

Image Compression using DCTQ

A mini project is submitted in partial completion of Image Compression by
Discrete Cosine Transform and Quantization using OBC and CORDIC

by

Lakshmi Sudha Rani Nimmakayala(N150435),Sravanthi
Kadagala(N150673),Syam Sundar Malla(N150404)



Department of Electronics and Communication Engineering
Rajiv Gandhi University of Knowledge Technologies
Nuzvid,India-533202

Certificate

It is certified that the work contained in this term project entitled '**Image Compression using DCTQ**' by Lakshmi Sudha Rani, Sravanthi, Syam Sundar has been carried out under my supervision and that it has not been submitted elsewhere for a degree.

Date: July 7, 2020.

Mr.Sk.Irfan Ali
Asst.Professor
RGUKT,Nuzvid

Mr.P.Shyam
Asst.Professor
RGUKT,Nuzvid

Acknowledgements

We would like to thank Mr.Sk.Irfan Ali, Asst.Professor and Mr.P.Shyam, Asst.Professor, Dept.of Electronics and Communication Engineering, Rajiv Gandhi University of Knowledge Technologies-Nuzvid, for imparting their deep sense of knowledge,timely suggestions,sincere guidance throughout this period of project work. We would like to thank our seniors who helped us to resolve the problems faced in the coding.

Abstract

Image Compression is the application of data compression on digital images, where tolerable degradation is accepted. With the wide use of computers, large scale of storage and transmission of data, efficient ways of storing data has become necessary. Image Compression minimizes the size of Mega bytes of graphics file into bytes without degrading the quality. It is used to reduce redundancy (i.e., avoiding duplicate data).

In this paper, it is being attempted to implement basic JPEG compression using MATLAB and Verilog. JPEG image compression standard use DCT which is fast transform. DCT is mostly used compression technique. DCT is a mathematical function that transforms digital image data from the spatial domain to frequency domain. To perform matrix multiplication, OBC is used, which is multiplier less technique to obtain inner product. After performing DCT, information is compacted to lower coefficients. Then quantization is performed to further compression. When we want to retrieve image, inverse operation can be done.

Contents

1	Image Compression	4
1.1	Introduction	4
1.1.1	What is an Image?	4
1.1.2	Need for Image Compression	5
1.1.3	Types of Image Compression	5
1.1.4	Image Compression Techniques	5
1.1.5	Applications of Image Compression	6
2	Literature Survey	7
2.1	Block diagram of Image Compression	7
2.2	2D-Discrete Cosine Transform and Quantization	8
2.2.1	Algorithm for Parallel Matrix Multiplication for DCTQ	8
2.2.2	Distributed Arithmetic (DA)	10
2.2.3	Offset Binary Coding(OBC)	12
3	Proposed Method	15
3.1	Verilog implementation of DCTQ with OBC	15
3.1.1	Block diagram of DCTQ	15
3.1.2	Reconstruction	17
4	Results and Simulations	18
4.1	Outputs of DCTQ code in simulation window	22
5	Conclusion and Future works	26
5.1	Conclusion	26
5.2	Future Works	26
6	References	27
A	Matlab code for image compression	29

B	Verilog code for Image Compression	31
B.1	Matlab code to generate text file for coe file	31
B.2	Verilog code for OBC	31
B.3	Image Compression using DCT and IDCT code	40
B.4	Matlab code to convert verilog output text file to image	48

List of Figures

2.1	Block diagram of Image Compression	7
2.2	Look up table for DA	11
2.3	DA-based implementation of a four-term inner-product computation	12
2.4	DA-based implementation using OBC (offset binary coding). .	14
2.5	LUT size reduction using OBC (offset binary coding).	14
3.1	DCTQ block diagram	15
3.2	C matrix	16
3.3	Q matrix	16
3.4	IDCTQ block diagram	17
4.1	coe file of RGUKT academic block image).	18
4.2	Searching for BMG.	19
4.3	Choosing Memory type as Single port ROM	19
4.4	choosing required settings	20
4.5	Loading the coe file	20
4.6	This appears after clicking OK	21
4.7	Copying the code	21
4.8	C and C^T matrix	22
4.9	$CX(1,1)=558$	22
4.10	$CXC^T(1,1)=1573$	23
4.11	$C^TDCT(1,1)=553$	23
4.12	IDCT(1,1)=196	24
4.13	IDCT(1,1)=196	25
4.14	reconstructed image from verilog output file	25
B.1	reconstructed image from verilog output file	49

Chapter 1

Image Compression

1.1 Introduction

Image compression is a type of data compression applied to digital images, to reduce their cost for storage or transmission. Demand for communication of multimedia data through the telecommunication network and accessing the multimedia data through Internet is growing extensively. With the use of digital cameras, requirements for storage, manipulation, and transfer of digital images, has grown immensely. These image files can be very large and can occupy a lot of memory. Even with the advances in bandwidth and storage capabilities, if images were not compressed many applications would be too costly. Therefore development of efficient techniques for image compression has become quite necessary.

1.1.1 What is an Image?

Basically, an image is a rectangular array of dots, called pixels. The size of the image is the number of pixels (width x height). Every pixel in an image is a certain color. When dealing with a black and white (where each pixel is either totally white, or totally black) image, the choices are limited since only a single bit is needed for each pixel. This type of image is good for line art, such as a cartoon in a newspaper. Another type of colorless image is a gray scale image. Gray scale images, often wrongly called "black and white" as well, use 8 bits per pixel, which is enough to represent every shade of gray that a human eye can distinguish. When dealing with color images, things get little trickier.

1.1.2 Need for Image Compression

An image, 1024 pixel x 1024 pixel x 24 bit, without compression, would require 3 MB of storage and 7 minutes for transmission, utilizing a high speed, 64 Kbit/s, ISDN line. If the image is compressed at a 10:1 compression ratio, the storage requirement is reduced to 300 KB and the transmission time drops to under 6 seconds. Seven 1 MB images can be compressed and transferred to a floppy disk in less time than it takes to send one of the original files, uncompressed, over an AppleTalk network.

In a distributed environment large image files remain a major bottleneck within systems. Compression is an important component of the solutions available for creating file sizes of manageable and transmittable dimensions. Increasing the bandwidth is another method, but the cost sometimes makes this a less attractive solution. Platform portability and performance are important in the selection of the compression/decompression technique to be employed. The easiest way to reduce the size of the image file is to reduce the size of the image itself. By shrinking the size of the image, fewer pixels need to be stored and consequently the file will take less time to load.

1.1.3 Types of Image Compression

- Lossless compression
- Lossy compression

The lossless compression technique, as the name indicates, produces no loss in the quality of image. This technique is used in places where the quality and accuracy of image is extremely important and cannot be compromised on. Some of the examples are technical drawings, medical images etc.

Lossy compression technique is one which produces a minor loss of quality to the output image. This minor loss is almost invisible and hard to identify. This technique finds use where minor alteration or loss of quality causes no problem like in photographs. There are different methods and algorithms used in lossless and lossy compression.

1.1.4 Image Compression Techniques

Methods for lossy compression:

- Transform coding – This is the most commonly used method.
- Discrete Cosine Transform (DCT) – The most widely used form of lossy compression. It is a type of Fourier-related transform, and was

originally developed by Nasir Ahmed, T. Natarajan and K. R. Rao in 1974. The DCT is sometimes referred to as "DCT-II" in the context of a family of discrete cosine transforms. It is generally the most efficient form of image compression. DCT is used in JPEG, the most popular lossy format.

- Chroma subsampling. This takes advantage of the fact that the human eye perceives spatial changes of brightness more sharply than those of color, by averaging or dropping some of the chrominance information in the image.
- Fractal compression.

Methods for lossless compression:

- Run-length encoding – used in default method in PCX and as one of possible in BMP, TGA, TIFF
- Area image compression
- Predictive coding – used in DPCM
- Entropy encoding – the two most common entropy encoding techniques are arithmetic coding and Huffman coding
- Adaptive dictionary algorithms such as LZW – used in GIF and TIFF
- DEFLATE – used in PNG, MNG, and TIFF
- Chain codes

1.1.5 Applications of Image Compression

Image compression has applications in transmission and storage of information. Image transmission applications are in broadcast television, remote sensing via satellite, military communications via aircraft, radar and sonar, teleconferencing, computer communications. Image storage applications are in education and business, documentation, medical image that arises in computer tomography, magnetic resonance imaging and digital radiology, motion pictures, satellite images weather maps, geological surveys and so on.

Chapter 2

Literature Survey

Number of bits required to represent the information in an image can be minimized by removing the redundancy present in it. There are three types of redundancies:

- (i) spatial redundancy, which is due to the correlation or dependence between neighbouring pixel values;
- (ii) spectral redundancy, which is due to the correlation between different color planes or spectral bands;
- (iii) temporal redundancy, which is present because of correlation between different frames in images.

Image compression research aims to reduce the number of bits required to represent an image by removing the spatial and spectral redundancies as much as possible.

2.1 Block diagram of Image Compression

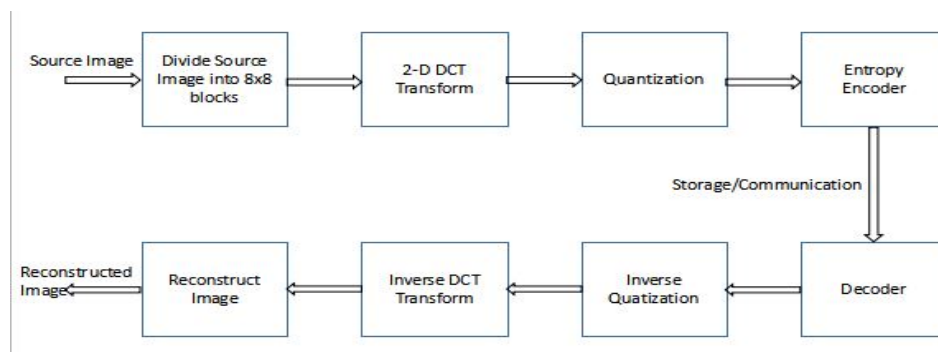


Figure 2.1: Block diagram of Image Compression

2.2 2D-Discrete Cosine Transform and Quantization

The discrete cosine transform closely approximates the Karhunen Loeve Transform (KLT), which is known to be optimal in the sense of de-correlating the data and maximizing the energy packed into the lowest order coefficients. However, unlike the KLT, the DCT involves much less computational complexity in implementation, and is, therefore, preferred in image and video compression work. A comprehensive treatment of DCT algorithms and applications can be found in reference [1]. The DCT, which exploits the spatial redundancy to prepare the ground for effective compression, has played a key role in video data compression standards such as JPEG, MPEG 1, MPEG 2, and H.26X.

Over the years, considerable amount of research work have been carried out in proposing new algorithms for the DCT and implementing them on general-purpose computers, DSPs, and ASICs. Direct 2-D approach [2] results in less parallelism, whereas separable row-column 1-D approach [3, 4] yields a faster algorithm. The authors of reference [4] have implemented 8×8 DCT by software on Pentium operating at 200 MHz. The fast algorithms with minimum numbers of multiplication are often realized by flexible software approaches on the DSPs. The speed requirement can be met by a high-speed DSP but it still needs to pay high hardware cost due to its inherent complexity of multipliers.

ICs have been fabricated [44, 45] for still image compression and decompression conforming to JPEG standard. In order to meet the real-time requirements, DCT and IDCT implementations use efficient and dedicated hardware. Furthermore, a very stringent utilization of VLSI technology is required for the design to meet the required speed criteria. A linear, highly pipelined, parallel algorithm and architecture have been proposed and implemented for 2D-DCT and Quantization on FPGAs.

