/JRPROC.1952.273898).
[2] P. K. Shukla, \textit{et al.}: “Multiple subgroup data compression technique based on Huffman coding,” First International Conference on Computational Intelligence, Communication Systems and Networks (2009) 397 (DOI: 10.1109/CICSYN.2009.86).
[3] W. W. Lu and M. P. Gough: “A fast-adaptive Huffman coding algorithm,” IEICE Trans. Commun. 41 (1993) 535 (DOI: 10.1109/26.223776).
[4] H. Park and V. K. Prasanna: “Area efficient VLSI architectures for Huffman coding,” IEEE Trans. Circuits Syst. 40 (1993) 568 (DOI: 10.1109/82.257334).
[5] A. Mukherjee, \textit{et al.}: “MARVLE: A VLSI chip for variable length encoding and decoding,” IEEE International Conference on Computer Design: VLSI in Computer & Processors (1992) 170 (DOI: 10.1109/ICCD.1992.276242).
[6] L. Y. Liu, \textit{et al.}: “Design and hardware architectures for dynamic Huffman coding,” IEE Proc. Comput. Digit. Tech. 142 (1995) 411 (DOI: 10.1049/ip-cdt:19952157).
[7] Y.-S. Lee, \textit{et al.}: “A memory-based architecture for very-high-throughput variable length codec design,” IEEE International Symposium on Circuits and Systems 3 (1997) 2096 (DOI: 10.1109/ISCAS.1997.621570).
[8] K. A. Babu and V. S. Kumar: “Implementation of data compression using Huffman coding,” International Conference on Methods and MODELS in Computer Science (2010) 70 (DOI: 10.1109/ICM2CS.2010.5706721).
[9] H.-C. Chang, \textit{et al.}: “A VLSI architecture design of VLC encoder for high data
rate video/image coding,” Circuits and Syst. 4 (1999) 398 (DOI: 10.1109/ISCAS.1999.780026).

[10] A. Mukherjee, et al.: “Efficient VLSI designs for data transformations of tree-based codes,” IEEE Trans. Circuits Syst. 38 (1991) 306 (DOI: 10.1109/31.101323).

[11] T. Kumaki, et al.: “CAM-based VLSI architecture for Huffman coding with real-time optimization of the code word table [image coding example],” IEEE International Symposium on Circuits and Systems 5 (2005) 5202 (DOI: 10.1109/ISCAS.2005.1465807).

[12] B. Chen, et al.: “Huffman coding method based on number character,” International Conference on Machine Learning and Cybernetics (2007) 2296 (DOI: 10.1109/ICMLC.2007.4370528).

