Image Compression using DCT Algorithm on TMS320C6748

Department of Electronics and Communication Engineering
National Institute of Technology, Warangal
2023-2024

DSP LAB PROJECT

Submitted by:

KDK Thanmay	21ECB0B24
M Lakshmi Chaitanya	21ECB0B29
MVS Srimanth	21ECB0B33

Faculty:

Dr. Ravi kumar Jatoth
Professor
Department of Electronics and Communication Engineering

CONTENTS

ABSTRACT

Image compression is a crucial aspect of modern multimedia applications, facilitating the efficient storage and transmission of visual information. This mini project explores a comprehensive approach to image compression, combining the power of Discrete Cosine Transform (DCT) and Huffman Coding. The proposed methodology aims to achieve a balance between compression ratio and image quality, making it suitable for various applications, including multimedia storage, transmission over networks, and real-time processing. The project begins with an in-depth exploration of the Discrete Cosine Transform, a widely employed technique in image compression. DCT transforms spatial information into frequency components, providing a compact representation of the image. This transformation is pivotal for reducing redundancy and exploiting the inherent visual characteristics of the image. The project discusses the mathematical foundation of DCT and its application to transform image data efficiently.

Following the DCT phase, the project integrates Huffman Coding, a renowned entropy coding technique, to further enhance compression efficiency. Huffman Coding assigns shorter codes to frequently occurring symbols, enabling more efficient representation of the transformed image data. The selection of appropriate coding strategies significantly influences the compression ratio and decoding complexity. The project addresses the design considerations and optimization techniques for implementing Huffman Coding in conjunction with DCT.

In conclusion, the integration of Discrete Cosine Transform and Huffman Coding offers a robust solution for image compression. The project presents a thorough examination of the underlying principles, the design and implementation of the compression algorithm, and an extensive evaluation of its performance. The results demonstrate the potential for achieving a balance between compression efficiency and image fidelity, making the proposed approach a valuable contribution to the field of image compression.

Keywords –Discrete Cosine Transform , Huffman Coding, Compression Ratio, Redundancy, Image Fidelity.

INTRODUCTION

A digital image obtained by sampling and quantizing a continuous tone picture requires an enormous storage. For instance, a 24 bit color image with 512x512 pixels will occupy 768 Kbyte storage on a disk, and a picture twice of this size will not fit in a single floppy disk. To transmit such an image over a 28.8 Kbps modem would take almost 4 minutes. There are different techniques for compressing images. They are broadly classified into two classes called lossless and lossy compression techniques

It's well known that Huffman's algorithm is generating minimum redundancy codes compared to other algorithms. The Huffman coding has effectively been used in text, image, video compression, and conferencing systems such as, JPEG, MPEG-2, MPEG-4, and H.263 etc.. The Huffman coding technique collects unique symbols from the source image and calculates its probability value for each symbol and sorts the symbols based on its probability value.

The JPEG process is a widely used form of Lossy image compression that centers around the Discrete Cosine Transform (DCT). The DCT works by separating images into parts with differing frequencies. During a step called Quantization. Where part of compression actually occurs, the less important frequencies are discarded, hence the use of the term "Lossy". Then, only the most important frequencies that remain are used to retrieve the image in the decompression process. As a result, reconstructed images contain some distortion; but as we shall soon see, this level of distortion can be adjusted during the Compression stage. The JPEG method is used for both color and black-and-white images.

Compression can be divided into two categories, as Lossless and Lossy compression. In lossless compression, the reconstructed image after compression is numerically identical to the original image. In lossy compression scheme, the reconstructed image contains degradation relative to the original. Lossy technique causes image quality degradation in each compression or decompression step. In general, lossy techniques provide for greater compression ratios than lossless techniques. The following are the some of the lossless and lossy data compression techniques :

1. **Lossless Coding Techniques**
   a. Run length encoding
   b. Huffman encoding
   c. Arithmetic encoding
   d. Entropy coding
   e. Area Coding

2. **Lossy Coding Techniques**
   a. Predictive Coding
   b. Huffman encoding
   c. Arithmetic encoding
   d. Entropy coding
   e. Area coding

Merits

- Huffman Coding is a Variable length Coding Technique, which efficiently represents a lesser number of bits.
- Huffman Coding is a Lossless compression Technique, ensures exact reconstruction of data.
- DCT concentrates signal energy in low frequency coefficients, allowing for a compact representation of image
- DCT decorrelates image by transforming spatial information into frequency components, reducing redundancy and capturing essential visual information.

Demerits

- Adapting to changing symbol probabilities in real-time applications can introduce complexity, impacting the adaptability of Huffman Coding.
- DCT-based compression, particularly in its most widely used form in JPEG, is inherently lossy, leading to some loss of image quality.