2.2.1 Algorithm for Parallel Matrix Multiplication for DCTQ

DCT is an orthogonal transform consisting of a set of vectors that are sampled cosine functions. 2D-DCT of a block of size 8×8 pixels of an image is defined as

$$DCT(u, v) = \frac{1}{4}c(u)c(v) \sum_{x=0}^7 \sum_{y=0}^7 f(x, y) \cos\left[\frac{(2x+1)u\pi}{16}\right] \cos\left[\frac{(2y+1)v\pi}{16}\right]$$

where $f(x, y)$ is the pixel intensity and

$c(u) = c(v) = \frac{1}{\sqrt{2}}$ for $u = v = 0$ and
 $= 1$ for $u, v = 1$ to 7 .

The DCT can be expressed conveniently in a matrix form:

$$DCT = CXCT^T \quad (2.1)$$

where X is the input image matrix, C the cosine coefficient matrix, and CT , its transpose with constants $(1/2)c(u)$ and $(1/2)c(v)$ absorbed in C and CT matrices respectively. For a clearer understanding, the DCT may be expressed in an expanded form:

$$DCT = \begin{bmatrix} c_{00} & c_{01} & \dots & c_{07} \\ c_{10} & c_{11} & \dots & c_{17} \\ \vdots & \vdots & \vdots & \vdots \\ c_{70} & c_{71} & \dots & c_{77} \end{bmatrix} \begin{bmatrix} x_{00} & x_{01} & \dots & x_{07} \\ x_{10} & x_{11} & \dots & x_{17} \\ \vdots & \vdots & \vdots & \vdots \\ x_{70} & x_{71} & \dots & x_{77} \end{bmatrix} \begin{bmatrix} c_{00} & c_{10} & \dots & c_{70} \\ c_{01} & c_{11} & \dots & c_{71} \\ \vdots & \vdots & \vdots & \vdots \\ c_{07} & c_{17} & \dots & c_{77} \end{bmatrix}$$

The two-stage matrix multiplication shown above can be implemented by parallel architecture, wherein eight partial products, which are the row vectors of CX generated in the first stage, are fed to the second stage. Subsequently, multiplying row vector of CX by the C^T matrix generates eight DCT coefficients, corresponding to a row of $C X C^T$. While computing the $(i + 1)$ th partial products of CX , the i th row DCT coefficients can also be computed simultaneously since the i th partial products of CX are already available. Application of DCT on an 8×8 pixel block, thus, generates 64 coefficients in a raster scan order.

Quantized outputs can be obtained by dividing each of the 64 DCT coefficients by the corresponding quantization table values given in the standards as per the expression:

$$DCTQ(u, v) = DCT(u, v)/q(u, v); \quad u, v = 0 \text{ to } 7 \quad (2.2)$$

These stages can be pipelined in such a way that one DCTQ output can be generated every clock cycle.

Similarly, inverse quantization can be computed by multiplying each of the 64 DCTQ coefficients by the corresponding quantization table values as per the expression:

$$DCT(u, v) = DCTQ(u, v)q(u, v); \quad u, v = 0 \text{ to } 7 \quad (2.3)$$

The image can be reconstructed from the DCT (u, v) by evaluating the (inverse DCT) product of the matrix:

$$IDCT = CT(DCT)C \quad (2.4)$$

In order to perform above matrix multiplications, DA or OBC is used.

2.2.2 Distributed Arithmetic (DA)

Distributed arithmetic (DA) is a technique for multiplier less implementation of inner product of two vectors, particularly when one of the vectors is a constant and known a prior. Inner product, also known as dot product or simply the weighted sum ($\sum A[i] * X[i]$).

DA-based implementation has been shown to provide area, power, performance efficiency for both ASIC as well as FPGA-based implementations. DA replaces multipliers with table look-ups of precalculated partial products. Therefore it is hardware efficient to perform vector multiplication.

Consider the following sum of products:

$$Y = \sum_{n=1}^N A[n] * X[n] \quad (2.5)$$

where $A[n]$ for $n = 1, 2, \dots, N$, are the fixed coefficients and $X[n]$ for $n = 1, 2, \dots, N$, constitute the variable vector. If $X[n]$ is a 2's-complement binary number scaled such that $|x[n]| < 1$ and K is the word size of each $X[n]$, then it can be represented as

$$X[n] = -b_{n0} + \sum_{k=1}^{K-1} b_{nk} * 2^{-k} \quad (2.6)$$

where the b_{nk} are the bits 0 or 1, b_{n0} is the sign bit and $b_{n,K-1}$ is the LSB. Using Eqs (2.5) in (2.6) we get

$$Y = \sum_{n=1}^N A[n] * (-b_{n0} + \sum_{k=1}^{K-1} b_{nk} * 2^{-k}) \quad (2.7)$$

Interchanging the order of the summations we get

$$Y = \sum_{k=1}^{K-1} \sum_{n=1}^N (A[n].b_{nk}) * 2^{-k} - \sum_{n=1}^N A[n] * b_{n0} \quad (2.8)$$

$$C_0 = - \sum_{n=1}^N A[n] * b_{n,0} \quad (2.9)$$

$$C_k = \sum_{n=1}^N A[n] * b_{n,k} \text{ where } k \neq 0 \quad (2.10)$$

$$Y = C_0 + \sum_{k=1}^{K-1} C_k * 2^{-k} \quad (2.11)$$

$$Y = \sum_{k=0}^{K-1} C_k * 2^{-k} \quad (2.12)$$

For example:

$$N=4; K=8;$$

$$Y = \sum_{k=0}^{8-1} C_k * 2^{-k}$$

$$= C_0 + C_1 2^{-1} + C_2 2^{-2} + C_3 2^{-3} + C_4 2^{-4} + C_5 2^{-5} + C_6 2^{-6} + C_7 2^{-7}$$

$$C_6 = \sum_{n=1}^4 A[n] * b_{n,6}$$

$$= A[1] * b_{1,6} + A[2] * b_{2,6} + A[3] * b_{3,6} + A[4] * b_{4,6}$$

DA LUT Section

b1	b2	b3	b4	y
0	0	0	0	0
0	0	0	1	A[4]
0	0	1	0	A[3]
0	0	1	1	A[3]+A[4]
0	1	0	0	A[2]
0	1	0	1	A[2]+A[4]
0	1	1	0	A[2]+A[3]
0	1	1	1	A[2]+A[3]+A[4]
1	0	0	0	A[1]
1	0	0	1	A[1]+A[4]
1	0	1	0	A[1]+A[3]
1	0	1	1	A[1]+A[3]+A[4]
1	1	0	0	A[1]+A[2]
1	1	0	1	A[1]+A[2]+A[4]
1	1	1	0	A[1]+A[2]+A[3]
1	1	1	1	A[1]+A[2]+A[3]+A[4]

Figure 2.2: Look up table for DA

DA Block diagram

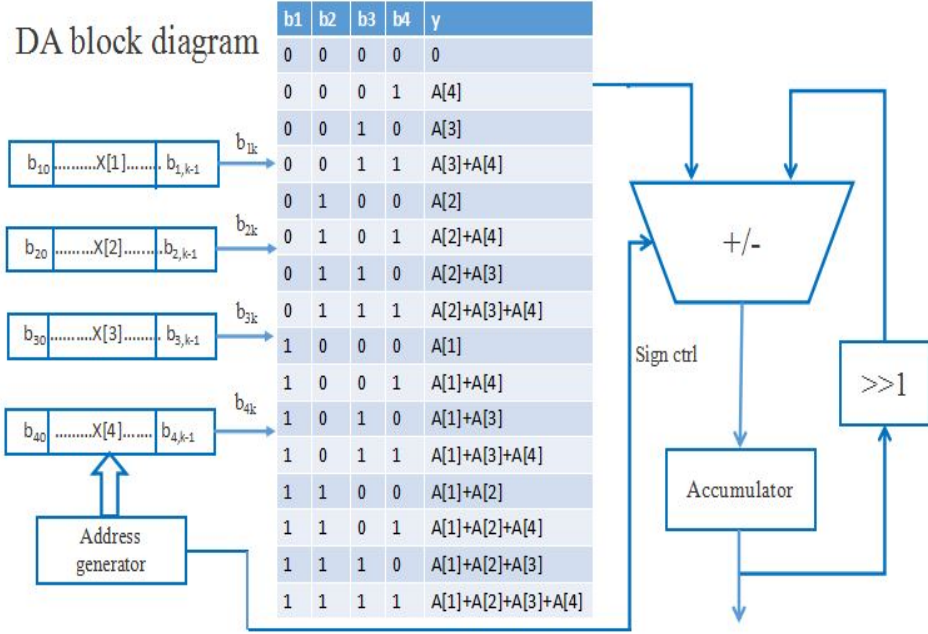


Figure 2.3: DA-based implementation of a four-term inner-product computation

2.2.3 Offset Binary Coding(OBC)

It is an optimised DA. The size of the LUT can be reduced to half by using offset binary coding that assumes each bit of the input to have a value of either 1 or 1 (as against 1 or 0 in case of the conventional binary coding).

Consider the following sum of products:

$$Y = \sum_{n=1}^N A[n] * X[n] \quad (2.13)$$

where $A[n]$ for $n = 1, 2, \dots, N$, are the fixed coefficients and $X[n]$ for $n = 1, 2, \dots, N$, constitute the variable vector. If $X[n]$ is a 2's-complement binary number scaled such that $|x[n]| < 1$ and K is the word size of each $X[n]$, then it can be represented as

$$X[n] = -b_{n0} + \sum_{k=1}^{K-1} b_{nk} * 2^{-k} \quad (2.14)$$

$$X[n] = -b_{n0}^- + \sum_{k=1}^{K-1} b_{nk}^- * 2^{-k} + 2^{-(K-1)} \quad (2.15)$$

where the b_{nk} are the bits 0 or 1, b_{n0} is the sign bit and $b_{n,K-1}$ is the LSB. $X[n]$ can be written as follows:

$$X[n] = \frac{1}{2}(X[n] - (-X[n])) \quad (2.16)$$

From Eqs (2.14), (2.15) and (2.16), we get

$$X[n] = \frac{1}{2}(-(b_{n0} - b_{n0}^-) + \sum_{k=1}^{K-1} (b_{nk} - b_{nk}^-) * 2^{-k} - 2^{-(K-1)}) \quad (2.17)$$

where b_{nk}^- denotes the complement of bit b_{nk} .

Let $c_{nk} = (b_{nk} - b_{nk}^-)$ for $k \neq 0$ and $c_{n0} = -(b_{n0} - b_{n0}^-)$, then from Eqs (2.13) and (2.17)

$$Y = \sum_{k=0}^{K-1} \left(\sum_{n=1}^N \frac{A[n]}{2} * c_{nk} \right) * 2^{-k} - \left(\sum_{n=1}^N \frac{A[n]}{2} \right) * 2^{-(K-1)} \quad (2.18)$$

It can be noted that $(b_{nk} = 0)$ translates to $(C_{nk} = 1)$ and $(b_{nk} = 1)$ translates to $(C_{nk} = 1)$. The first term in Eq. (2.18) can thus be computed using the LUT as shown in 2.4. The second term is a constant that can be added to the accumulator at the end of the accumulation of LUT outputs, to get the final result.

