# EFFICIENT HUFFMAN DECODING WITH TABLE LOOKUP

*Mohamed F. Mansour*

DSPS R&D Center, Texas Instruments Inc., USA
mfmansour@ ti.com

## ABSTRACT

We describe an efficient algorithm for Huffman decoding using table lookup. The algorithm is optimized for ROM-based Huffman decoding. It is a two-step process of prefix template matching followed by a direct table access. We propose an efficient algorithm for choosing the prefix templates according to different optimization criteria. Also, we propose different implementations for the prefix template procedure.

***Index Terms***— Huffman decoding, Table lookup

## 1. INTRODUCTION

Huffman codes [1] have been widely used for source coding and have shown high efficiency in exploiting the source redundancy. Huffman codes along with run-length codes have been widely used in most international multimedia standards (e.g., MPEG and ISO standards [5], [7]).
Huffman decoding can be implemented with a lookup-table (LUT) [2] or multiple lookup-tables [3]. If a single LUT is used, the decoder throughput can be one codeword per cycle whereas the throughput for multiple lookup tables is not deterministic and in the worst case it equals the number of lookup tables (assuming each LUT is processed in a single cycle). The single LUT approach is usually adopted in high efficiency Huffman decoder while the multiple LUTs approach is usually used in low-power systems.
In this work, we propose a novel LUT-based approach for Huffman decoding. The decoder has an LUT for a set of prefix templates for the table codewords. Each prefix template is associated with a direct access table for the children codewords. During decoding, the input bits after the prefix template are used to directly address the associated codeword table to retrieve the correct codeword and its length. We propose a novel approach for designing the prefix templates which depends on a generic optimization criterion that can be adjusted to the system. We propose different criteria that can be employed in typical systems.

## 2. DECODING PROCEDURE

### 2.1. Background

Any Huffman code can be represented by a non-balanced binary tree. The tree leaves represent the codewords of the code. Any codeword has three attributes: the length, the value, and the corresponding source symbol. An example of a Huffman table of size 8 is shown in table 1 and the corresponding tree representation is shown in Fig. 1. The value of each internal node in Fig. 1 is the sum of its children values and it is a measure of the internal node probability.

| Symbol | Codeword | Length | Symbol | Codeword | Length |
|---|---|---|---|---|---|
| 1 | 00111 | 5 | 5 | 010 | 3 |
| 2 | 00110 | 5 | 6 | 000 | 3 |
| 3 | 0010 | 4 | 7 | 11 | 2 |
| 4 | 011 | 3 | 8 | 10 | 2 |

Table 1. Example of Huffman Table of size 8.

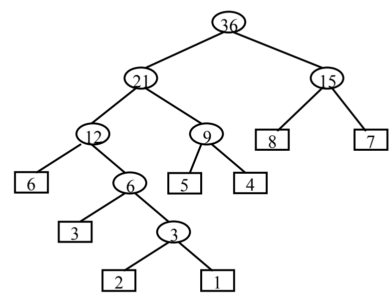


Figure 1. Huffman Tree of the code in table 1

In general, the length of each codeword in the Huffman table is inversely proportional to the probability of the corresponding source symbol.
In our implementation, we use a set of prefix templates that represent some internal nodes in the Huffman tree. Each prefix template is parameterized by three attributes:

1. *length* ($L$): the length of the prefix value
2. *value* ($V$): the bit value of this prefix
3. *Maximum child length* ($M$): the maximum length of the template children codewords.

For example, the internal node with label 12 in the Huffman tree of Fig. 1, has the following attributes: $L$ = 2, $V$ = “00”,

$M = 5$ (which is equivalent to codewords 1 and 2). The choice of the prefix templates is discussed in section 4.

### 2.2. Decoder structure

Each prefix template is associated with a sub-table that contains all children codewords. The size of the sub-table is $2^{M-L}$, where $M$ and $L$ are the attributes of the prefix template as defined earlier. The indexing within the sub-table is done using the last $M$-$L$ bits of the input word that follow the $L$ bits of the prefix template. The sub-table is filled with the children codewords of the templates with possible repetition of certain codewords. For example, if the node with frequency 12 in the Huffman Tree of Fig. 1 is selected as a prefix template, then the size of its sub-table will be 8 and it is organized as:

| Sub-table Address | No. of symbols | Sub-table Address | No. of symbols |
|---|---|---|---|
| 000 | 6 | 100 | 3 |
| 001 | 6 | 101 | 3 |
| 010 | 6 | 110 | 2 |
| 011 | 6 | 111 | 1 |

Table 2. Memory map of the sub-table example

In this example we have only four codewords while the overall memory is eight, i.e., we have a redundancy factor of two. This redundancy is minimized by proper choice of the prefix templates as will be discussed in section 4.