1 Introduction

Huffman coding is a widely used lossless data compression algorithm that assigns shorter codewords to more frequent symbols and longer codewords to relatively rare symbols so as to reduce the size of the original information [1]. Huffman coding is also known as optimal variable length coding, and the algorithm provides the highest lossless compression rate for compression methods encoding symbols separately [2]. The VLSI implementation of Huffman encoders is a perennial topic of interest, and various VLSI designs have been proposed in the literature. However, current Huffman coding schemes require considerable time and computational resources for constructing the Huffman tree and generating codewords, which leads to lower coding speed and higher hardware cost [3]. In this paper, we propose a new data structure for constructing a Huffman tree, and the proposed data structure is employed to develop a new algorithm to improve the efficiency of Huffman coding by constructing the Huffman tree synchronously with the generation of codewords, where the bit width of symbols does not affect the compression time. The proposed new data structure does not require the storage of a Huffman tree in static RAM, which increases available memory resources. High rate serial data encoders are essential for many applications such as image compression, data transmission, and data communication. Therefore, the proposed technological innovation is employed in a high-rate serial data encoding application implemented in VLSI using the Verilog hardware description language (HDL), and simulated using the Modelsim electronic design automation (EDA) tool of Mentor Graphics Inc. The chip was synthesized using the Design Compiler 2009 electronic design automation (EDA) tool (Synopsys Inc.), and the chip occupies a cell area of only 340,114 µm² using SMIC 0.18 micron standard CMOS library cells. Simulation and synthesis results show that the proposed coding scheme provides a greater coding speed and lower hardware cost compared with existing parallel Huffman encoder VLSI implementations [4, 5]. The proposed design requires an average of 3.28 clock cycles to compress an 8 bit symbol, which is much less than that of a previous work [5] that requires an average of 7 clock cycles to compress an 8 bit symbol. Adequate direct comparisons with existing parallel Huffman encoder VLSI implementations in terms of their silicon area are not possible owing to the
different technologies employed, i.e., $3.5 \times 3.5$ mm$^2$ using 1.2 micron CMOS-N standard library cells [4] and $6.8 \times 6.9$ mm$^2$ using 2 micron SCMOS standard library cells [5]. However, the required memory is the primary component of area consumption in each architecture. Therefore, the memory bits required by the various implementations are employed here as a comparison measurement of silicon area. Compared with previous works, which require $256 \times 9 + 64 \times 18 = 3456$ bits of memory [4], and $512 \times 12 = 6144$ bits of memory [5] to process 8 bit symbols, the proposed design requires $256 \times 8$ bits of input symbol memory and $73 \times 10$ bits of temporarily available memory, for a total of $2778$ bits of memory to process 8 bit symbols. This verifies a substantial decrease in silicon area for the proposed architecture.

2 Proposed architecture

Considering that the code elements employed in the conventional data structure are inconvenient for constructing the Huffman tree in synchronization with the process of generating codewords, we propose constructing the Huffman tree using a new data structure element composed of frequency, identification, and symbol fields, all of which are expressed in binary code. The fields in the new data structure are depicted in Fig. 1.

Fig. 1. New data structure element for a source message comprising $N$ symbols composed of an alphabet of $n$ symbols, where $F = \sum_{k=0}^{n-1} x_k$, which depends on the cumulative sum of all symbol values $x_k$ after quantizing and encoding.

Here, the frequency field encodes the frequency of each alphabet symbol. The identification field records the position of each alphabet symbol on the Huffman tree, and the initial value of the identification field is zero. The symbol field encodes each alphabet symbol. The number of elements employed for constructing the Huffman tree is equivalent to the number of symbols in the alphabet. The uniqueness of the proposed data structure is that the Huffman tree is constructed using the full value of the combined fields of the data structure (denoted henceforth as the element value) rather than just the frequency. For example, an element value of 000101000.0000.00010000001 in binary would represent a decimal element value of 655,425. The bit widths of the individual fields of the proposed data structure are defined for a general message comprising $N$ symbols composed of an alphabet of $n$ symbols.
Accordingly, the bit width of the frequency field is

$$\text{int}[\log_2(N)] + 1, \quad (1)$$

where \(\text{int}[\cdot]\) represents the integer part of the value. The bit width of the identification field is

$$\text{int}[\log_2((2n - 2)/2)] + 1, \quad (2)$$

which is based on the maximum number of times the two smallest element values would be selected as nodes while constructing the Huffman tree. The bit width of the symbol field is

$$\text{int}[\log_2(F)] + 1, \quad (3)$$

where \(F = \sum_{k=0}^{n-1} x_k\), which represents the cumulative sum of all assigned symbol values \(x_k\) after quantizing and encoding. To clarify, examples are provided in the following paragraph.

An example of the proposed data structure scheme is presented in Fig. 2 based on an 8 bit ASCII code input, where \(N = 256\) and \(n = 10\). The alphabet symbols and their frequencies are given at the top of the figure.

| Symbol | Frequency |
|--------|-----------|
| A      | 20        |
| B      | 40        |
| C      | 30        |
| D      | 50        |
| E      | 10        |
| F      | 12        |
| G      | 33        |
| H      | 25        |
| I      | 5         |
| J      | 31        |