It can be noted that the contents of the upper half of the LUT are same as the lower half, but with the sign reversed. This symmetry can be exploited to store only the upper half of the LUT, and using XORs to reverse the sign when required. The resultant DA structure is shown in 2.5.

As can be seen from Figure 2.5, the LUT size is reduced to half, but there is an overhead of XOR gates. The power is impacted by the “N” XORs performed every cycle, and also due to one more addition required $\frac{1}{2}(A[1] + A[2] + A[3] + A[4])$ over the baseline DA implementation.

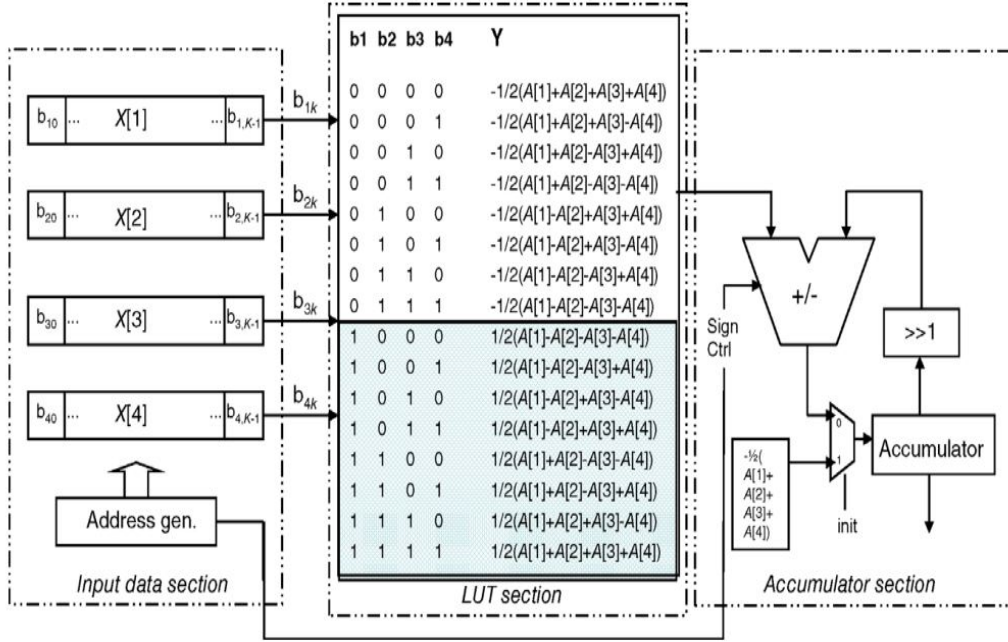


Figure 2.4: DA-based implementation using OBC (offset binary coding).

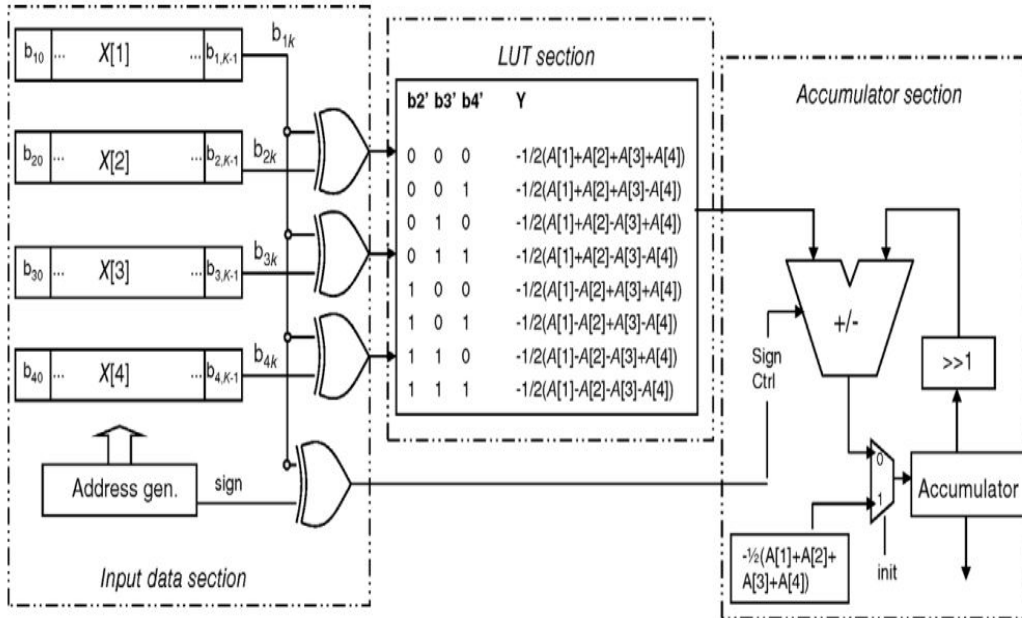


Figure 2.5: LUT size reduction using OBC (offset binary coding).

Chapter 3

Proposed Method

3.1 Verilog implementation of DCTQ with OBC

The Process

- 1.The image is broken into 8x8 blocks of pixels.
- 2.Working from left to right , top to bottom, the DCT is applied to each block.
- 3.Each block is compressed through quantization.
- 4.The array of compressed blocks that constitute the image is stored in drastically reduced amount of space.
- 5.When desired ,the image is reconstructed through decompression, a process that uses the Inverse Discrete Cosine Transform (IDCT).

3.1.1 Block diagram of DCTQ

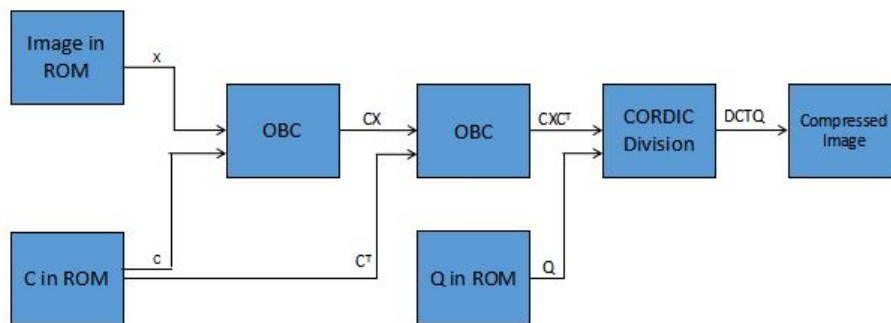


Figure 3.1: DCTQ block diagram

First image pixels are loaded in the form of 8X8 blocks in ROM. It is multiplied (matrix multiplied) with C matrix. This multiplication is done using OBC

$$C = \begin{bmatrix} .3536 & .3536 & .3536 & .3536 & .3536 & .3536 & .3536 & .3536 \\ .4904 & .4157 & .2778 & .0975 & -.0975 & -.2778 & -.4157 & -.4904 \\ .4619 & .1913 & -.1913 & -.4619 & -.4619 & -.1913 & .1913 & .4619 \\ .4157 & -.0975 & -.4904 & -.2778 & .2778 & .4904 & .0975 & -.4157 \\ .3536 & -.3536 & -.3536 & .3536 & .3536 & -.3536 & -.3536 & .3536 \\ .2778 & -.4904 & .0975 & .4157 & -.4157 & -.0975 & .4904 & -.2778 \\ .1913 & -.4619 & .4619 & -.1913 & -.1913 & .4619 & -.4619 & .1913 \\ .0975 & -.2778 & .4157 & -.4904 & .4904 & -.4157 & .2778 & -.0975 \end{bmatrix}$$

Figure 3.2: C matrix

The resultant obtained from above is again multiplied with C^T using OBC. Thus DCT is done on image.

Quantization

Our 8X8 block of DCT coefficients is now ready for compression by quantization. A remarkable and highly useful feature of the JPEG process is that in this step, varying levels of image compression and quality are obtained through selection of specific quantization matrices. This enables the user to decide on quality levels. Subjective experiments involving the human visual system have resulted in the JPEG standard quantization matrix. With a high quality level of 50, this matrix renders both high compression and excellent decompressed image quality.

$$Q_{50} = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Figure 3.3: Q matrix

DCT output from OBC is divided by Q matrix using cordic. This resultant coefficients which has maximum information can be stored or transmitted. While receiving or retrieving the image Inverse operation is performed.

3.1.2 Reconstruction

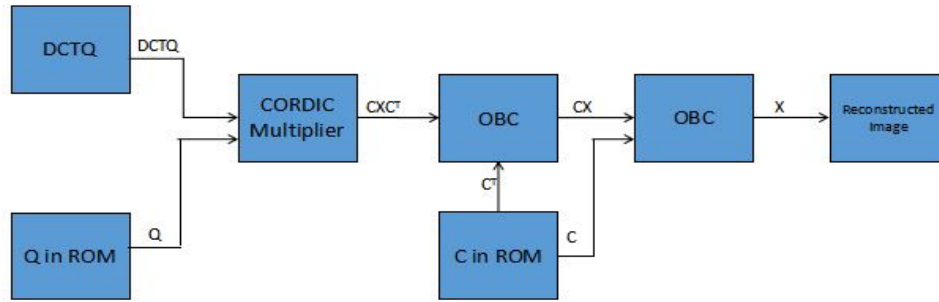


Figure 3.4: IDCTQ block diagram

Chapter 4

Results and Simulations

This project is implemented using both Matlab and Verilog.

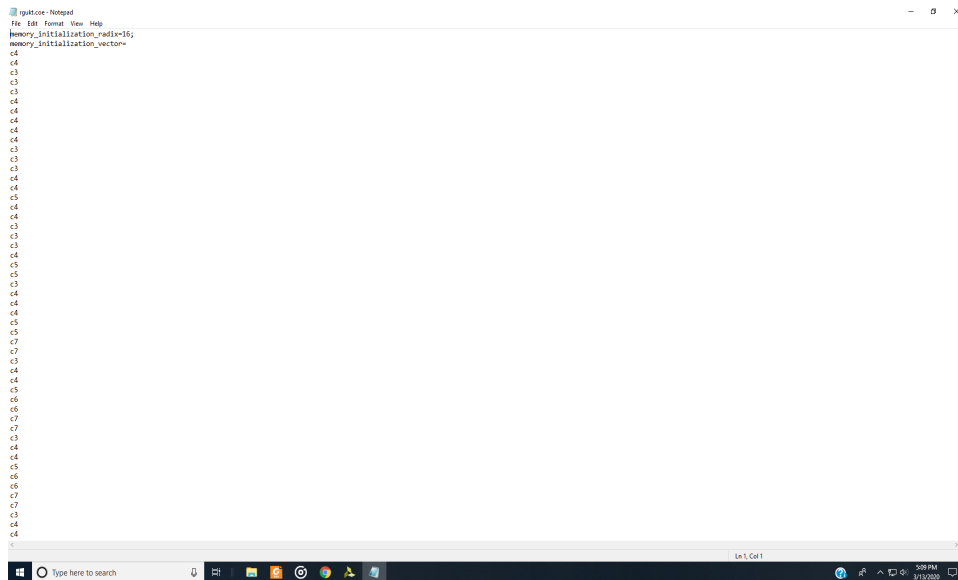
4.1 is the pixel values of an image that needs to be compressed is converted into coe file. B.1 code gives text file. we need to write two lines at the beginning of the file. i.e.,

memory_initialization_radix=16;

memory_initialization_vector=

and finally after all pixel values,place semicolon(;) symbol.