METHODOLOGY

A. Huffman Coding

Huffman code procedure is based on the two observations

a. More frequently occurring symbols will have shorter code words than symbols that occur less frequently.

b. The two symbols that occur least frequently will have the same length.

Original source		Source reduction			
S	P	1	2	3	4
a2	0.4	0.4	0.4	0.4	0.6
a6	0.3	0.3	0.3	0.3	0.4
a1	0.1	0.1	0.2	0.3	
a4	0.1	0.1	0.1		
a3	0.06	0.1			
a5	0.04				

S-source, P-probability

Table 1: Huffman source reduction.

Original source		Source reduction			
S	P	1	2	3	4
a2	0.4[1]	0.4[1]	0.4[1]	0.4[1]	0.6[0]
a6	0.3[00]	0.3[00]	0.3[00]	0.3[00]	0.4[1]
a1	0.1[011]	0.1[011]	0.2[010]	0.3[01]	
a4	0.1[0100]	0.1[0100]	0.1[011]		
a3	0.06[01010]	0.1[0101]			
a5	0.04[01011]				

. Table 2 : Huffman Code Assignment Procedure

At the far left of the table I the symbols are listed and corresponding symbol probabilities are arranged in decreasing order and now the least two probabilities are merged as here 0.06 and 0.04 are merged, this gives a compound symbol with

probability 0.1, and the compound symbol probability is placed in source reduction column1 such that again the probabilities should be in decreasing order. so this process is continued until only two probabilities are left at the far right shown in the above table as 0.6 and 0.4. The second step in Huffman's procedure is to code each reduced source, starting with the smallest source and working back to its original source. The minimal length binary code for a two-symbol source, of course, is the symbols 0 and 1. As shown in table II these symbols are assigned to the two symbols on the right (the assignment is arbitrary; reversing the order of the 0 and would work just and well). As the reduced source symbol with probabilities 0.6 was generated by combining two symbols in the reduced source to, the 0 used to code it is now assigned to both of these symbols, and a 0 and 1 are arbitrarily appended to each to distinguish them from each other. This operation is then repeated for each reduced source until the original course is reached. The final code appears at the far-left in table 1.8. The average length of the code is given by the average of the product of probability of the symbol and number of bits used to encode it. This is calculated below:

Lavg = (0.4)(1) +(0.3)(2) + (0.1)(3) + (0.1)(4) + (0.06)(5) + (0.04)(5) = 2.2 bits/ symbol

and the entropy of the source is 2.14bits/symbol, the resulting Huffman code efficiency is2.14/2.2 = 0.973.

Entropy, $H=-\sum P(aj)\log P(aj)$

Huffman's procedure creates the optimal code for a set of symbols and probabilities subject to the constraint that the symbols be coded one at a time.

B. Discrete Cosine Transform Compression / Decompression

Image compression is a technique used to reduce the storage and transmission costs. The existing techniques used for compressing image files are broadly classified into two categories, namely lossless and lossy compression techniques. In lossy compression techniques, the original digital image is usually transformed through an invertible linear transform into another domain, where it is highly de-correlated by the transform. This de-correlation concentrates the important image information into a more compact form. The transformed coefficients are then quantized yielding bitstreams containing long stretches of zeros. Such bitstreams can be coded efficiently to remove the redundancy and store it into a compressed file. The decompression reverses this process to produce the recovered image.

For a N*N image whose pixel values are represented by the matrix I(r, c) where r represents the row-wise pixel and c represent column-wise pixel, the 2-D DCT Equation is given by,

$$D(u,v) = \alpha(u)\alpha(v)\sum_{r=0}^{N-1}\sum_{c=0}^{N-1} I(r,c)\cos\left[\frac{(2r+1)u\pi}{2N}\right]\cos\left[\frac{(2c+1)v\pi}{2N}\right]$$

$$\alpha(u),\alpha(v) = \begin{cases} \sqrt{\frac{1}{N}} & \text{if } u=0 \\ \sqrt{\frac{2}{N}} & \text{if } u>0 \end{cases}$$

Where u=0,1,…N-1 along rows and v=0,1,…N-1 along columns

But as the value of N increases, the computational complexity of the DCT operation increases. Hence to reduce the complexity, DCT is performed in terms of predefined coefficient Blocks. The Equation of DCT in Matrix form is given by,

$$D(u,v) = \begin{cases} \sqrt{\frac{1}{N}} & \text{if } u = 0 \\ \sqrt{\frac{2}{N}} \cos\left(\frac{(2v+1)u\pi}{2N}\right) & \text{if } u > 0 \end{cases}$$