**Leaf elements**

| Leaf | Elements                      |
|------|-------------------------------|
| A    | 000010100_0000_0001000001     |
| B    | 000101000_0000_0001000010     |
| C    | 00011110_0000_0001000011      |
| D    | 000100010_0000_0001000100     |
| E    | 000010100_0000_0001000010     |
| F    | 000011000_0000_0001000110     |
| G    | 000100001_0000_0001000111     |
| H    | 000011000_0000_0001001000     |
| I    | 000001010_0000_0001001001     |
| J    | 000011111_0000_0001001010     |

*Fig. 2. An example of the proposed data structure scheme.*

The bit width of the frequency field is then equal to the integer part of \(\log_2(256) + 1 = 9\). The number of nodes (except for the root node) required for constructing the Huffman tree is \(2 \times 10 - 2\), such that the maximum number of times the two smallest element values are selected as nodes while constructing the Huffman tree is \((2 \times 10 - 2)/2 = 9\), so the bit width of the identification field is \(\text{int}[\log_2((2 \times 10 - 2)/2)] + 1 = 4\). Quantizing and coding the alphabet symbols result in continuous binary code values of 01000001–01001010, such that the cumulative sum of all symbol values in a decimal format is \(F = 65 + 66 + 67 + 68 + 69 + 70 + 71 + 72 + 73 + 74 = 695\), so the bit width of the symbol field is \(\text{int}[\log_2(695)] + 1 = 10\).

As shown at the bottom of Fig. 2, a listing of 10 elements is required for constructing the Huffman tree. Here, we note that, while all leaf elements can be encoded using a bit width of only 4, use of the element value to construct the
Huffman tree requires a predefined bit width for the symbol field to accommodate non-leaf elements, which employ a symbol field that is the sum of the symbol fields of their branches. This is illustrated in Fig. 3, which describes the process of generating codewords using the proposed data structure elements based on the example presented in Fig. 2. For example, the symbol fields of non-leaf elements N7 and N8 are 0101011011 and 0101011100 in binary, such that their sum, which represents the root node, are $0101011011 + 0101011100 = 1010110111$, or 695 in decimal.

![Fig. 3. The process of generating codewords using the proposed data structure elements.](image)

### 3 Encoding algorithm

The key task of a Huffman encoder is constructing the Huffman tree. The Huffman tree is a directed binary tree in which each branch represents a codeword and each leaf node represents a symbol. Each non-leaf node of the tree is a set containing all the symbols in the leaves that lie below the node. In addition, each symbol at a leaf node is assigned a weight that depends on the relative frequency of the symbol within the source message, and each non-leaf node contains a weight value that is the sum of all the weights of the leaves lying below it. Current Huffman coding methods employ the symbol weights for constructing the Huffman tree, which requires considerable computation time for determining the relative position of each symbol on the tree [6, 7]. Therefore, the present work employs the data structure elements proposed in the previous section to construct the Huffman tree, which simplifies the complexity of the construction process by synchronously construct-
ing the Huffman tree with the process of codeword generation. The memory in our design is used to store input symbols and for the temporary storage of intermediate variables in the calculation process. The newly proposed data structure allows the architecture to obtain all node positions in the process of constructing the Huffman tree with minimal computation. Moreover, the proposed data structure eliminates the requirement for storing the Huffman tree in static RAM. Hence, few simple arithmetic operations are needed in our design.

To clearly describe how the developed encoder algorithm functions, the following presents details based on the example presented in Figs. 2 and 3. The process of constructing the Huffman tree synchronously with the generation of codewords is illustrated in Fig. 4.
Fig. 4. The process of constructing the Huffman tree synchronously with the generation of codewords.
The encoding algorithm flow diagram is illustrated in Fig. 5.

