Fault Tolerant Huffman Coding for JPEG Image Coding System

Cung Nguyen
Department of Electrical and Computer Engineering,
University of California, Davis, CA 95616
Phone: (530) 756-3243 – Fax: (530) 752-8428
email: cunguyen@ece.ucdavis.edu

Abstract – In this paper, the tolerance of Huffman Coding to memory faults is considered. Many pointer-based and array-based data structures are highly nonresilient to faults. A single fault in a memory array or a tree node may result in loss of entire data or an incorrect code stream. In this paper, a fault tolerant design scheme is developed to protect the JPEG image compression system.

1 Introduction

Huffman codes are widely used and very effective technique for compression data; saving of 20% to 90% are typical, depending on the characteristics of the data being compressed. Huffman coding starts by assign the shorter code words for more probable symbols and longer codewords for the less probable symbols. This variable-length codewords belong to entropy coding scheme. Huffman coding is one of the entropy coding techniques that JPEG uses in its compression standard. There are only the codes in which no codeword is also a prefix of some other codeword. Such codes are called prefix codes. It is possible to show that the optimal data compression achievable by a character code can always be achieved with a prefix code.

The Huffman code assignment procedure is based on coding tree structure. This tree is developed by a sequence of pairing operations in which the two least probable symbols are joined at a "node" to form two "branches" of the tree. As the tree is constructed, each node at which two branched meet is treated as a single symbol with a combined probability that is the sum of the probabilities for all symbols combined at that node. Such described tree is so called a binary tree structure. In a binary tree structure there is a certain probability that a node or a link can fail. The tree structure is physically static; if any node or link fails, the tree structure no longer exists. For many applications, the binary tree structure must be maintained during execution of a task. Redundant nodes and links must be employed for providing fault tolerance against node and links failures. Fault tolerance issues in tree architectures have been studies by several researchers and design procedures for k-fault-tolerant tree structures have been developed.

JPEG image compression standard applies the Huffman table instead of tree structure, the fault tolerant design for this coding method must be modified.In this correspond, the fault tolerance issues in the Huffman coding structure which using code table is considered. Each table has a table head, which is the address of the first item. Table head provides the reference for accessing to the entire table's content by computing the actual address by the displacement from head to an item in the table. Protection of table head requires

a small computation cost. With our design scheme, error detection must be available for the required table lookup procedures at the Huffman encoder. The codewords have variable lengths so the encoder cannot employ fixed-width memory. Huffman's greedy algorithm uses a table of the frequencies of occurrence of characters to build up an optimal way of representing each character as a binary string.

In this report, the JPEG Huffman entropy coding system is analyzed. Beside that, the redundancy parity checking codes are also introduced. These steps provide the background information for the further fault-tolerant schemes to protect against failures. Finally, the computer simulation results for the fault protection scheme are presented and the detailed reliability analysis and estimation are performed using C++ and MATLAB programming.

2 Constructing of Huffman Code for JPEG

To simplify the problem, this section only discusses the JPEG Huffman entropy coder implementations for the baseline mode operation. Baseline sequential coding is for images with 8-bit samples and uses Huffman coding only, and its decoder can store only two sets of Huffman tables (one AC table and DC table per set). Prior to entropy coding, there usually few nonzero and many zeros-valued coefficients. The task of entropy coding is to encode these few coefficients efficiently. The description of Baseline sequential entropy coding is given in two steps: conversion of quantized DCT coefficients into an intermediate sequence of symbols and assignment of variable-length codes to the symbols.

In the intermediate symbol sequence, each nonzero AC coefficients is represented in combination with the "runlength" of zero-valued AC coefficients which precede it in the zig-zag. Each such runlength-nonzero coefficient combination is represented by a pair of symbols:

$$\begin{aligned} symbol\text{-}1 &\Leftrightarrow \text{(RUNLENGTH, SIZE)} \\ symbol\text{-}2 &\Leftrightarrow \text{(AMPLITUDE)} \end{aligned} \tag{1}$$

$symbol$-1 represents two pieces of information, RUNLENGTH and SIZE. $symbol$-2 represents the single piece of information designated AMPLITUDE, which is simply the amplitude of the nonzero AC coefficient. RUNLENGTH is the number of consecutive zero-valued AC coefficients in the zig-zag sequence preceding the nonzero AC coefficient being represented. SIZE is the number of bits used to encode AMPLITUDE.