The most commonly used one is 8*8 DCT Coefficient Matrix. It is given by,

0.3536	0.3536	0.3536	0.3536	0.3536	0.3536	0.3536	0.3536
0.4904	0.4157	0.2778	0.0975	-0.0975	-0.2778	-0.4157	-0.4904
0.4619	0.1913	-0.1913	-0.4619	-0.4619	-0.1913	0.1913	0.4619
0.4157	-0.0975	-0.4904	-0.2778	0.2778	0.4904	0.0975	-0.4157
0.3536	-0.3536	-0.3536	0.3536	0.3536	-0.3536	-0.3536	0.3536
0.2778	-0.4904	0.0975	0.4157	-0.4157	-0.0975	0.4904	-0.2778
0.1913	-0.4619	0.4619	-0.1913	-0.1913	0.4619	-0.4619	0.1913
0.0975	-0.2778	0.4157	-0.4904	0.4904	-0.4157	0.2778	-0.0975

Figure. 8*8 DCT Coefficient Matrix.

The DCT is designed to work on pixel values ranging from values -0.5 to 0.5 for binary image and -127 to 128 for grayscale images. Hence Thresholding or Level-off operation is performed before applying DCT operation i.e. 0.5 or 128 is subtracted from the image pixel values.

The Final DCT of an 8*8 Image I, where D is 8*8 DCT Coefficient Matrix, is given by,

$$\mathbf{C = D\ I\ D^T}$$

Similarly, The Inverse DCT is obtained by,

$$\mathbf{I = D^T C D}$$

After Applying the DCT operation, Quantization operation is performed to use only high energy coefficients. For that there are predefined Matrices based on the required quality of the image ranging from Q1 to Q100. Q1 accounts to lowest quality and Q100 accounts to highest quality. Generally for the overall balance of visual experience and compression Q50 is used. It is give by,

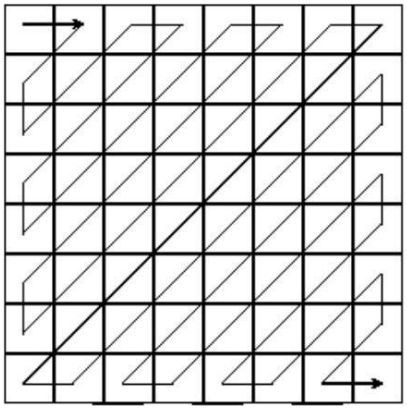
16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Figure. Q50 Quantization Matrix.

In the quantization process after the DCT calculation, the following formula is applied:

$$\mathbf{C'(i,j) = Round(C(i,j)/Q(i,j))}$$

After the Quantization process, the Coefficients in the Matrix are encoded by taking the following Zig-Zag Pattern,



0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

After all the above steps, Huffman Coding is done to the obtained DCT Coefficients. The Block Diagram of steps used in JPEG standard is,

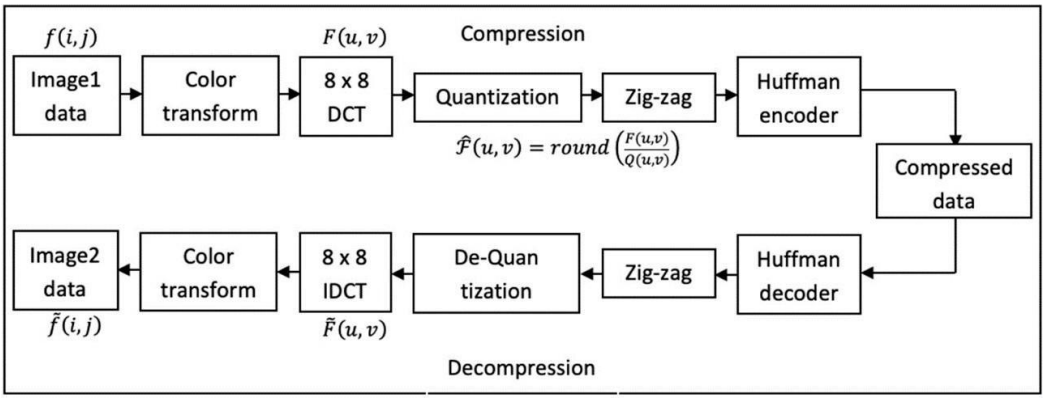


Figure . Block Diagram of Process Steps in JPEG Standard

128	75	72	105	149	169	127	100
122	84	83	84	146	138	142	139
118	98	89	94	136	96	143	188
122	106	79	115	148	102	127	167
127	115	106	94	155	124	103	155
125	115	130	140	170	174	115	136
127	110	122	163	175	140	119	87
146	114	127	140	131	142	153	93

Let us Consider a 8*8 Block Image, whose pixel values are given by :