Final output:



```
igbtcoe - Notepad
File Edit Format View Help
memory_initialization_radix=16;
memory_initialization_vector=
c4
c4
c3
c3
c3
c4
c4
c4
c4
c3
c3
c3
c4
c4
c5
c4
c4
c3
c3
c3
c4
c4
c5
c5
c3
c4
c4
c4
c5
c5
c7
c3
c4
c4
c5
c6
c6
c7
c3
c4
c4
c5
c6
c6
c7
c3
c4
c4
```

Figure 4.1: coe file of RGUKT academic block image).

This rgukt.coe file has to be loaded in vivado using IP Catalog.
Search for block memory generator.

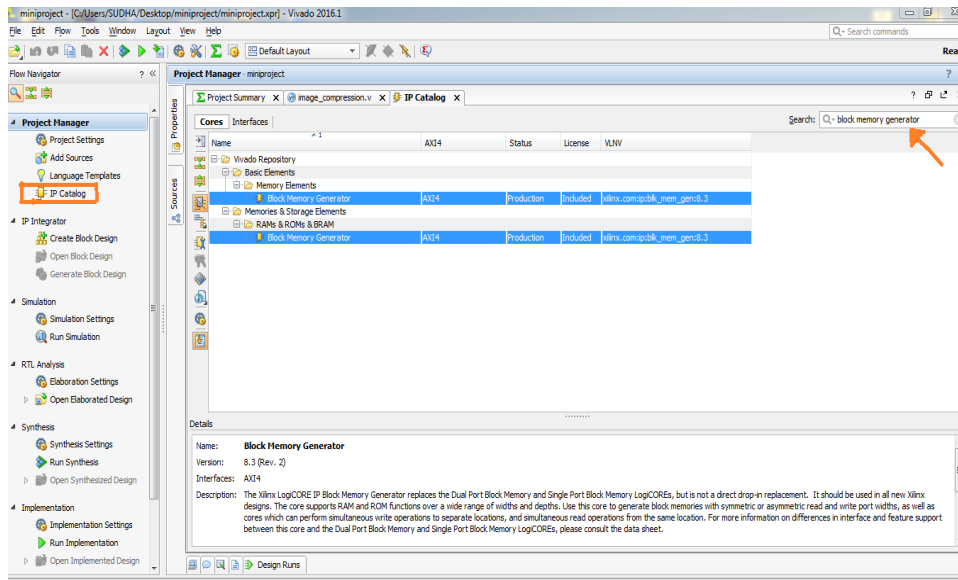


Figure 4.2: Searching for BMG.

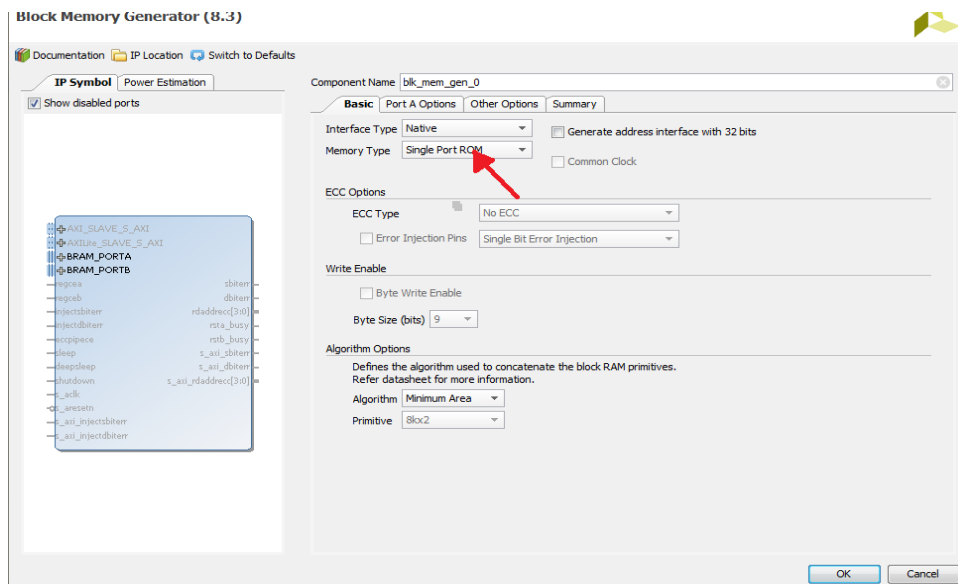


Figure 4.3: Choosing Memory type as Single port ROM .

- Keep the each pixel size as 12 bits.
- port depth=16386 since total no.of lines in coe file is 16386(128X128+2)
- port type is always enabled

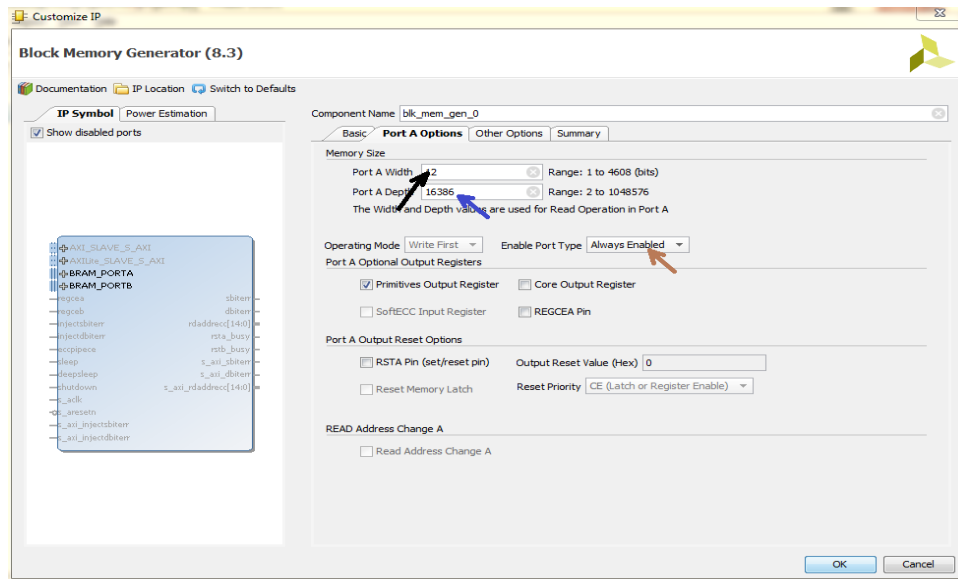


Figure 4.4: choosing required settings

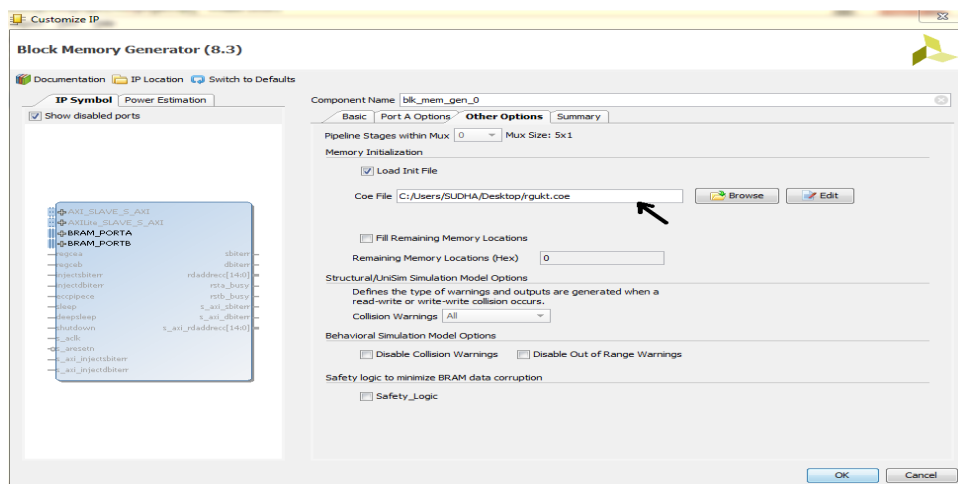


Figure 4.5: Loading the coe file

C matrix and C^T matrix

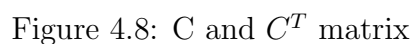
[illegible]

Figure 4.9: CX(1,1)=558

Output after calculating $CXC^T(1,1)$:

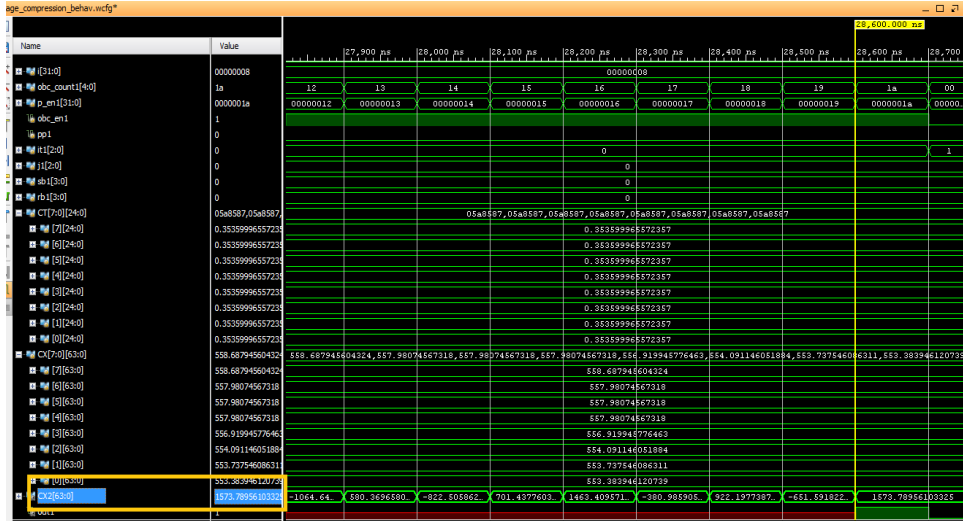


Figure 4.10: $CXC^T(1,1)=1573$

Output after calculating $C^T DCT(1,1)$:

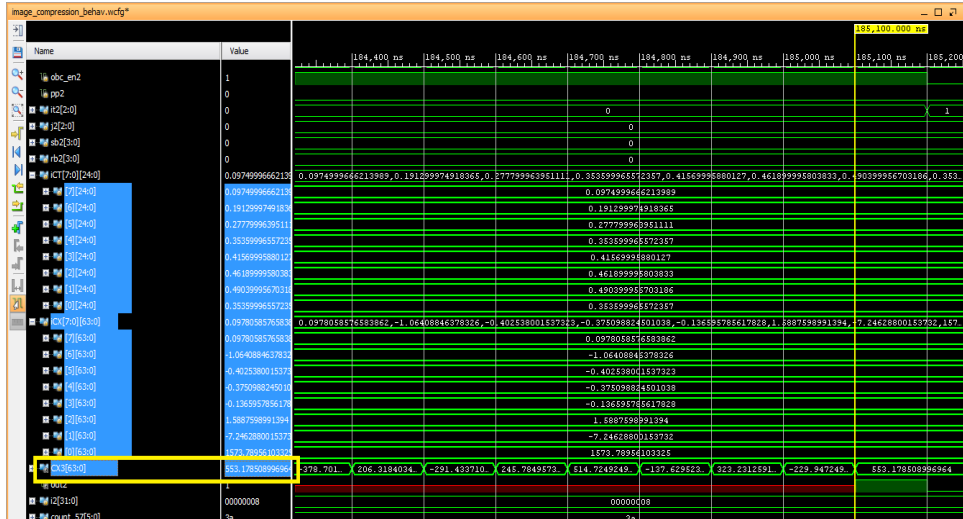


Figure 4.11: $C^T DCT(1,1)=553$

Output after calculating IDCT(1,1):