Note that, each symbol in the prefix sub-table has two attributes: the value of the corresponding source symbol and the codeword length.

The prefix templates are chosen such that, no template is a prefix of another template. Therefore when we match the input bitstream with the prefix templates, one and only one template will be matched. This is also a design criterion that is considered while generating the prefix templates.

### 2.3. Decoding Procedure

The decoding process consists of three basic steps:

1. Matching the prefix templates
2. Getting the codeword symbol from the sub-table of the selected prefix template using the bits that follow the template for indexing within the sub-table.
3. Progressing in the input bitstream by a number of bits equals the codeword length to decode the following symbol.

In step 1, to match a certain prefix template of attributes ($L$,$V$,$M$), the first $L$ bits of the bitstream should equal $V$. Two attributes are associated with each prefix template, which are, the number of indexing bits in its subtable, and the starting address of its subtable. The overall decoding procedure is illustrated in Fig. 2.

The input module is responsible for aligning the input bitstream so that decoding starts at the correct word boundary. The alignment is controlled by the length of the last decoded codeword. The alignment procedure is similar to previous algorithms (e.g., [2], [4]). The input to the prefix LUT module has a length $L_{max}$ which is the maximum template length. The inputs to the sub-table index generator are the attributes $L$ and $M$–$L$ of the matched template and $M_{max}$ bits of the input bitstream which is the maximum codeword length in the Huffman table. The output is the $M$–$L$ bits from the bit stream starting from the $(L+1)^{st}$ bit.

The prefix LUT module is the most energy-demanding module in the decoder. The objective of this work is to propose efficient implementation of this module as will be discussed in the following two sections.

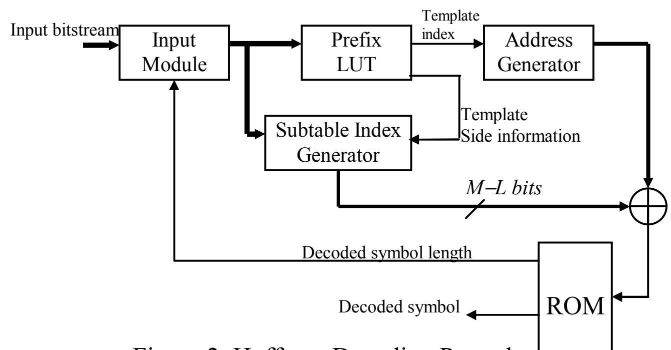


Figure 2. Huffman Decoding Procedure

## 3. PREFIX LUT IMPLEMENTATION

The prefix LUT module can be implemented in different ways that depend on the structure of the Huffman table and the target application.

The first choice is to use a programmable logic array (PLA) as suggested in [4]. The cost of the PLA is proportional to the number of templates which is significantly less than the size of the Huffman table (which is used in [4]). In this case, the prefix template matching can be performed in a single cycle regardless of the matched template.

The second choice is to use a single comparator for matching the prefix templates one at a time. This would require a number of registers equals the number of templates. To minimize the matching time, the templates are arranged in descending order according to their probabilities. The template probability equals the sum of the probabilities of all its children (assuming source symbols are independent). In Huffman codes the probability of each source symbol is inversely proportional to the length of the corresponding codeword. Therefore the probability of each template is inversely proportional to the sum of the lengths of its children codewords. The more accurate probability for each template is obtained by scaling all individual probabilities in (1) such that they sum to one. In the worst case the number of cycles for prefix LUT equals the number of the templates. However, the average number of cycles is much less and equals

$$N_{av} = \sum_{i=1}^{M} i.p(i) \qquad (1)$$

Where $p(i)$ denotes the probability of the $i^{th}$ ordered template and $M$ is the number of templates.

The prefix tree can be converted to a balanced tree where all the leaves are in the last tree level. In this case, the template matching can be viewed as a binary tree search. At each node we perform a binary comparison upon which we decide the next child node to investigate. If the number of templates is a power of two then we have a complete binary tree. The number of cycles needed for template matching equals the tree depth if one comparison is performed per cycle.