Figure . Pixel Values of a 8*8 Sample Image

After thresholding operation, the pixel values looks like,

0	-53	-56	-23	21	41	-1	-28
-6	-44	-45	-44	18	10	14	11
-10	-30	-39	-34	8	-32	15	60
-6	-22	-49	-13	20	-26	-1	39
-1	-13	-22	-34	27	-4	-25	27
-3	-13	2	12	42	46	-13	8
-1	-18	-6	35	47	12	-9	-41
18	-14	-1	12	3	14	25	-35

Figure . Pixel Values of the Sample Image after Thresholding

After Applying DCT Operation to the above Image, the resultant pixel values are:

-28.5000	-76.5184	-13.7954	74.9966	60.7500	-18.1278	3.2788	26.0613
-50.9050	-61.4830	54.5059	17.3142	17.8372	-18.3422	0.9247	10.7207
-1.5715	19.4880	-43.0986	52.6368	-36.8119	53.4907	3.4335	-7.9910
11.9062	11.8799	-52.5027	37.4529	-7.9181	-19.5318	32.7068	-15.9874
-16.5000	10.9450	13.9867	12.6455	-2.2500	5.9000	-2.8169	-6.1124
-0.5212	6.0888	-28.0692	-0.9305	19.9330	9.2643	-17.2415	-11.0720
15.8044	-11.1664	13.1835	-8.8973	-7.2116	2.2942	11.3486	-7.6886
-12.6256	7.3221	7.5976	-9.2059	5.1053	-7.7872	-1.4507	10.2659

Figure . DCT of the Sample Image

The pixels values of the image after Quantization operation using Q50 Matrix is,

-2	-7	-1	5	3	0	0	0
-4	-5	4	1	1	0	0	0
0	1	-3	2	-1	1	0	0
1	1	-2	1	0	0	0	0
-1	0	0	0	0	0	0	0
0	0	-1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure . Quantized Matrix of the Sample Image

The Huffman Codes and Redundancy of the Image size if given by,

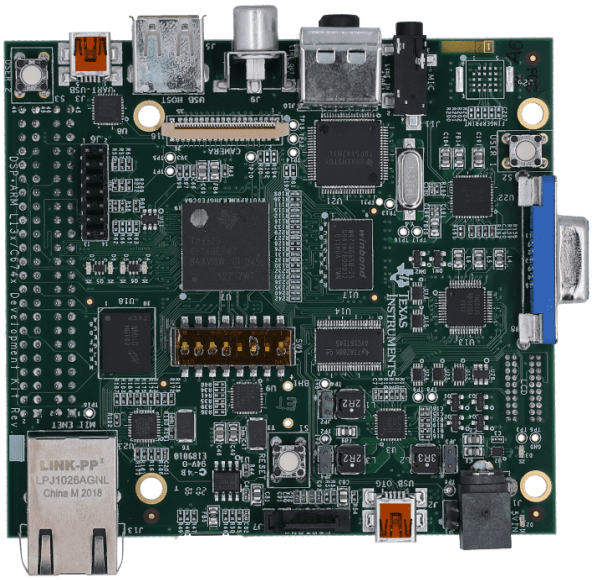
```
Huffmann Codes:

pixel values ->   Code

    -5      -> 10010
    -4      -> 10011
    -3      -> 10000
    -2      -> 110
    -1      -> 111
    0      -> 01
    1      -> 00
    2      -> 10001
    3      -> 10110
    4      -> 10111
    5      -> 10100
    7      -> 10101
Before Huffman Coding Total No. of  Bits = 512
After  Huffman Coding Total No. of  Bits = 158
Redundancy - 69.140625 %
```

Figure . Huffman Codes and Redundancy of the Sample Image

C. Deployment

TMS320C6748 DSP Processor:

a. The TMS320C6748 is a high-performance, fixed-point DSP processor designed by Texas Instruments.
a. It features a C67x DSP core with floating-point and fixed-point capabilities, making it suitable for a wide range of signal processing applications.
b. The processor includes various peripherals such as GPIO, UART, I2C, SPI, and more, providing flexibility for interfacing with external components.



TMS320C6748 Board

Development Environment:

c. Code Composer Studio (CCS): The integrated development environment for TMS320C6748, facilitating code development, debugging, and system analysis.
d. TMS320C6748 Evaluation Board: This board serves as the hardware platform for testing and validating the implemented DCT algorithm.

a. Processor Features:

- C67x DSP Core: Provides high computational power for signal processing tasks.
- Memory: Includes on-chip memory such as L1 and L2 caches for efficient data access.
- Peripherals: Features various peripherals for versatile connectivity and interfacing.

b. Connectivity:

- Ethernet Port: Enables network connectivity for data transfer.
- USB Ports: Facilitate data exchange with external devices.
- Audio In/Out: Support for audio processing applications.
- GPIO and Other Interfaces: Provide general-purpose digital and analog interfacing.

c. Power and Clocking:

- Low Power Consumption: Suitable for applications with power constraints.
- Flexible Clocking Options: Adaptable to different processing speed requirements.