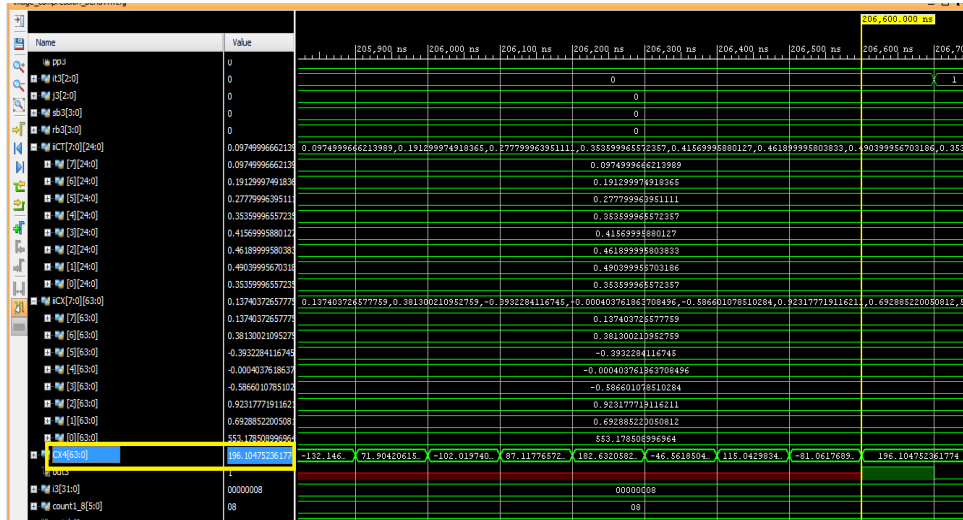


Figure 4.12: IDCT(1,1)=196

Matlab output for above results:

```

10 C = double(D);
COMMAND WINDOW

CX(1,1)=5.533840e+02
CXCT(1,1)=1.573790e+03
CT*DCT(1,1)=5.476110e+02
IDCT(1,1)=1.922214e+02
>>

```

So results in both vivado and matlab are almost same.

Here is the text file generated by vivado.

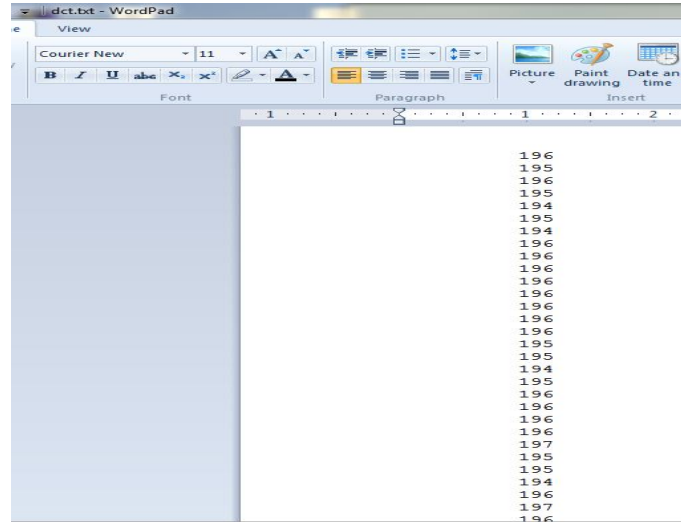


Figure 4.13: $IDCT(1,1)=196$

Figure 4.14 is the reconstructed image from matlab by using vivado generated output text file.

Reconstructed image



Figure 4.14: reconstructed image from verilog output file

Chapter 5

Conclusion and Future works

5.1 Conclusion

Implementation of Discrete Cosine Transform(DCT) and Inverse Discrete Cosine Transform(IDCT) is completed. In this we presented a new hardware for DCTQ using OBC with pipelining. OBC is a multiplier less matrix multiplication uses LUT. Image is taken from Matlab and processed in Verilog(Vivado) and again viewed in Matlab. Image is reconstructed and looks like the original image even though some deviation in pixel values after processing in verilog.

5.2 Future Works

- We have to perform Quantization and Dequantization using Cordic for further Compression of Image.
- We need to implement the same project with Distributed Arithmetic(DA) to compare Area, Power and Speed with DCTQ using OBC and see which one is best.
- Unfolding(parallel) and Pipelining DCTQ will be implemented such that all 8 elements in row computed at a time to increase throughput.

Chapter 6

References

- 1.Promod Kumar Meher,Thanos Stouraistis – Distributed Arithmetic and Offset Binary Coding.
- 2.Seetharaman Ramachandran - Digital VLSI Systems Design_ A Design Manual for Implementation of Projects on FPGAs and ASICs Using Verilog.

Bibliography

- [1] N. Ahmed, T. Natarajan and K.R. Rao, Discrete cosine transform, IEEE Trans. Comput., C-23, pp. 90–93, 1974.
- [2] Yung-Pin Lee, Thou-Ho Chen, Liang-Gee Chen, Mei-Juan Chen and Chung-Wei Ku, A cost-effective architecture for 8×8 2-D DCT/IDCT using direct method, IEEE Trans. Circuits Syst. Video Technol., 7, 1997.
- [3] Yukihiro ARAI, Takeshi AGUI and Masayuki NAKAJIMA, A fast DCT-SQ scheme for images, Trans. IEICE, E71, pp. 1095–1097, 1997.
- [4] Yi-Shin Tung, Chia-Chiang Ho and Ja-Lung Wu, MMX-based DCT and MC Algorithms for real-time pure software MPEG decoding, IEEE Computer Society Circuits and Systems, Signal Processing, 1, Florence, Italy, pp. 357–362, 1999.

Appendix A

Matlab code for image compression

```
=A=imread('1.jpg');
A=imresize(A,[128 128]);
subplot(1,2,1);
B=rgb2gray(A);
imshow(B)
title("original image");
C=double(B);
c=[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 -0.4517 -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];
cT=c';
cx=zeros(size(C));
q=[16 11 10 16 24 40 51 61
12 12 14 19 26 58 60 65
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 98 112 101 103 99];
for i=1:8:size(C,1)-7
```

```

for j=1:8:size(C,2)-7
    cx(i:i+7,j:j+7)=c*C(i:i+7,j:j+7);
end
end
cxcT=zeros(size(C));
for i=1:8:size(C,1)-7
    for j=1:8:size(C,2)-7
        cxcT(i:i+7,j:j+7)=cx(i:i+7,j:j+7)*cT;
    end
end
for i=1:8:size(C,1)-7
    for j=1:8:size(C,2)-7
        idct(i:i+7,j:j+7)=cT*cxcT(i:i+7,j:j+7);
    end
end
idct_ori=zeros(size(C));
for i=1:8:size(C,1)-7
    for j=1:8:size(C,2)-7
        idct_ori(i:i+7,j:j+7)=idct(i:i+7,j:j+7)*c;
    end
end
dup=uint8(idct_ori);
subplot(1,2,2);
imshow(uint8(idct_ori))
title("reconstructed image")

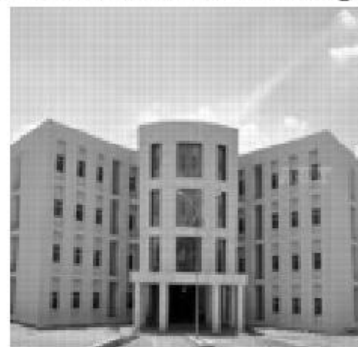
```

Matlab output:

original image



reconstructed image



Appendix B

Verilog code for Image Compression

B.1 Matlab code to generate text file for coe file

```
a=imread("1.jpg");
b=imresize(a,[128 128]);
c=rgb2gray(b);
d=[];
for i=1:8:128
for j=1:8:128
d=[d,c(i:i+7,j:j+7)];
end
end
v=d(:);
fid = fopen('rgukt.txt', 'w');
% Hex value write to the txt file
fprintf(fid, '%x',v);
% Close the txt file
fclose(fid);
```

B.2 Verilog code for OBC

```
module obc_new(x1,x2,x3,x4,x5,x6,x7,x8,a1,a2,a3,a4,a5,a6,a7,a8,clk,ACC,p,out);
input  clk,p;
```

```

input  [24:0] x1,x2,x3,x4,x5,x6,x7,x8;
input  [63:0]a1,a2,a3,a4,a5,a6,a7,a8;
output reg [63:0] ACC;
output reg out;
reg [31:0]i,c;
reg [63:0]lout;
reg [6:0]T;
always@(posedge clkp)
begin
ACC=(-(a1+a2+a3+a4+a5+a6+a7+a8));
ACC=(ACC[63],631'b0)---(ACC<1);
i=0;c=0;
end
always@(posedge clk p) begin
T[6]=(x1[i] ^ x2[i]);
T[5]=(x1[i] ^ x3[i]);
T[4]=(x1[i] ^ x4[i]);
T[3]=(x1[i] ^ x5[i]);
T[2]=(x1[i] ^ x6[i]);
T[1]=(x1[i] ^ x7[i]);
T[0]=(x1[i] ^ x8[i]);
i=i+1;
lutout(a1,a2,a3,a4,a5,a6,a7,a8,T,lout);
if(i=24)
begin
if(~(x1[i-1]))
begin
ACC=ACC+lout;
ACC=(ACC[63],631'b0)|| (ACC >> 1);
c = c + 1;
end
else
begin
ACC = ACC - lout;
ACC = (ACC[63], 631'b0)|(ACC >> 1);
end
end
elseif(i == 25)
begin
out = 1;
if((x1[i - 1]))

```

ACC = *ACC* + *lout*;

else

ACC = *ACC* - *lout*;

end

else

out = 0;

end

tasklutout;

input[63 : 0]*a1, a2, a3, a4, a5, a6, a7, a8*;

input[6 : 0]*B*;

outputreg[63 : 0]*lout*;

reg [63:0] *M1,M2 ,M3 ,M4 ,M5 ,M6 ,M7 ,M8 ,M9 ,M10 ,M11 ,M12 ,M13 ,M14 ,M15 ,M16 ,M17 ,M18 ,M19 ,M20 ,M21 ,M22 ,M23 ,M24 ,M25 ,M26 ,M27 ,M28 ,M29 ,M30 ,M31 ,M32 ,M33 ,M34 ,M35 ,M36 ,M37 ,M38 ,M39 ,M40 ,M41 ,M42 ,M43 ,M44 ,M45 ,M46 ,M47 ,M48 ,M49 ,M50 ,M51 ,M52 ,M53 ,M54 ,M55 ,M56 ,M57 ,M58 ,M59 ,M60 ,M61 ,M62 ,M63 ,M64 ,M65 ,M66 ,M67 ,M68 ,M69 ,M70 ,M71 ,M72 ,M73 ,M74 ,M75 ,M76 ,M77 ,M78 ,M79 ,M80 ,M81 ,M82 ,M83 ,M84 ,M85 ,M86 ,M87 ,M88 ,M89 ,M90 ,M91 ,M92 ,M93 ,M94 ,M95 ,M96 ,M97,M98,M99,M100,M101,M102,M103,M104,M105,M106 ,M107,M108,M109,M110,M111,M112,M113,M114,M115,M116,M117,M118,M119 ,M120,M121,M122,M123,M124,M125,M126,M127,M128*;

begin

M1 = $-(a1+a2+a3+a4+a5+a6+a7+a8)$);

M2 = $-(a1+a2+a3+a4+a5+a6+a7-a8)$);

M3 = $-(a1+a2+a3+a4+a5+a6-a7+a8)$);