RUNLENGTH represent zero-runs of length 0 to 15. Actual zero-runs in the zig-zag sequence can be greater than 15, so the $symbol$-1 value $(15, 0)$ is interpreted as the extension symbol with runlength $= 16$. There can be up to three consecutive $(15, 0)$ extensions before the terminating $symbol$-1 whose RUNLENGTH value complete the actual runlength. The terminating $symbol$-1 is always followed by a single $symbol$-2 except for the case in which the last run of zeros include the last (63rd) AC coefficient. In this frequent case, the special $symbol$-1 value $(0, 0)$ means EOB (end-of-block) symbol, which terminates the $8{\times}8$ sample block.

The DC Huffman code is partitioned into two bit-groups: The first bit-group consists of consecutive 1s ending with a single 0. Let k denotes the number of 1-bits in the first group; The second bit-group (with k bits length) represents the difference ($\triangle_{DC}$) between the DC value of the current block and that of the previous block. Let B denotes the equivalent decimal value of the second bit-group. If $B \geq 2^{k-1}$, then $\triangle_{DC} = B$. Otherwise, $\triangle_{DC} = B - 2^k + 1$. Therefore, the DC value of the current block is the sum of $\triangle_{DC}$ and the DC value which is already decoded from the previous block.

The possible range of quantized AC coefficients determine the range of values which both the AMPLITUDE and the SIZE information must be represent. If the input data are N-bit integers, then the non-fractional part of the DCT coefficients can grow by at most 3 bits. Baseline sequential has 8-bit integer source samples in the range $-2^7, 2^7 - 1$, so quantized AC coefficient amplitudes are covered by integers in the range $[-2^{10}, 2^{10} - 1]$. The signed-integer encoding uses symbol-2 AMPLITUDE codes of 1 to 10 bits in length, so SIZE also represents values from 1 to 10, and RUNLENGTH represents values from 0 to 15 as discussed previously. Figure provides an example of baseline coding of a single 8×8 sample

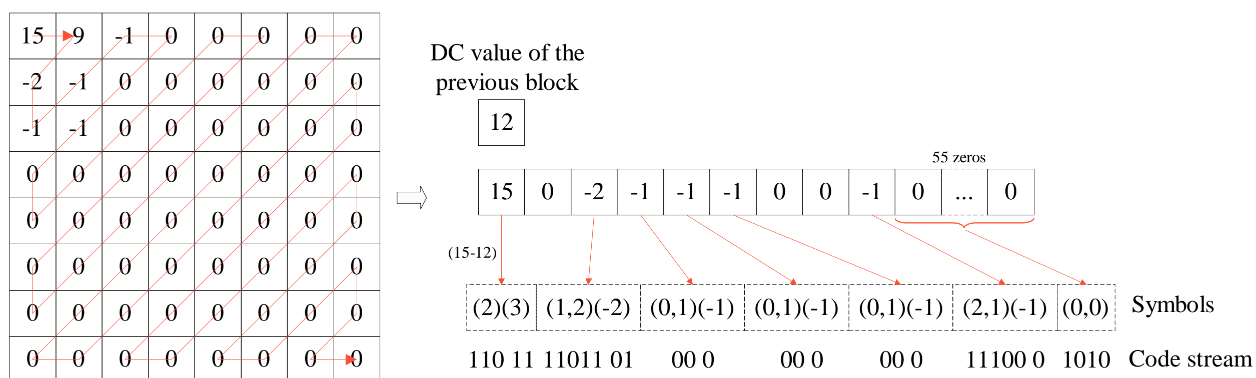

Figure 1: Huffman coding processes for an 8×8 DCT block.

block. For the demonstration purpose, it omits the operation of complete JPEG interchange format information (parameters, headers, quantization). The 8×8 block shows the resulting quantized DCT coefficients of an image component. It involved the zig-zag arrangement, the appropriate intermediate symbol (RUNLENGTH, SIZE)(AMPLITUDE) pairs for the AC coefficients. More information about base-line run-length and Huffman code can be found in [2] (pg. 190-201 and 441-449). Finally, the code stream for the block is formed by applying the code Huffman code tables as shown in Tables 3 and 4.