CODE

```
#include <stdio.h>
#include<math.h>
#include<string.h>
#include"input_image.h"
#include"huff.h"
#define pi 3.142857
const int N = 256;
float dctc[8][8];
float dctct[8][8];
// Function to find discrete cosine transform and print it
void dctTransform()
{
    int i, j;
    for (i = 0; i < 8; i++) {
        for (j = 0; j < 8; j++) {
            if(i == 0){
                dctc[i][j]=1/sqrt(8);
            }
            else{
                dctc[i][j]=(sqrt(2)/sqrt(8))*cos((2*j+1)*i*pi/(2*8));
            }
                dctct[j][i]=dctc[i][j];
        }
    }
}



int q[8][8]={{16,11,10,16,24,40,51,61},{12,12,14,19,26,58,60,55},
```

```
{14,13,16,24,40,57,69,56},
{14,17,22,29,51,87,80,62},
{18,22,37,56,68,109,103,77},
{24,35,55,64,81,104,113,92},
{149,64,78,87,103,121,120,101},{72,92,95,98,112,100,103,99}};
int im[256][256],im2[256][256];
int main()
{
   dctTransform();
   int i,j,k;
   float dum[8][8],dum2[8][8];
   int dct[8][8],dct2[8][8];
   printf("Discrete Cosine Transform Coefficient Matrix\n");
   for(i=0;i<8;i++){
      for(j=0;j<8;j++){
         printf("%f ",dctc[i][j]);
      }
      printf("\n");
   }
   printf("Quantization Q50 Matrix\n");
      for(i=0;i<8;i++){
         for(j=0;j<8;j++){
            printf("%d ",q[i][j]);
         }
         printf("\n");
      }

   for(i=0;i<N;i++){
      for(j=0;j<N;j++){
         image[i][j]-=128;
```

```
    }
}
int r=0,c=0;
for(r=0;r<N;r=r+8){
    for(c=0;c<N;c=c+8){
        for(i=0;i<8;i++){
            for(j=0;j<8;j++){
                dum[i][j]=0;
                for(k=0;k<8;k++){
                    dum[i][j] += dctc[i][k]*image[r+k][c+j];
                }
            }
        }
        for(i=0;i<8;i++){
            for(j=0;j<8;j++){
                dct[i][j]=0;
                for(k=0;k<8;k++){
                    dct[i][j] += dum[i][k]*dctct[k][j];
                }
                dct[i][j] = round((double)dct[i][j]/(double)q[i][j]);
                im[r+i][c+j]=dct[i][j];
            }
        }
    }
}
for(r=0;r<N;r=r+8){
    for(c=0;c<N;c=c+8){
        for(i=0;i<8;i++){
            for(j=0;j<8;j++){
                im[r+i][c+j]=round((double)im[r+i][c+j]*(double)q[i][j]);
```

```
            }
        }
    }
  }
for(r=0;r<N;r=r+8){
    for(c=0;c<N;c=c+8){

        for(i=0;i<8;i++){
            for(j=0;j<8;j++){
                dum2[i][j]=0;
                for(k=0;k<8;k++){
                    dum2[i][j] += dctct[i][k]*im[r+k][c+j];
                }
            }
        }
        for(i=0;i<8;i++){
            for(j=0;j<8;j++){
                dct2[i][j]=0;
                for(k=0;k<8;k++){
                    dct2[i][j] += dum2[i][k]*dctc[k][j];
                }
                im2[r+i][c+j]=dct2[i][j]+128;
            }
        }
    }
  }
  for(i=0;i<N;i++){
        for(j=0;j<N;j++){
            image[i][j]+=128;
        }
```

```
    }
maxval=maxval+256;
int hist[maxval];
   for(i=0;i<maxval;i++){
                hist[i]=0;
                freq[i]=0;
        }
        for( i=0;i<256;i++){
                for( j=0;j<256;j++){
                        hist[255+im[i][j]]++;
                }
        }
   if(65536%min_freq==0){
        min_prob=65536/min_freq;
        }
        else{
        min_prob=65536/min_freq+1;
        } fib_func();
        maxcodelen=maxlen(min_prob);

struct pixfreq {
   int pix;
   float freq;
   struct pixfreq *left, *right;
   char code[maxcodelen];
        };
        struct huffcode {
   int pix, arrloc;
   float freq;
   };
```

```
 struct pixfreq* pix_freq;
 struct huffcode* huffcodes;
 int totalnodes = 2 * nodes -
 1;
pix_freq = (struct pixfreq*)malloc(sizeof(struct pixfreq) * totalnodes);
huffcodes = (struct huffcode*)malloc(sizeof(struct huffcode) * nodes);
j = 0;
int totpix = 65536;
while (n < 2*nodes - 1)
{
   sumprob = huffcodes[nodes - n - 1].freq + huffcodes[nodes - n - 2].freq;
   sumpix = huffcodes[nodes - n - 1].pix + huffcodes[nodes - n - 2].pix;
   pix_freq[nextnode].pix = sumpix;
   pix_freq[nextnode].freq = sumprob;
   pix_freq[nextnode].left = &pix_freq[huffcodes[nodes - n - 2].arrloc];
   pix_freq[nextnode].right = &pix_freq[huffcodes[nodes - n - 1].arrloc];
   pix_freq[nextnode].code[0] = '$';
   i = 0;
   while (sumprob <= huffcodes[i].freq){
      i++;
   for (k = nodes-1; k >= 0; k--)
   {
      if (k == i)
      {
         huffcodes[k].pix = sumpix;
         huffcodes[k].freq = sumprob;
         huffcodes[k].arrloc = nextnode;
      }
      else if (k > i)
         huffcodes[k] = huffcodes[k - 1];
   }
```

```
    n += 1;
    nextnode += 1;
}
// Assigning Code through backtracking
char left = '0';
char right = '1';
int index;
for (i = totalnodes - 1; i >= nodes; i--) {
        if (pix_freq[i].left != NULL) {
                strconcat(pix_freq[i].left->code, pix_freq[i].code, left, maxcodelen);
        }
        if (pix_freq[i].right != NULL) {
                strconcat(pix_freq[i].right->code, pix_freq[i].code, right, maxcodelen);
        }
}
int len;
for (i = 0; i < nodes; i++) {
    len = strlen(pix_freq[i].code);
    for(j=0;j<len;j++){
        if(pix_freq[i].code[j]=='$'){
                pix_freq[i].code[j]='\0';
                }
        }
}
    printf("Pixel Value
    Probability\n"); for(i=0;i<308;i++){
       printf("%d          %f\n",pixval[i],freqval[i]);
    }
    printf("Pixel Value          Huffman
       Code\n"); for(i=0;i<308;i++){
```

```
        printf("%d          %s\n",pixval[i],hc[i]);
      }
    printf("Pixel Values after Thresholding, Discrete Cosine Transform and Quantization\n");
        for(i=0;i<N;i++){
            for(j=0;j<N;j++){
              printf("%d ", im[i][j]);
            }
            printf("\n");
        }
      printf("Pixel Values after Decompression\n");
        for(i=0;i<N;i++){
             for(j=0;j<N;j++){
               printf("%d ", im2[i][j]);
             }
             printf("\n");
         }
        int prevsize=65536*8;
        for(i=0;i<308;i++){
           currsize += strlen(hc[i])*65536*freqval[i];
        }
        printf("Before Huffman Coding Total No. of Bits = %d
        \n",prevsize); printf("After Huffman Coding Total No. of Bits = %d
                                                                \n",currsize);
       float red = (1-((float)currsize/(float)prevsize))*100;
        printf("Redundancy - %f %% \n",redu);

   return 0;
}
```