M4 = $-(a1+a2+a3+a4+a5+a6-a7-a8)$);

M5 = $-(a1+a2+a3+a4+a5-a6+a7+a8)$);

M6 = $-(a1+a2+a3+a4+a5-a6+a7-a8)$);

M7 = $-(a1+a2+a3+a4+a5-a6-a7+a8)$);

M8 = $-(a1+a2+a3+a4+a5-a6-a7-a8)$);

M9 = $-(a1+a2+a3+a4-a5+a6+a7+a8)$);

M10 = $-(a1+a2+a3+a4-a5+a6+a7-a8)$);

M11 = $-(a1+a2+a3+a4-a5+a6-a7+a8)$);

M12 = $-(a1+a2+a3+a4-a5+a6-a7-a8)$);

M13 = $-(a1+a2+a3+a4-a5-a6+a7+a8)$);

M14 = $-(a1+a2+a3+a4-a5-a6+a7-a8)$);

M15 = $-(a1+a2+a3+a4-a5-a6-a7+a8)$);

M16 = $-(a1+a2+a3+a4-a5-a6-a7-a8)$);

M17 = $-(a1+a2+a3-a4+a5+a6+a7+a8)$);

M18 = $-(a_1+a_2+a_3-a_4+a_5+a_6+a_7-a_8)$);
M19 = $-(a_1+a_2+a_3-a_4+a_5+a_6-a_7+a_8)$);
M20 = $-(a_1+a_2+a_3-a_4+a_5+a_6-a_7-a_8)$);
M21 = $-(a_1+a_2+a_3-a_4+a_5-a_6+a_7+a_8)$);
M22 = $-(a_1+a_2+a_3-a_4+a_5-a_6+a_7-a_8)$);
M23 = $-(a_1+a_2+a_3-a_4+a_5-a_6-a_7+a_8)$);
M24 = $-(a_1+a_2+a_3-a_4+a_5-a_6-a_7-a_8)$);
M25 = $-(a_1+a_2+a_3-a_4-a_5+a_6+a_7+a_8)$);
M26 = $-(a_1+a_2+a_3-a_4-a_5+a_6+a_7-a_8)$);
M27 = $-(a_1+a_2+a_3-a_4-a_5+a_6-a_7+a_8)$);
M28 = $-(a_1+a_2+a_3-a_4-a_5+a_6-a_7-a_8)$);
M29 = $-(a_1+a_2+a_3-a_4-a_5-a_6+a_7+a_8)$);
M30 = $-(a_1+a_2+a_3-a_4-a_5-a_6+a_7-a_8)$);
M31 = $-(a_1+a_2+a_3-a_4-a_5-a_6-a_7+a_8)$);
M32 = $-(a_1+a_2+a_3-a_4-a_5-a_6-a_7-a_8)$);
M33 = $-(a_1+a_2-a_3+a_4+a_5+a_6+a_7+a_8)$);
M34 = $-(a_1+a_2-a_3+a_4+a_5+a_6+a_7-a_8)$);
M35 = $-(a_1+a_2-a_3+a_4+a_5+a_6-a_7+a_8)$);
M36 = $-(a_1+a_2-a_3+a_4+a_5+a_6-a_7-a_8)$);
M37 = $-(a_1+a_2-a_3+a_4+a_5-a_6+a_7+a_8)$);
M38 = $-(a_1+a_2-a_3+a_4+a_5-a_6+a_7-a_8)$);
M39 = $-(a_1+a_2-a_3+a_4+a_5-a_6-a_7+a_8)$);
M40 = $-(a_1+a_2-a_3+a_4+a_5-a_6-a_7-a_8)$);
M41 = $-(a_1+a_2-a_3+a_4-a_5+a_6+a_7+a_8)$);
M42 = $-(a_1+a_2-a_3+a_4-a_5+a_6+a_7-a_8)$);
M43 = $-(a_1+a_2-a_3+a_4-a_5+a_6-a_7+a_8)$);
M44 = $-(a_1+a_2-a_3+a_4-a_5+a_6-a_7-a_8)$);
M45 = $-(a_1+a_2-a_3+a_4-a_5-a_6+a_7+a_8)$);
M46 = $-(a_1+a_2-a_3+a_4-a_5-a_6+a_7-a_8)$);
M47 = $-(a_1+a_2-a_3+a_4-a_5-a_6-a_7+a_8)$);
M48 = $-(a_1+a_2-a_3+a_4-a_5-a_6-a_7-a_8)$);
M49 = $-(a_1+a_2-a_3-a_4+a_5+a_6+a_7+a_8)$);
M50 = $-(a_1+a_2-a_3-a_4+a_5+a_6+a_7-a_8)$);
M51 = $-(a_1+a_2-a_3-a_4+a_5+a_6-a_7+a_8)$);
M52 = $-(a_1+a_2-a_3-a_4+a_5+a_6-a_7-a_8)$);
M53 = $-(a_1+a_2-a_3-a_4+a_5-a_6+a_7+a_8)$);
M54 = $-(a_1+a_2-a_3-a_4+a_5-a_6+a_7-a_8)$);
M55 = $-(a_1+a_2-a_3-a_4+a_5-a_6-a_7+a_8)$);
M56 = $-(a_1+a_2-a_3-a_4+a_5-a_6-a_7-a_8)$);
M57 = $-(a_1+a_2-a_3-a_4-a_5+a_6+a_7+a_8)$);
M58 = $-(a_1+a_2-a_3-a_4-a_5+a_6+a_7-a_8)$);

M59 = $-(a_1+a_2-a_3-a_4-a_5+a_6-a_7+a_8)$;
M60 = $-(a_1+a_2-a_3-a_4-a_5+a_6-a_7-a_8)$;
M61 = $-(a_1+a_2-a_3-a_4-a_5-a_6+a_7+a_8)$;
M62 = $-(a_1+a_2-a_3-a_4-a_5-a_6+a_7-a_8)$;
M63 = $-(a_1+a_2-a_3-a_4-a_5-a_6-a_7+a_8)$;
M64 = $-(a_1+a_2-a_3-a_4-a_5-a_6-a_7-a_8)$;
M65 = $-(a_1-a_2+a_3+a_4+a_5+a_6+a_7+a_8)$;
M66 = $-(a_1-a_2+a_3+a_4+a_5+a_6+a_7-a_8)$;
M67 = $-(a_1-a_2+a_3+a_4+a_5+a_6-a_7+a_8)$;
M68 = $-(a_1-a_2+a_3+a_4+a_5+a_6-a_7-a_8)$;
M69 = $-(a_1-a_2+a_3+a_4+a_5+a_6+a_7+a_8)$;
M70 = $-(a_1-a_2+a_3+a_4+a_5-a_6+a_7-a_8)$;
M71 = $-(a_1-a_2+a_3+a_4+a_5-a_6-a_7+a_8)$;
M72 = $-(a_1-a_2+a_3+a_4+a_5-a_6-a_7-a_8)$;
M73 = $-(a_1-a_2+a_3+a_4-a_5+a_6+a_7+a_8)$;
M74 = $-(a_1-a_2+a_3+a_4-a_5+a_6+a_7-a_8)$;
M75 = $-(a_1-a_2+a_3+a_4-a_5+a_6-a_7+a_8)$;
M76 = $-(a_1-a_2+a_3+a_4-a_5+a_6-a_7-a_8)$;
M77 = $-(a_1-a_2+a_3+a_4-a_5-a_6+a_7+a_8)$;
M78 = $-(a_1-a_2+a_3+a_4-a_5-a_6+a_7-a_8)$;
M79 = $-(a_1-a_2+a_3+a_4-a_5-a_6-a_7+a_8)$;
M80 = $-(a_1-a_2+a_3+a_4-a_5-a_6-a_7-a_8)$;
M81 = $-(a_1-a_2+a_3-a_4+a_5+a_6+a_7+a_8)$;
M82 = $-(a_1-a_2+a_3-a_4+a_5+a_6+a_7-a_8)$;
M83 = $-(a_1-a_2+a_3-a_4+a_5+a_6-a_7+a_8)$;
M84 = $-(a_1-a_2+a_3-a_4+a_5+a_6-a_7-a_8)$;
M85 = $-(a_1-a_2+a_3-a_4+a_5-a_6+a_7+a_8)$;
M86 = $-(a_1-a_2+a_3-a_4+a_5-a_6+a_7-a_8)$;
M87 = $-(a_1-a_2+a_3-a_4+a_5-a_6-a_7+a_8)$;
M88 = $-(a_1-a_2+a_3-a_4+a_5-a_6+a_7-a_8)$;
M89 = $-(a_1-a_2+a_3-a_4-a_5+a_6+a_7+a_8)$;
M90 = $-(a_1-a_2+a_3-a_4-a_5+a_6+a_7-a_8)$;
M91 = $-(a_1-a_2+a_3-a_4-a_5+a_6-a_7+a_8)$;
M92 = $-(a_1-a_2+a_3-a_4-a_5+a_6-a_7-a_8)$;
M93 = $-(a_1-a_2+a_3-a_4-a_5-a_6+a_7+a_8)$;
M94 = $-(a_1-a_2+a_3-a_4-a_5-a_6+a_7-a_8)$;
M95 = $-(a_1-a_2+a_3-a_4-a_5-a_6-a_7+a_8)$;
M96 = $-(a_1-a_2+a_3-a_4-a_5-a_6-a_7-a_8)$;
M97 = $-(a_1-a_2-a_3+a_4+a_5+a_6+a_7+a_8)$;
M98 = $-(a_1-a_2-a_3+a_4+a_5+a_6+a_7-a_8)$;
M99 = $-(a_1-a_2-a_3+a_4+a_5+a_6-a_7+a_8)$;

```

M100= -(a1-a2-a3+a4+a5+a6-a7-a8));
M101= -(a1-a2-a3+a4+a5-a6+a7+a8));
M102= -(a1-a2-a3+a4+a5-a6+a7-a8));
M103= -(a1-a2-a3+a4+a5-a6-a7+a8));
M104= -(a1-a2-a3+a4+a5-a6-a7-a8));
M105= -(a1-a2-a3+a4-a5+a6+a7+a8));
M106= -(a1-a2-a3+a4-a5+a6+a7-a8));
M107= -(a1-a2-a3+a4-a5+a6-a7+a8));
M108= -(a1-a2-a3+a4-a5+a6-a7-a8));
M109= -(a1-a2-a3+a4-a5-a6+a7+a8));
M110= -(a1-a2-a3+a4-a5-a6+a7-a8));
M111= -(a1-a2-a3+a4-a5-a6-a7+a8));
M112= -(a1-a2-a3+a4-a5-a6-a7-a8));
M113= -(a1-a2-a3-a4+a5+a6+a7+a8));
M114= -(a1-a2-a3-a4+a5+a6+a7-a8));
M115= -(a1-a2-a3-a4+a5+a6-a7+a8));
M116= -(a1-a2-a3-a4+a5+a6-a7-a8));
M117= -(a1-a2-a3-a4+a5-a6+a7+a8));
M118= -(a1-a2-a3-a4+a5-a6+a7-a8));
M119= -(a1-a2-a3-a4+a5-a6-a7+a8));
M120= -(a1-a2-a3-a4+a5-a6-a7-a8));
M121= -(a1-a2-a3-a4-a5+a6+a7+a8));
M122= -(a1-a2-a3-a4-a5+a6+a7-a8));
M123= -(a1-a2-a3-a4-a5+a6-a7+a8));
M124= -(a1-a2-a3-a4-a5+a6-a7-a8));
M125= -(a1-a2-a3-a4-a5-a6+a7+a8));
M126= -(a1-a2-a3-a4-a5-a6+a7-a8));
M127= -(a1-a2-a3-a4-a5-a6-a7+a8));
M128= -(a1-a2-a3-a4-a5-a6-a7-a8));
case(B)
0 :lout={M1[63],{63{1'b0}}}|(M1 >>1);
1 :lout={M2[63],{63{1'b0}}}|(M2 >>1);
2 :lout={M3[63],{63{1'b0}}}|(M3 >>1);
3 :lout={M4[63],{63{1'b0}}}|(M4 >>1);
4 :lout={M5[63],{63{1'b0}}}|(M5 >>1);
5 :lout={M6[63],{63{1'b0}}}|(M6 >>1);
6 :lout={M7[63],{63{1'b0}}}|(M7 >>1);
7 :lout={M8[63],{63{1'b0}}}|(M8 >>1);
8 :lout={M9[63],{63{1'b0}}}|(M9 >>1);
9 :lout={M10[63],{63{1'b0}}}|(M10 >>1);
10 :lout={M11[63],{63{1'b0}}}|(M11 >>1);