![Encoding algorithm flow diagram](image)

**Fig. 5.** Encoding algorithm flow diagram.

i) Parse the source message serially to obtain each alphabet symbol and its frequency. A counter is established for each symbol of the source alphabet and for the total number of symbols in the message, and each counter is initially set to 0. Each time a symbol occurs in the source message, its counter is increased by 1, and
the counter for the total number of symbols in the message is increased by 1. These source message statistics are employed to construct the proposed data structure elements, as shown in Fig. 2.

ii) Sort the n elements of the source message according to their element values and create a new element with a value that is the sum of the values of the two lowest elements, and increase the value of the identification field by 1 in the new element. The two selected elements are employed as nodes in the Huffman tree.

iii) The two lowest-value elements are removed from the original set, and the new element is inserted into the set.

iv) Repeat steps ii) and iii) until only two elements remain in the set, and the last two elements become the last two nodes of the Huffman tree. In this way, a total of \(2^n - 2\) nodes are generated with all nodes arranged in descending order. The value of the root node is the sum of the values of the final two elements, and the value of the identification field is increased by 1 in the root node. The root node of the Huffman tree is assigned the single-bit codeword “1”.

v) Select a target node from the node set beginning with the root node in descending order, and assign its codeword as the current codeword after decreasing the value of the identification field by 1. Then, search the remaining node set to find the two nodes that have a combined value equal to the value of the target node. If found, these two nodes are assigned as the left and right branches of the target node, with the node of the larger value forming the right branch. Otherwise, the target node is a leaf node, and the process proceeds to step vii).

vi) The current codeword is concatenated with “1” as the right branch codeword and concatenated with “0” as the left branch codeword, and these codewords are assigned to the nodes on the right and left branches, respectively.

vii) Repeat steps v) and vi) until all nodes in the node set have been considered. In this way, the Huffman tree is constructed synchronously with the generation of codewords.

viii) Remove the leading “1” indicative of the root node from all codewords, to obtain the Huffman code for each leaf node.
4 Application in high rate serial data encoding

The architecture for the proposed Huffman serial data encoding application is illustrated in Fig. 6.

After constructing the Huffman tree, the original elements saved previously are used to identify the leaf nodes from the Huffman tree and create the Huffman code table. The original elements are the 10 elements composed of the original frequency, identification, and symbol fields. The Huffman code table and the bit stream of the input source message presented for compression are output. The output data format is one bit per clock cycle, which meets the requirements of most Huffman decoders [8, 9].

The VLSI implementation of the encoder was realized by Verilog code, and synthesized using the Design Compiler 2009 EDA tool. The coding performance was evaluated by calculating the number of clock cycles required to complete the encoding of an input serial data sequence comprising 256 symbols based on an alphabet composed of the 10 letter sequence A–J, each of which is represented by a 8-bit binary number in the range of 01000001 to 01001010 based upon its position in the English alphabet [10, 11, 12]. The serial data Huffman encoder timing diagram is shown in Fig. 7, and is summarized as follows.

(1) The start signal is high after reset, and the 256 symbols of the serial data are input consecutively, where the data_in data width is 8 bits, and the input requires 256 clock cycles.

(2) A high code_start signal initiates the Huffman coding process and a high code_done signal indicates the completion of the encoding process.

(3) Output the Huffman code table and the encoding result.

The clock cycle frequency employed in the simulation was 50 MHz. The simulation results show that the circuit required only 840 clock cycles to encode 256 8-bit symbols.
5 Conclusion

In this paper, we proposed a new data structure for constructing a Huffman tree, and the proposed data structure was employed to develop a new algorithm to improve the efficiency of Huffman coding by constructing the Huffman tree in synchronization with the process of generating codewords. The proposed data structure eliminates the need for storing the Huffman tree in static RAM, which decreases the memory requirements. In addition, we presented the architecture for a high-rate serial data encoder, and verified its performance. The VLSI implementation of the high-rate serial data encoder design was realized by Verilog code, simulated by Modelsim, and synthesized using the Design Compiler 2009 EDA with the SMIC 0.18 micron standard CMOS cell library. The implementation includes a gate count of 9,962 K, and its cell area is 340,114 µm². The circuit was demonstrated to require only 840 clock cycles to encode 256 8-bit symbols. The proposed Huffman coding architecture can be applied to many cost-efficient and high-compression-rate lossless data compressors.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 61404030).