RESULTS

Input Image :

Figure. Input Image in Code Composer Studio

Output Decompressed Image :

Figure. Output Image in Matlab

Figure. Output Image in CCS

Huffman Codes of Compressed Image Pixel Values :

```
Huffmann Codes:

pixel values ->   Code

    -253      -> 11000000110110
    -252      -> 111110000010
    -250      -> 111110000000
    -247      -> 111110011011
    -242      -> 100101010001
    -240      -> 110001011010
    -238      -> 11000000110111
    -232      -> 10011001100101
    -231      -> 11000000110100
    -230      -> 1100000010110
    -228      -> 1011100101011
    -224      -> 10010101111
    -221      -> 11000000110101
    -220      -> 100101010010
    -216      -> 111110011100
    -210      -> 110000011111
    -209      -> 110000010010
    -208      -> 100101010111
    -204      -> 1000010101000
    -200      -> 110001011001
    -198      -> 110000011110
    -196      -> 1100000011000
    -195      -> 11000000001010
    -192      -> 1101011001
    -190      -> 1100000010100
    -187      -> 111110011000
    -182      -> 100101010000
    -180      -> 11000101111
    -176      -> 10011001110
    -174      -> 11000000001011
    -171      -> 1011100101111
    -170      -> 1011100100010
    -168      -> 1101101110
    -165      -> 100101010011
    -162      -> 11000000001000
    -160      -> 1100001001
```

```
    -156      -> 1111100100
    -154      -> 11111101000
    -153      -> 11000000001001
    -152      -> 1100000010011
    -150      -> 111110011111
    -149      -> 110000011101
    -145      -> 11000000001110
    -144      -> 111110111
    -143      -> 110000010111
    -140      -> 10111001100
    -138      -> 11000000001111
    -136      -> 11000000001100
    -133      -> 111110000011
    -132      -> 1000001111
    -130      -> 10000101001
    -128      -> 1100010101
    -126      -> 1100001010
    -124      -> 11000000001101
    -122      -> 110000011100
    -121      -> 111110000110
    -120      -> 110001100
    -119      -> 1011100101100
    -117      -> 110000010011
    -116      -> 11101011010
    -114      -> 110000010110
    -113      -> 11000000000010
    -112      -> 111010101
    -110      -> 1110101111
    -109      -> 110000010100
    -108      -> 1110101110
    -104      -> 1101011011
    -102      -> 11010111011
    -100      -> 100001010110
    -99       -> 11111000111
    -98       -> 1100001000
    -96       -> 11101001
    -95       -> 111110011110
    -91       -> 110001011011
    -90       -> 11010111000
    -88       -> 1101011110
```

```
    -87       -> 10010101100
    -85       -> 110001011000
    -84       -> 101001011
    -81       -> 11111000101
    -80       -> 10110100
    -78       -> 1001100101
    -77       -> 11010111010
    -76       -> 10000101000
    -74       -> 10111001000000
    -72       -> 11101000
    -70       -> 100110001
    -69       -> 1100010100
    -68       -> 1010010010
    -66       -> 1101011111
    -65       -> 10000101100
    -64       -> 11010100
    -62       -> 10011001111
    -61       -> 111110100
    -60       -> 11011000
    -58       -> 100101111
    -57       -> 11110011
    -56       -> 10011011
    -55       -> 11111110
    -54       -> 1000111110
    -52       -> 110001000
    -51       -> 10100011
    -50       -> 10100110110
    -48       -> 1011001
    -44       -> 100001001
    -42       -> 10100010
    -40       -> 1011000
    -39       -> 1010011100
    -38       -> 111111000
    -37       -> 101000001
    -36       -> 10010010
    -35       -> 100001000
    -34       -> 111110101
    -33       -> 1100001011
    -32       -> 10000110
    -30       -> 1000001101
```

```
    -29       -> 11110001
    -28       -> 1101000
    -26       -> 1101001
    -24       -> 00
    -22       -> 111000
    -20       -> 100110000
    -19       -> 10010011
    -18       -> 10001100
    -17       -> 10010000
    -16       -> 110010
    -14       -> 101010
    -13       -> 10010001
    -12       -> 1001111
    -11       -> 1000100
    -10       -> 10111011
    0       -> 01
    10       -> 10000001
    11       -> 1110110
    12       -> 1001110
    13       -> 10000111
    14       -> 110011
    16       -> 1000101
    17       -> 1101111
    18       -> 10001101
    19       -> 10110101
    20       -> 101001010
    22       -> 111001
    24       -> 101011
    26       -> 1110111
    28       -> 1011110
    29       -> 10110111
    30       -> 1000001110
    32       -> 1111010
    33       -> 110110101
    34       -> 101110001
    35       -> 101000010