3 Fault Tolerant Design for the JPEG Huffman Coding System

From the Huffman coding process as described in the previous section, the Huffman code tables play an important role in constructing the code stream. Before coding, the Huffman code tables are loaded into memory and will be available at all time. Since the code table may be error due to the flipped bits inside the

memory cells. This flipped will change a valid code into another code that may or may not already existed in the code table. In any circumstance, the changing of code table will change the entire output stream. Each code in the table associates with the row an column indices. These indices are not only used to find the memory location for the code itself, but they also provide information about the zero runlength in the zigzag sequence and the bit-size used to code the non-zero values of coefficients. A flipped bit occurred inside the code table could change entire context of the block and yield a totally wrong code stream. Decoder will not able to decode the block, even though only a partial of the block. Since the code table at the decoder suppose to de identical to that of the encoder, the changing the code table at the encoder will not allow the decoder to continue its decoding process as soon it encounters the corrupted code stream.

3.1 Single bit Error Detection Design for Huffman Code Table

To analyze the effects of the errors in the code table, assume that the decoder will will be able to skip the rest of bits represent for the remain coefficients of the current 8×8 block as soon an error is detected. The rest coefficients of the current block are filled with zeros and starts to the next block. With this decoding method, the reconstructed image is degraded and large of its details will be lost. Figure 2 shows the original

Figure 2: Original (*left*) and reconstructed (*right*) images under the memory fault effects. In this case, the error correction function was not activated. The coding for an 8×8 DCT blocks is skipped when error is detected in the Huffman code table

and reconstructed images using the above decoding strategy in case flipped bits occurred in the encoder's Symbol-1 code table (In this case, the code bits at (C1,Z4) flipped from 111011 to 111100, (C1,Z5) flipped from 1111010 to 1111011 and the code at (C2,Z0) flipped from 01 to 10). The reconstructed mean square errors for Red, Green and Blue components are 179.5, 206.8, and 246.9, respectively. The bypass of error coefficients creates artifact phenomenon due to the great jumps in image levels between the adjacent blocks.

To improve the quality of the reconstructed image, the Huffman code table needs to be strengthen such that it can be corrected if some errors occur. One effective way is to add the parity check bits to each row and column of the code table. The code table's size is 11×16 and each table item is represented by maximum 16 bits. There are 17 sixteen-bit words are used for parity check, which 11 words are for row parity, and 16 words are for column parity. The organization of parity bits are shown in Figure 3. Assume the code table contents are located inside the 11×16 array. The 11st row is used to store the parity check bits for the codes in every column. Similar, the 16th column is used to store the parity check bits for the codes in every row. Before coding process, the 16 parity check bits for each column are formed by XOR all the code words in