```



```

11 :lout={M12[63],{63{1'b0}}}|(M12 >>1);
12 :lout={M13[63],{63{1'b0}}}|(M13 >>1);
13 :lout={M14[63],{63{1'b0}}}|(M14 >>1);
14 :lout={M15[63],{63{1'b0}}}|(M15 >>1);
15 :lout={M16[63],{63{1'b0}}}|(M16 >>1);
16 :lout={M17[63],{63{1'b0}}}|(M17 >>1);
17 :lout={M18[63],{63{1'b0}}}|(M18 >>1);
18 :lout={M19[63],{63{1'b0}}}|(M19 >>1);
19 :lout={M20[63],{63{1'b0}}}|(M20 >>1);
20 :lout={M21[63],{63{1'b0}}}|(M21 >>1);
21 :lout={M22[63],{63{1'b0}}}|(M22 >>1);
22 :lout={M23[63],{63{1'b0}}}|(M23 >>1);
23 :lout={M24[63],{63{1'b0}}}|(M24 >>1);
24 :lout={M25[63],{63{1'b0}}}|(M25 >>1);
25 :lout={M26[63],{63{1'b0}}}|(M26 >>1);
26 :lout={M27[63],{63{1'b0}}}|(M27 >>1);
27 :lout={M28[63],{63{1'b0}}}|(M28 >>1);
28 :lout={M29[63],{63{1'b0}}}|(M29 >>1);
29 :lout={M30[63],{63{1'b0}}}|(M30 >>1);
30 :lout={M31[63],{63{1'b0}}}|(M31 >>1);
31 :lout={M32[63],{63{1'b0}}}|(M32 >>1);
32 :lout={M33[63],{63{1'b0}}}|(M33 >>1);
33 :lout={M34[63],{63{1'b0}}}|(M34 >>1);
34 :lout={M35[63],{63{1'b0}}}|(M35 >>1);
35 :lout={M36[63],{63{1'b0}}}|(M36 >>1);
36 :lout={M37[63],{63{1'b0}}}|(M37 >>1);
37 :lout={M38[63],{63{1'b0}}}|(M38 >>1);
38 :lout={M39[63],{63{1'b0}}}|(M39 >>1);
39 :lout={M40[63],{63{1'b0}}}|(M40 >>1);
40 :lout={M41[63],{63{1'b0}}}|(M41 >>1);
41 :lout={M42[63],{63{1'b0}}}|(M42 >>1);
42 :lout={M43[63],{63{1'b0}}}|(M43 >>1);
43 :lout={M44[63],{63{1'b0}}}|(M44 >>1);
44 :lout={M45[63],{63{1'b0}}}|(M45 >>1);
45 :lout={M46[63],{63{1'b0}}}|(M46 >>1);
46 :lout={M47[63],{63{1'b0}}}|(M47 >>1);
47 :lout={M48[63],{63{1'b0}}}|(M48 >>1);
48 :lout={M49[63],{63{1'b0}}}|(M49 >>1);
49 :lout={M50[63],{63{1'b0}}}|(M50 >>1);
50 :lout={M51[63],{63{1'b0}}}|(M51 >>1);
51 :lout={M52[63],{63{1'b0}}}|(M52 >>1);

```

```

52 :lout={M53[63],{63{1'b0}}}|(M53 >>1);
53 :lout={M54[63],{63{1'b0}}}|(M54 >>1);
54 :lout={M55[63],{63{1'b0}}}|(M55 >>1);
55 :lout={M56[63],{63{1'b0}}}|(M56 >>1);
56 :lout={M57[63],{63{1'b0}}}|(M57 >>1);
57 :lout={M58[63],{631'b0}}}|(M58 >>1);
58 :lout={M59[63],{631'b0}}}|(M59 >>1);
59 :lout={M60[63],{631'b0}}}|(M60 >>1);
60 :lout={M61[63],{631'b0}}}|(M61 >>1);
61 :lout={M62[63],{631'b0}}}|(M62 >>1);
62 :lout={M63[63],{631'b0}}}|(M63 >>1);
63 :lout={M64[63],{631'b0}}}|(M64 >>1);
64 :lout={M65[63],{631'b0}}}|(M65 >>1);
65 :lout={M66[63],{631'b0}}}|(M66 >>1);
66 :lout={M67[63],{631'b0}}}|(M67 >>1);
67 :lout={M68[63],{631'b0}}}|(M68 >>1);
68 :lout={M69[63],{631'b0}}}|(M69 >>1);
69 :lout={M70[63],{631'b0}}}|(M70 >>1);
70 :lout={M71[63],{631'b0}}}|(M71 >>1);
71 :lout={M72[63],{631'b0}}}|(M72 >>1);
72 :lout={M73[63],{631'b0}}}|(M73 >>1);
73 :lout={M74[63],{631'b0}}}|(M74 >>1);
74 :lout={M75[63],{631'b0}}}|(M75 >>1);
75 :lout={M76[63],{631'b0}}}|(M76 >>1);
76 :lout={M77[63],{631'b0}}}|(M77 >>1);
77 :lout={M78[63],{631'b0}}}|(M78 >>1);
78 :lout={M79[63],{631'b0}}}|(M79 >>1);
79 :lout={M80[63],{63{1'b0}}}|(M80 >>1);
80 :lout={M81[63],{63{1'b0}}}|(M81 >>1);
81 :lout={M82[63],{63{1'b0}}}|(M82 >>1);
82 :lout={M83[63],{63{1'b0}}}|(M83 >>1);
83 :lout={M84[63],{63{1'b0}}}|(M84 >>1);
84 :lout={M85[63],{63{1'b0}}}|(M85 >>1);
85 :lout={M86[63],{63{1'b0}}}|(M86 >>1);
86 :lout={M87[63],{63{1'b0}}}|(M87 >>1);
87 :lout={M88[63],{63{1'b0}}}|(M88 >>1);
88 :lout={M89[63],{63{1'b0}}}|(M89 >>1);
89 :lout={M90[63],{63{1'b0}}}|(M90 >>1);
90 :lout={M91[63],{63{1'b0}}}|(M91 >>1);
91 :lout={M92[63],{63{1'b0}}}|(M92 >>1);
92 :lout={M93[63],{63{1'b0}}}|(M93 >>1);

```

```

93 :lout={M94[63],{63{1'b0}}}|(M94 >>1);
94 :lout={M95[63],{63{1'b0}}}|(M95 >>1);
95 :lout={M96[63],{63{1'b0}}}|(M96 >>1);
96 :lout={M97[63],{63{1'b0}}}|(M97 >>1);
97 :lout={M98[63],{63{1'b0}}}|(M98 >>1);
98 :lout={M99[63],{63{1'b0}}}|(M99 >>1);
99 :lout={M100[63],{63{1'b0}}}|(M100>>1);
100 :lout={M101[63],{63{1'b0}}}|(M101>>1);
101 :lout={M102[63],{63{1'b0}}}|(M102>>1);
102 :lout={M103[63],{63{1'b0}}}|(M103>>1);
103 :lout={M104[63],{63{1'b0}}}|(M104>>1);
104 :lout={M105[63],{63{1'b0}}}|(M105>>1);
105 :lout={M106[63],{63{1'b0}}}|(M106>>1);
106 :lout={M107[63],{63{1'b0}}}|(M107>>1);
107 :lout={M108[63],{63{1'b0}}}|(M108>>1);
108 :lout={M109[63],{63{1'b0}}}|(M109>>1);
109 :lout={M110[63],{63{1'b0}}}|(M110>>1);
110 :lout={M111[63],{63{1'b0}}}|(M111>>1);
111 :lout={M112[63],{63{1'b0}}}|(M112>>1);
112 :lout={M113[63],{63{1'b0}}}|(M113>>1);
113 :lout={M114[63],{63{1'b0}}}|(M114>>1);
114 :lout={M115[63],{63{1'b0}}}|(M115>>1);
115 :lout={M116[63],{63{1'b0}}}|(M116>>1);
116 :lout={M117[63],{63{1'b0}}}|(M117>>1);
117 :lout={M118[63],{63{1'b0}}}|(M118>>1);
118 :lout={M119[63],{63{1'b0}}}|(M119>>1);
119 :lout={M120[63],{63{1'b0}}}|(M120>>1);
120 :lout={M121[63],{63{1'b0}}}|(M121>>1);
121 :lout={M122[63],{63{1'b0}}}|(M122>>1);
122 :lout={M123[63],{63{1'b0}}}|(M123>>1);
123 :lout={M124[63],{63{1'b0}}}|(M124>>1);
124 :lout={M125[63],{63{1'b0}}}|(M125>>1);
125 :lout={M126[63],{63{1'b0}}}|(M126>>1);
126 :lout={M127[63],{63{1'b0}}}|(M127>>1);
127 :lout={M128[63],{63{1'b0}}}|(M128>>1);
endcase
end
endtask

```

B.3 Image Compression using DCT and IDCT code

```

module image_compression(input clk);
wire [24:0]c[7:0][7:0]; /// DCT coffecient matrix
reg [24:0]ct[7:0][7:0]; /// C transpose matrix
reg [14:0]addra; /// accessing address of image from the coe file
wire [11:0]x1; /// serial pixel from coe file
reg [63:0]x[127:0][127:0]; /// image in serial format stored to matrix
format
reg [63:0]cx[127:0][127:0]; /// CX matrix
reg [63:0]cxct[127:0][127:0]; /// CTCX matrix
reg [63:0]ctDCT[127:0][127:0];
reg [63:0]iDCT[127:0][127:0];
//////////block memory generator for input image matrix//////////
blk_mem_gen_0 your_instance_name (
.clka(clk), // input wire clka
.addra(addra), // input wire [14 : 0] addra
.douta(x1) // output wire [11 : 0] douta
);
//////////code for accessing serial input pixels into 128X128
matrix format//////////
reg [14:0]l,n,o;
reg flag;
reg [3:0]count;
initial
begin
addra=0;
l=0;
o=0;
n=0;
flag=1;
end
always @(posedge clk && flag)
begin
if(addra>=2) ///pixels start from address 2;
begin
x[o][l]=x1,24'h000000;
o=o+1;
if(o==(n+8))