    36       -> 10000000
    37       -> 101000011
    38       -> 100101110
    39       -> 1001011001
```

```
    40      -> 1101110
    42      -> 10011010
    44      -> 11111111
    48      -> 1011111
    50      -> 1100000011000
    51      -> 10110110
    52      -> 1011110000
    54      -> 10000101101
    55      -> 100101000
    56      -> 1111011
    57      -> 11110010
    58      -> 100000100
    60      -> 10111010
    61      -> 1001011011
    62      -> 1111100101
    64      -> 11011001
    65      -> 11010111001
    66      -> 1010010001
    68      -> 1000111100
    69      -> 1010011101
    70      -> 100101001
    72      -> 11110000
    74      -> 1100000011001
    76      -> 1000010101111
    77      -> 1001100011011
    78      -> 1010011001
    80      -> 11010101
    81      -> 11111101001
    84      -> 1010000000
    85      -> 1011100100111
    87      -> 1101011010
    88      -> 10000101111
    90      -> 10111001111
    91      -> 1001100110000
    92      -> 11000000000011
    95      -> 10010101110
    96      -> 1000000101
    98      -> 1101011000
    99      -> 11111101011
    100      -> 10100110101
```

```
    102      -> 10111001110
    104      -> 1111110111
    108      -> 1111110110
    109      -> 11000000000000
    110      -> 1101101100
    111      -> 1100000011111
    112      -> 110001001
    113      -> 11000000000001
    114      -> 11111000110
    116      -> 11101011000
    117      -> 1100000010001
    120      -> 110110100
    121      -> 1100000010000
    126      -> 10100110100
    128      -> 10000101110
    130      -> 11111101010
    132      -> 1000111111
    136      -> 11000000000110
    138      -> 11000000000111
    140      -> 11101011001
    143      -> 1100000010101
    144      -> 110000110
    149      -> 1011100101010
    150      -> 1000010101010
    152      -> 11000000000100
    153      -> 1000010101001
    154      -> 10010101101
    156      -> 1101101101
    160      -> 1010011110
    165      -> 1011100101001
    168      -> 10111001101
    169      -> 1011100101110
    170      -> 11000000000101
    171      -> 11000000011010
    174      -> 11000000011011
    176      -> 1000111101
    180      -> 1101101111
    182      -> 11000101110
    187      -> 11000000011000
```

```
    190      -> 1011100100001
    192      -> 110001101
    196      -> 1100000010000
    200      -> 1011100100110
    204      -> 11000000011001
    208      -> 1010010000
    210      -> 1011100100100
    216      -> 1000010101011
    220      -> 1100000010101
    224      -> 110001111
    228      -> 100101010110
    230      -> 1011100100101
    231      -> 11111000100
    234      -> 11000000011110
    240      -> 111111001
    242      -> 1001100110001
    250      -> 11000000011111
    252      -> 100110011010
    253      -> 11000000011100
    256      -> 111110110
    260      -> 1100000011001
    264      -> 1111100000001
    270      -> 1100000011110
    272      -> 100011101
    275      -> 1111100000111
    276      -> 1111100000101
    286      -> 11000000011101
    288      -> 100011100
    294      -> 1100000011100
    297      -> 1111100011001
    304      -> 1100011110
    308      -> 1100000011101
    310      -> 1100000010010
    312      -> 11000000010010
    319      -> 1111100011101
    320      -> 1100000111
    322      -> 11000000010011
    330      -> 100101010100
    336      -> 1001011010
    340      -> 100101010101
```

```
    350      -> 11000000010000
    352      -> 1110100100
    360      -> 1100000010111
    368      -> 1010011000
    372      -> 11000000010001
    374      -> 11101011011
    380      -> 11000000010110
    384      -> 1000001100
    385      -> 11000000010111
    400      -> 1001100100
    407      -> 11000000010100
    416      -> 1001011000
    418      -> 1011100101000
    420      -> 11000000010101
    432      -> 1010010011
    440      -> 11000001101010
    444      -> 1111100000100
    448      -> 1010011111
    451      -> 11000001101011
    456      -> 1111100011010
    460      -> 11000001101000
    462      -> 1011100101101
    464      -> 10100110111
    468      -> 11000001101001
    480      -> 1011100100011
    484      -> 11000001101110
    496      -> 11000001101111
    504      -> 11000001101100
    506      -> 11000001101101
    560      -> 10011001100110
    561      -> 10011001100111
    572      -> 10011001100100
Before Huffman Coding Total No. of  Bits = 524288
After  Huffman Coding Total No. of  Bits = 179948
Redundancy - 65.677643 %
```