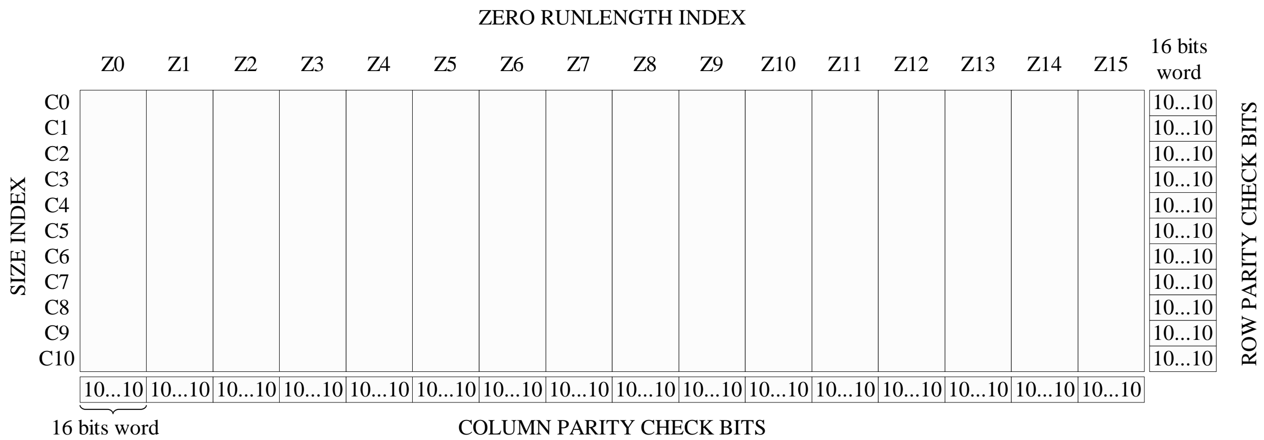

Figure 3: Row and column parity check bit design for the AC Huffman code table

that column. Similar, the row parity check bits are formed by XOR all the code words of the same row. To check for the correctness of a table item, the parity check bits for the row and column whose intersection is located at the current table item. These reconstructed parity bits are then compared with the correspond precalculated parity row and column parity bits. If both row and column parity are not matched, the error was occurred in the current code item. This error checking method only works if no more than one error bit occurs at the same bit position in a table row or column. For instance, the code item at row 1 column 2 is being considered. Assume 4th bit is flipped from 1 to 0, and no 4th bits occur to other items in row 1 and column 2. If that condition is violated, false or miss detection may occur depend on particular situation.

3.2 Single bit Error Correction Design for Huffman Code Table

The error detection scheme as shown in the previous section is the most simple way to detect errors occur inside the 2-dimension code table. Since the detection always followed by error handle processes. If the fault tolerance design only stop at the fault detection, the error handle must either repeat the work or skip the rest coefficients of that 8×8 data block, and encode for the next one. In either case, the time overhead is possibly high and the quality of the entire image will be lower due to the errors still remain in the code table.

To overcome this problem, error correction using biresidue codes can be applied to correct all the single bit errors inside the Huffman code table.

Biresidue codes This is the separate code with two or more residue checks. Let A_1 and A_2 be the check bases; and the code needs to be checked is $N \in Z_m$. Given $m = 2^k - 1$, A_i can be chosen such that $A_1 = 2^c - 1, A_2 = 2^d - 1$ for positive integers c and d such that c divides k and d also divides k. For each base $A_i \;\; \{i = 1, 2\}$, the low-cost residue code for each codeword N is created. For the symbol-1 Huffman code table as shown in Table 3, the maximum length of each codeword is 16 bits long. Therefore, if the check bases are selected to be $A_1 = 3$ and $A_2 = 511$, then the low-cost residue codes have lengths $c = 2$ and $d = 9$. In this case, the data occupy 18 bits long. To see how the biresidue codes are formed, consider an 18-bit code $N = 110111010011$. Its check codes $|N|_3$ and $|N|_5 11$ are to be generated. For the check code $|N|_3$, N is divided into bytes of length 2 starting from right, and these bytes are added modulo 3 (i.e., with an end-around-carry as in 1's complement number).

Let $N = 00|00|00|11|01|11|01|00|11 = 3539$. The 2-bit bytes are added as follows.

$$\begin{array}{r} 11 \\ 00 \\ \hline 11 \end{array} \quad \begin{array}{r} 11 \\ 01 \\ \hline 100 \\ 1 \\ \hline 01 \end{array} \quad \begin{array}{r} 01 \\ 11 \\ \hline 100 \\ 1 \\ \hline 01 \end{array} \quad \begin{array}{r} 01 \\ 01 \\ \hline 10 \end{array} \quad \begin{array}{r} 10 \\ 11 \\ \hline 101 \\ 1 \\ \hline 10 \end{array} \qquad C_1 = |N|_3 = 10_2 = 2$$

Figure 4: The residue code computation for a 18-bit N modulo 3

Similar, the 9-bit bytes are added as follows

$$\begin{array}{r} 000000110 \\ 111010011 \\ \hline 111011001 \end{array} \qquad C_2 = |N|_{511} = 111011001_2 = 473$$

Figure 5: The residue code computation for a 18-bit N modulo 511

The single bit error correction is done by computation the syndromes of a 3-tuple (N, C_1, C_2) is given by

$$\mathbf{s}(N, C_1, C_2) = (s_1, s_2) = (|N - C_1|_3, |N - C_2|_{511}) \tag{2}$$

It is assumed here that the three components of the code are processed is separate units, called the data, checker 1 and checker 2, respectively, and therefore at most one of the components of the result will be erroneous at any given time. A triple (N, C_1, C_2) is a codeword if and only if $\mathbf{s}(N, C_1, C_2) = (0, 0)$. For a codeword (N, C_1, C_2), consider the three separate cases of errors:

1. Error in the data unit. Let N', C_1, C_2 denote the erroneous word due to an error in the data, such that $N' = |N + e|_{2^{18}-1}$. Then the syndrome

$$\mathbf{s}(N', C_1, C_2) = \mathbf{s}(e, 0, 0) = \big(|N' - C_1|_3, |N' - C_2|_{511}\big) = (|e|_3, |e|_{511}) \tag{3}$$

2. Error checker 1. The erroneous word is of the form (N, C_1', C_2) where $C_1' = |C_1 + e|_3$. Its syndrome

$$\mathbf{s}(N, C_1', C_2) = \big(|N - C_1'|_3, |N - C_2|_{511}\big) = (|-e|_3, 0) \tag{4}$$

3. Error in checker 2. The erroneous word is of the form (N, C_1, C_2') where $C_2' = |C_2 + e|_{511}$. Its syndrome

$$\mathbf{s}(N, C_1, C_2') = \big(|N - C_1|_3, |N - C_2'|_{511}\big) = (0, |-e|_{511}) \tag{5}$$

From the above three cases, the syndrome **s**=(s_1, s_2) indicates the erroneous status

$s_1 = 0, s2 = 0$: no error; $s_1 \neq 0, s2 \neq 0$: error in the data

$s_1 \neq 0, s2 = 0$: error in checker 1; $s_1 = 0, s2 \neq 0$: error in checker 2

For the cases when there is an error in one of the checkers, it can be easily corrected by computing the check value from N. When the error is in the data unit, the syndromes corresponding to the set of single errors must be distinct as shown in the table1. Evidently, the syndromes are all clearly distinct for $i = 0, 1, \ldots, 17$. However for $i = 18$, the syndrome $s(2^{18}, 0, 0) = (1, 1) = s(2^0, 0, 0)$. Thus the code can correct single errors in the data unit for $k = 18$ More information about biresidue code can be found in [1].

Table 1: Syndromes of single errors for the biresidue code with $k = 18$, $c = 2$ and $d = 9$

i	Syndrome $\mathbf{s}(2^i, 0, 0)$	$\mathbf{s}(m_0 - 2^i, 0, 0)$	i	Syndrome $\mathbf{s}(2^i, 0, 0)$	$\mathbf{s}(m_0 - 2^i, 0, 0)$
0	(1,1)	(2,510)	9	(2,1)	(1,510)
1	(2,2)	(1,509)	10	(1,2)	(2,509)
2	(1,4)	(2,507)	11	(2,4)	(1,507)
3	(2,8)	(1,503)	12	(1,8)	(2,503)
4	(1,16)	(2,495)	13	(2,16)	(1,495)
5	(2,32)	(1,479)	14	(1,32)	(2,479)
6	(1,64)	(2,447)	15	(2,64)	(1,447)
7	(2,128)	(1,383)	16	(1,128)	(2,383)
8	(1,256)	(2,255)	17	(2,256)	(1,255)

3.3 Experiment results

The proposed single error detection and correction is implemented in software using visual C programming. First, the raw data image is transformed by the 2-dimensional discrete cosine transform and quantized by a default luminance quantization table. The data then pass through the Huffman coding process, where the error is randomly injected into the quantization table. In the first process, the table is checked, but not corrected. If an error occurs when coding a certain block symbol,the block is forced to terminate there, and the process continues to code the next block. In the second process, single error correction (SEC) function is deployed to correct the single bit errors occurred to a codeword before that codeword is inserted into the codestream. The decoder is performed in reverse order to reconstructed the input image. The mean square error for the reconstructed images is computed and the images are displayed for observation.

Table 2: Comparison of Mean Square Errors when (SEC) inactivate and activated

image	without SEC	with SEC
egret	50	37
man	229	180
flamingo	160	143
hawk	386	324
heron	285	233
winter	25	21

3.4 Conclusions

In this report, the JPEG Huffman entropy coding has been analyzed, implemented and tested successfully. The redundancy single error correction/double error detection codes are also implemented. The computer simulation takes the color input images, performs discrete cosine transform and scalar quantization steps before enters to the Huffman coding. The single and double-bit errors are randomly occurred inside the Symbol-1 Huffman code table. Symbol-2 Figure 4 can be computed directly without using the table. During the coding process, all the single-bit errors are removed by using the biresidue codes. In most cases, double-bit errors in the table are detected. For the double-bit errors, the rest coefficients in the current block are skipped over and replaced by an end-of block code. The reconstructed image for the cases of single and double-bit errors are showed in the Figures 6. The performance is improved in terms of both mean square error and image perception.