In particular, assume we have 16 prefix templates, then we have a binary tree of height 4 and 15 internal nodes. Each internal node is associated with a reference parameter, or in other words a *threshold*, that is compared by the input stream, i.e., we need a total of 15 thresholds. We illustrate the previous arguments by an example constructed using one of the mp3 Huffman tables c*ode table* 24 [5], which has 256 codewords. After running the prefix template construction algorithm to be described in section 4, we get the templates listed in table 3.

| template | Value (binary) | Length | template | Value (binary) | Length |
|---|---|---|---|---|---|
| 0 | 000 | 3 | 8 | 01001 | 5 |
| 1 | 1000 | 4 | 9 | 101 | 3 |
| 2 | 0100000 | 7 | 10 | 00101 | 5 |
| 3 | 0110 | 4 | 11 | 010001 | 6 |
| 4 | 00100 | 5 | 12 | 01011 | 5 |
| 5 | 01010 | 5 | 13 | 0111 | 4 |
| 6 | 11 | 2 | 14 | 1001 | 4 |
| 7 | 0011 | 4 | 15 | 0100001 | 7 |

Table 3. Prefix templates of the mp3 Code Table 24

The first step to compute the thresholds is to order the prefix templates according to their values. For example, in the above table the maximum template length is 7, therefore we augment each template of length $L$ bits by "$7-L$" zeros (from right). Then we order the augmented templates. After ordering, we apply successive refinement to get the thresholds. In particular, we take the eighth codeword as the first level threshold, and the fourth and twelfth codewords as the second level thresholds and so on. For the above tables, the threshold binary tree is as shown in Fig. 3, where each internal node is associated with the corresponding threshold.

The implementation of the search algorithm of the above balanced thresholds tree requires four comparators and a set of multiplexers to decide each comparator reference value.

The balance tree implementation is also convenient if the Huffman decoder is implemented on a general-purpose hardware, e.g., a digital signal processor. In this case, the templates may be stored in ROM and the comparators are replaced by subtraction which is common on all general-purpose hardware. In this case, we search may be optimized by stopping the search if the difference with between the input and the reference threshold is zero (because the thresholds are themselves valid templates). In this case the average matching cycles is:

$$N_{av} = \gamma.\sum_{i=1}^{D} i.\sum_{j=1}^{i} p(T_j^{(i)}) \qquad (2)$$

Where $\gamma$ is the number of cycles per comparison, and $p(T_j^{(i)})$ is the probability of the $j^{th}$ template at the $i^{th}$ tree level.

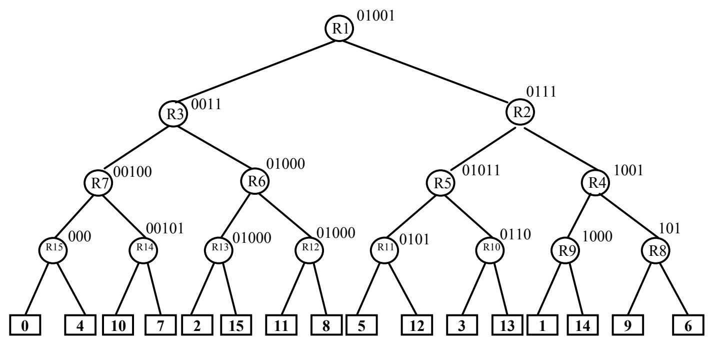


Figure 3. Balanced thresholds tree of table 3 with the codewords as leaves

## 4. PREFIX TEMPLATES SELECTION

The proper design of the prefix templates is crucial for the overall efficiency of the algorithm. In the following, we describe an algorithm for generating a fixed number of templates such that a certain objective function is optimized. The algorithm is similar to the *k-means* algorithm for constructing the codebooks in vector quantization schemes [6]. The inputs to the algorithm are the Huffman table and the maximum number of prefix templates $N$. The output is the prefix templates. The algorithm proceeds as follows:

1. Start with the root node of the Huffman tree and split it to its two children, add them to the templates table, and set the number of templates to two.
2. For each node in the templates table compute the objective function
3. Pick the template with the worst value of the objective function and split it to its two direct children by padding zero and one to the current template value and increase its length by one. Then, increase the number of templates by one.
4. If the number of templates equals $N$ or if the algorithm converges, stop. Otherwise go to step 2.

The algorithm terminates if the objective function reaches a global optimal value; otherwise it is terminated when the number of templates reaches its maximum.