```

```

begin
l=l+1; //coloum shift
o=n;
if(l==128)
begin
o=n+8;
l=0;
n=o;
if(n==128)
flag=0;
end
end
end
addra=addra+1;
count=(addra>=2) ? (count+1) :0;
end
//////////C matrix//////////
assign
c[0][0]=25'h5A8587,c[1][0]=25'h7D8ADA,c[2][0]=25'h763F14,c[3][0]=25'h6A6B50,
c[4][0]=25'h5A8587,c[5][0]=25'h471DE6,c[6][0]=25'h30F909,c[7][0]=25'h18F5C2,
c[0][1]=25'h5A8587,c[1][1]=25'h6A6B50,c[2][1]=25'h30F909,c[3][1]=25'h1E70A3E,
c[4][1]=25'h1A57A79,c[5][1]=25'h1827526,c[6][1]=25'h189C0EC,c[7][1]=25'h1B8E21A,
c[0][2]=25'h5A8587,c[1][2]=25'h471DE6,c[2][2]=25'h1CF06F7,c[3][2]=25'h1827526,
c[4][2]=25'h1A57A79,c[5][2]=25'h18F5C2,c[6][2]=25'h763F14,c[7][2]=25'h6A6B50,
c[0][3]=25'h5A8587,c[1][3]=25'h18F5C2,c[2][3]=25'h189C0EC,c[3][3]=25'h1B8E21A,
c[4][3]=25'h5A8587,c[5][3]=25'h6A6B50,c[6][3]=25'h1CF06F7,c[7][3]=25'h1827526,
c[0][4]=25'h5A8587,c[1][4]=25'h1E70A3E,c[2][4]=25'h189C0EC,c[3][4]=25'h471DE6,
c[4][4]=25'h5A8587,c[5][4]=25'h19594B0,c[6][4]=25'h1CF06F7,c[7][4]=25'h7D8ADA,
c[0][5]=25'h5A8587,c[1][5]=25'h1B8E21A,c[2][5]=25'h1CF06F7,c[3][5]=25'h7D8ADA,
c[4][5]=25'h1A57A79,c[5][5]=25'h1E70A3E,c[6][5]=25'h763F14,c[7][5]=25'h19594B0,
c[0][6]=25'h5A8587,c[1][6]=25'h19594B0,c[2][6]=25'h30F909,c[3][6]=25'h18F5C2,
c[4][6]=25'h1A57A79,c[5][6]=25'h7D8ADA,c[6][6]=25'h189C0EC,c[7][6]=25'h471DE6,
c[0][7]=25'h5A8587,c[1][7]=25'h1827526,c[2][7]=25'h763F14,c[3][7]=25'h19594B0,
c[4][7]=25'h5A8587,c[5][7]=25'h1B8E21A,c[6][7]=25'h30F909,c[7][7]=25'h1E70A3E;
//////////C transpose matrix//////////
integer row,column;
always@(*)
begin
for(row=0;row>8;row=row+1)
for(column=0;column<8;column=column+1)
ct[row][column]=c[column][row];

```

```

end
////////// required variable declaration for CX //////////
reg [4:0] obc_count;
reg [31:0] p_en;
reg obc_en,pp;
reg [2:0] it,j;
reg [3:0] sb,rb;
reg [24:0] C[7:0];
reg [63:0] X[7:0];
wire [63:0] CX1;
wire out;
integer i;
////////// required variable declaration for CXCT //////////
reg [4:0] obc_count1;
reg [31:0] p_en1;
reg obc_en1,pp1;
reg [2:0] it1,j1;
reg [3:0] sb1,rb1;
reg [24:0] CT[7:0];
reg [63:0] CX[7:0];
wire [63:0] CX2;
wire out1;
integer i1;
reg [5:0] count_8;
reg nxt_lvl;
////////// required variable declaration for CT*dct //////////
reg [4:0] obc_count2;
reg [31:0] p_en2;
reg obc_en2,pp2;
reg [2:0] it2,j2;
reg [3:0] sb2,rb2;
reg [24:0] iCT[7:0];
reg [63:0] iCX[7:0];
wire [63:0] CX3;
wire out2;
integer i2;
reg [5:0] count_57;
reg nxt_lvl1;
////////// required variable declaration for idct //////////
reg [4:0] obc_count3;
reg [31:0] p_en3;

```

```

reg  obc_en3,pp3;
reg  [2:0] it3,j3;
reg  [3:0]sb3,rb3;
reg  [24:0]iiCT[7:0];
reg  [63:0]iiCX[7:0];
wire  [63:0]CX4;
wire  out3;
integer i3;
reg  [5:0] count1_8;
reg  nxt_lvl2;
////////// file  variables//////////
integer file_id;
//////////
initial  file_id=$fopen("C:\\Users\\SUDHA\\Desktop\\dct.txt","w");
////////// Initialisations//////////
initial  begin  obc_count=1;it=0;p_en = 0;j = 0;sb = 0;rb = 0;
obc_count1 = 0;it1 = 0;p_en1 = 0;j1 = 0;sb1 = 0;rb1 = 0;count_8 =
0;nxt_lvl = 0;
obc_count2 = 0;it2 = 0;p_en2 = 0;j2 = 0;sb2 = 0;rb2 = 0;count_57 =
0;nxt_lvl1 = 0;
obc_count3 = 0;it3 = 0;p_en3 = 0;j3 = 0;sb3 = 0;rb3 = 0;count1_8 =
0;nxt_lvl2 = 0;end
//////////CX calculation//////////
always@(posedgeclk)
begin
if((count == 9)|| (obc_count == 0))
begin
for(i = 0;i <= 7;i = i + 1)
begin
C[i] = c[j][i];
X[i] = x[i + (rb * 8)][it + (8 * sb)];
end
obc_en = 1;
end
if(obc_en)
begin
if(p_en == 0)
pp = 1;
else
pp = 0;
p_en = p_en + 1;

```

```

obc_count = obc_count + 1;
if(obc_count == 27)
  begin
    obc_count = 0;
    obc_en = 0;
    p_en = 0;
    if(out)
      begin
        if(count_8 == 7)
          next_lvl = 1;
          count_8 = count_8 + 1;
          cx[j + (rb * 8)][it + (sb * 8)] = CX1;
          if(it == 7)
            begin
              if(j == 7)
                begin
                  if(sb == 15)
                    begin
                      rb = rb + 1;
                    end
                    sb = sb + 1;
                  end
                end
              j = j + 1;
            end
          it = it + 1;
        end
      end
    end
    //////////////////////////////////(CT)(CX)/////////////////////////////////
    if(next_lvl == 1)
      begin
        if(obc_count1 == 0)
          begin
            for(i1 = 0; i1 <= 7; i1 = i1 + 1)
              begin
                CT[i1] = ct[i1][it1];
                CX[i1] = cx[j1 + (rb1 * 8)][i1 + (sb1 * 8)];
              end
            obc_en1 = 1;
          end
        if(obc_en1)

```



```

begin
if(p_en1 == 0)
pp1 = 1;
else
pp1 = 0;
p_en1 = p_en1 + 1;
obc_count1 = obc_count1 + 1;
if(obc_count1 == 27)
begin
obc_count1 = 0;
obc_en1 = 0;
p_en1 = 0;
if(out1)
begin
if(count_57 == 57)
nxt_lvl1 = 1;
count_57 = count_57 + 1;
cxct[j1 + (rb1 * 8)][it1 + (sb1 * 8)] = CX2;
if(it1 == 7)
begin
if(j1 == 7)
begin
if(sb1 == 15)
begin
rb1 = rb1 + 1;
end
sb1 = sb1 + 1;
end
j1 = j1 + 1;
end
it1 = it1 + 1;
end
end
end
end
////////////////////////////////////
if(nxt_lvl1 == 1)
begin
if(obc_count2 == 0)
begin
for(i2 = 0; i2 <= 7; i2 = i2 + 1)

```

```

begin
iCT[i2] = ct[j2][i2];
iCX[i2] = cxct[i2 + (rb2 * 8)][it2 + (sb2 * 8)];
end
obc_en2 = 1;
end
if(obc_en2)
begin
if(p_en2 == 0)
pp2 = 1;
else
pp2 = 0;
p_en2 = p_en2 + 1;
obc_count2 = obc_count2 + 1;
if(obc_count2 == 27)
begin
obc_count2 = 0;
obc_en2 = 0;
p_en2 = 0;
if(out2)
begin
if(count1_8 == 7)
nxt_lvl2 = 1;
count1_8 = count1_8 + 1;
ctDCT[j2 + (rb2 * 8)][it2 + (sb2 * 8)] = CX3;
if(it2 == 7)
begin
if(j2 == 7)
begin
if(sb2 == 15)
begin
rb2 = rb2 + 1;
end
sb2 = sb2 + 1;
end
j2 = j2 + 1;
end
it2 = it2 + 1;
end
end
end

```

```

end
////////////////////////////////////
if(next_lvl2 == 1)
begin
if(obc_count3 == 0)
begin
for(i3 = 0; i3 <= 7; i3 = i3 + 1)
begin
iiCT[i3] = c[i3][it3];
iiCX[i3] = ctDCT[j3 + (rb3 * 8)][i3 + (sb3 * 8)];
end
obc_en3 = 1;
end
if(obc_en3)
begin
if(p_en3 == 0)
pp3 = 1;
else
pp3 = 0;
p_en3 = p_en3 + 1;
obc_count3 = obc_count3 + 1;
if(obc_count3 == 27)
begin
obc_count3 = 0;
obc_en3 = 0;
p_en3 = 0;
if(out3)
begin
iDCT[j3 + (rb3 * 8)][it3 + (sb3 * 8)] = CX4;
$fwrite(file_id, "/n", CX4[63 : 24]);
if(sb3 == 15 & rb3 == 15 & it3 == 7 & j3 == 7)
$fclose(file_id);
if(it3 == 7)
begin
if(j3 == 7)
begin
if(sb3 == 15)
begin
rb3 = rb3 + 1;
end
sb3 = sb3 + 1;

```

```

end
j3 = j3 + 1;
end
it3 = it3 + 1;
end
end
end
end
end
obc_new a1(C[0], C[1], C[2], C[3], C[4], C[5], C[6], C[7], X[0], X[1], X[2], X[3], X[4], X[5],
X[6], X[7], clk, CX1, pp, out);
obc_new a2(CT[0], CT[1], CT[2], CT[3], CT[4], CT[5], CT[6], CT[7], CX[0], CX[1], CX[2], CX[3],
CX[4], CX[5], CX[6], CX[7], clk, CX2, pp1, out1);
obc_new a3(iCT[0], iCT[1], iCT[2], iCT[3], iCT[4], iCT[5], iCT[6], iCT[7], iCX[0], iCX[1],
iCX[2], iCX[3], iCX[4], iCX[5], iCX[6], iCX[7], clk, CX3, pp2, out2);
obc_new a4(iiCT[0], iiCT[1], iiCT[2], iiCT[3], iiCT[4], iiCT[5], iiCT[6], iiCT[7], iiCX[0],
iiCX[1], iiCX[2], iiCX[3], iiCX[4], iiCX[5], iiCX[6], iiCX[7], clk, CX4, pp3, out3);
endmodule

```

B.4 Matlab code to convert verilog output text file to image

```

fileID=fopen('dct.txt','r');
A=fscanf(fileID,'%d');
fclose(fileID);
c=1;
d=[];
for l=0:8:120
for k=0:8:120
for j=1:1:8
for i=1:1:8
d(j+1,k+i)=(A(c));
c=c+1;
end
end
end
end
rc=uint8(d);

```

```
imshow(uint8((d)));
```

Matlab output:



Figure B.1: reconstructed image from verilog output file