References

[1] Rao T.R.N; E. Fujiwara. *Error control coding for computer systems*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[2] W.B. Pennebaker and J.L. Mitchell. *JPEG Still Image Data Compression Standard*. International Thomson Publishing, New York, NY, 1993.

Figure 6: (*left*): original image;(*middle*): the reconstructed image without SEC for the code table; (*right*): reconstructed image with SEC/DED for the code table

Table 3: Baseline Huffman Coding Symbol-2 Structure.

Size	Amplitude									DCCodeWord
0	0									0
1	-1	1								10
2	-3	-2	2	3						110
3	-7	-6	-5	-4	4	5	6	7		1110
4	-15	-14	-13	-12	...	12	13	14	15	11110
5	-31	-30	-29	-28	...	28	29	30	31	111110
6	-63	-62	-61	-60	...	60	61	62	63	1111110
7	-127	-126	-125	-124	...	124	125	126	127	11111110
8	-255	-254	-253	-252	...	252	253	254	255	111111110
9	-511	-510	-509	-508	...	508	509	510	511	1111111110
10	-1023	-1022	-1021	-1020	...	1020	1021	1022	1023	11111111110
11	-2047	-2046	-2045	-2044	...	2044	2045	2046	2047	111111111110

Table 4: Baseline Huffman Coding Symbol-1 Structure.

	Runlength			
Size	Z0	Z1	Z2	Z3
C1	00	1100	11100	111010
C2	01	11011	11111001	111110111
C3	100	1111001	1111110111	111111110101
C4	1011	111110110	111111110100	111111110001111
C5	11010	11111110110	111111110001001	111111110010000
C6	1111000	111111110000100	111111110001010	111111110010001
C7	11111000	111111110000101	111111110001011	111111110010010
C8	1111110110	111111110000110	111111110001100	111111110010011
C9	111111110000010	111111110000111	111111110001101	111111110010100
C9	111111110000010	111111110000111	111111110001101	111111110010100
Size	Z4	Z5	Z6	Z7
C1	111011	1111010	1111011	11111010
C2	1111111000	11111110111	111111110110	111111110111
C3	111111110010110	111111110011110	111111110100110	111111110101110
C4	111111110010111	111111110011111	111111110100111	111111110101111
C5	111111110011000	111111110100000	111111110101000	111111110110000
C6	111111110011001	111111110100001	111111110101001	111111110110001
C7	111111110011010	111111110100010	111111110101010	111111110110010
C8	111111110011011	111111110100011	111111110101011	111111110110011
C9	111111110011100	111111110100100	111111110101100	111111110110100
C10	111111110011101	111111110100101	111111110101101	111111110110101
Size	Z8	Z9	Z10	Z11
C1	111111000	111111001	111111010	1111111001
C2	111111111000000	111111110111110	111111111000111	111111111010000
C3	111111110110110	111111110111111	111111111001000	111111111010001
C4	111111110110111	111111111000000	111111111001001	111111111010010
C5	111111110111000	111111111000001	111111111001010	111111111010011
C6	111111110111001	111111111000010	111111111001011	111111111010100
C7	111111110111010	111111111000011	111111111001100	111111111010101
C8	111111110111011	111111111000100	111111111001101	111111111010110
C9	111111110111100	111111111000101	111111111001110	111111111010111
C10	111111110111101	111111111000110	111111111001111	111111110111101
Size	Z12	Z13	Z14	Z15
C1	1111111010	11111111000	111111111101011	111111111110101
C2	111111111011001	111111111100010	111111111101100	111111111110110
C3	111111111011010	111111111100011	111111111101101	111111111110111
C4	111111111011011	111111111100100	111111111101110	111111111111000
C5	111111111011100	111111111100101	111111111101111	111111111111001
C6	111111111011101	111111111100110	111111111110000	111111111111010
C7	111111111011110	111111111100111	111111111110001	111111111111011
C8	111111111011111	111111111101000	111111111110010	111111111111100
C9	111111111100000	111111111101001	111111111110011	111111111111101
C10	111111111100001	111111111101010	111111111110100	111111111111110