The objective function varies according to the system requirements and the structure of the prefix template LUT module. For example, if the template matching process is performed using successive matching, then the objective function is to minimize the overall matching cycles in (2) for a given limit of the storage space of the sub-tables. Note that, the minimum time would be when we have a single template, but in this case the sub-table size will be $2^{L_{\max}}$ words, where $L_{max}$ is the maximum codeword length. The objective function in this case is to minimize (2) subject to the maximum storage limit. At each iteration we compute the objective function after splitting each node, and split the node that gives minimal increase in the objective function. The optimization iteration stops when the overall subtables size is below the maximum limit.

In some Huffman tables (e.g., in JPEG and MPEG-2 video tables), the Huffman tree is very sparse away from the main branch (the branch of all ones or all zeros), e.g., consider the following Huffman table of size 16, from the JPEG standard [7] (table K.3 for luminance DC coefficients) :

| Symbol | codeword | Symbol | codeword |
|---|---|---|---|
| 0 | 00 | 6 | 1110 |
| 1 | 010 | 7 | 11110 |
| 2 | 011 | 8 | 111110 |
| 3 | 100 | 9 | 1111110 |
| 4 | 101 | 10 | 11111110 |
| 5 | 110 | 11 | 11111111 |

Table 4. Table K.3 for luminance DC coefficients in the JPEG Standard

In this case, the prefix templates may be chosen such that it is either zero or a string of ones. The template matching procedure in this case is reduced to counting the number of leading ones (or leading zeros for zero-leading tables). This procedure is in general very efficient for Huffman tables used in video and image standard. However, in most audio standards, minimum variance Huffman tables are frequently encountered and these templates will be memory inefficient.

## 5. DISCUSSION

We propose a generic algorithm for universal variable length decoding. The algorithm is suited for Huffman tables in current international multimedia coding standards. However, it is general to decode any existing prefix code. The algorithm generates a set of prefix templates and associates each codeword to one of the templates. The decoding process includes template matching and codeword retrieval using direct table access. We proposed an efficient algorithm for generating the prefix templates to optimize a generic objective function and we gave several examples of the objective function. Moreover, we described efficient algorithms for implementing the template matching using hardware and hybrid software/hardware approaches.

We evaluated the algorithm on a general purpose digital signal processor using the objective function of minimizing the overall memory requirement. The evaluation was on all the Huffman tables of the two most common MPEG audio standards, namely, mp3 and AAC. The results are summarized in Table 5. The redundancy in this worst case is 1.82; whereas if we use the templates of regular Huffman tables (that is all zeros or all ones) the redundancy is 9.

| | Total Codewords | Algorithm Requirement (words) | |
|---|---|---|---|
| | | N = 16 | N = 20 |
| Mp3 | 1378 | 2516 | 2294 |
| AAC | 1362 | 1692 | 1672 |

Table 5. Total Storage requirement for the proposed algorithm with AAC and mp3 audio standards

The proposed decoding algorithm can be adapted in different ways according to the underlying application. For fast Huffman decoding with regular Huffman tables, the implementation of the prefix template matching with counting the number of leading ones or zeros is the most appropriate. For fast Huffman decoding with minimum variance Huffman tables, the prefix LUT using PLA is recommended along with minimum sub-table storage. For low-power Huffman decoding either the balanced tree template matching or a multi-step comparison (using a single comparator) with the templates designed to minimize the average number of decoding cycles.

## REFERENCES


[1] D. Huffman, "A method for the construction of minimum redundancy code", Proc. IRE, vol. 40, pp. 1098-1101, 1952.

[2] S. Choi, and M. Lee, " High Speech Pattern Matching for a Fast Huffman Decoder", IEEE Transactions on Consumer Electronics, vol. 41, pp. 97-103, February 1995.

[3] S. Cho, T. Xanthopoulos, and A. Chandrakasan, "A low-power Variable Length Decoder for MPEG-2 Based on Nonuniform Fine-Grain Table Partitioning", IEEE Transactions on VLSI systems, vol. 7, no. 2, pp. 249-257, June 1999.

[4] S. Lei, and M. Sun, " An entropy coding system for digital HDTV applications", IEEE Transactions on Circuits and Systems for Video Technology, vol. 2, No. 1, pp. 147-155, March 1991.

[5] ISO/iec 11172-3:1993 "Information Technology – coding of Moving Pictures and associated audio for Digital Storage Media at up to about 1.5 Mbit/s –part 3 : Audio "

[6] A. Gersho, and A. Gray, "Vector Quantization and Signal Compression", Kluwer Academic Publications, 1991.

[7] CCITT Recommendation T.81, "Digital compression and coding of continuous-tone still images", 1992.