CONCLUSION

While Encoding using any of the Encoding Techniques, One Can observe that the most probable or most repeated element after the quantization operation is 0. Because, There are only a fewer number of low-frequency Discrete Cosine Transform (DCT) Coefficients that contain significant signal energy that allows for compact representation. The disadvantage with the DCT based Image Compression is that it is lossy compression Technique. Once an image is compressed using DCT, it cannot be reverted or decompressed back to exactly the same state as before which is compromised by the significant redundancy in the image size. One can observe that in the results, the probability of symbol 0 is more than 86% and Hence using the Huffman Coding Technique, a Codeword with least number of bits '01' is assigned to the zero. It reduces the data required to represent the image size by

A significant number. The implemented model in the project is used in the standard Joint Photographic Experts Group (JPEG) standard.

In this Mini Project, a 8-bit 256*256 grayscale image is taken as the input to the Image compression model that comprises DCT block and Huffman Coding Block as its major components. The whole Image Compression Model is implemented on DSP process TMS320C6748

And using Code Composer Studio (CCS) software. This implementation took a lot of time because of the computational complexity in the Huffman Coding Technique. There are many other compression techniques using Wavelet Transform , Fourier Transform, Predictive Coding, Entropy based Encoding techniques, Sub-Sampling based techniques which have their own merits and de- merits. Based on the requirement and redundancy required, multiple Techniques can be used.

References:

[1]Image Compression Using Fast 2-D DCT Technique Anjali Kapoor, Renu Dhir Department of computer science and engineering, National Institute of Technology, Jalandhar, India, International Journal on Computer Science and Engineering (IJCSE)

[2]Image Compression Using the Discrete Cosine Transform Andrew B. Watson NASA Ames Research Center

[3]Ahmed, N., Natarajan, T., & Rao, K. R. (1974). Discrete Cosine Transform. IEEE Transactions on Computers, 100(1), 90-93.

[4] W. Yan, P. Hao, C. Xu, ”Matrix factorization for fast DCT algorithms”, IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 3, 2006.