REPUBLIC OF TURKEY
ISTANBUL GELISIM UNIVERSITY
INSTITUTE OF GRADUATE STUDIES

Department of Electrical and Electronics Engineering

FPGA IMPLEMENTATION OF DISCRETE COSINE TRANSFORM USING DIFFERENCE BASED ADDER GRAPH ALGORITHM

Master Thesis

Abdurrahman KNIFATI

Supervisor

Assoc. Prof. Dr. Indrit MYDERRIZI

Istanbul – 2021

THESIS INTRODUCTION FORM

NAME AND SURNAME OF THE AUTHOR	**:**	Abdurrahman KNIFATI
LANGUAGE OF THE THESIS	**:**	English
NAME OF THE THESIS	**:**	FPGA Implementation of Discrete Cosine Transform using Difference Based Adder Graph Algorithm
INSTITUTE	**:**	Istanbul Gelisim University Institute of Graduate Studies
DEPARTMENT	**:**	Electrical and Electronics Engineering
TYPE OF THE THESIS	**:**	Master
DATE OF THE THESIS	**:**	23.02.2021
PAGE NUMBER	**:**	65
THESIS SUPERVISORS	**:**	Assoc. Prof. Doc. Indrit MYDERRIZI
INDEX TERMS	**:**	2-D DCT, FPGA design, MCM algorithm, row-column decomposition, DiffAG, FREYR, DSP application, image compression, Matlab
TURKISH ABSTRACT	**:**	Günümüzde Dijital Sinyal İşleme (DSP), özellikle Alan Programlanabilir Geçit Dizilimi (FPGA) kullanarak devre uygulaması gibi donanım tasarımı ile ilgili konularda aktif bir araştırma alanıdır.
DISTRIBUTION LIST	**:**	1. To the Institute of Graduate Studies of Istanbul Gelisim University 2. To the National Thesis Center of YÖK (Higher Education Council)

Abdurrahman KNIFATI

REPUBLIC OF TURKEY
ISTANBUL GELISIM UNIVERSITY
INSTITUTE OF GRADUATE STUDIES

Department of Electrical and Electronics Engineering

FPGA IMPLEMENTATION OF DISCRETE COSINE TRANSFORM USING DIFFERENCE BASED ADDER GRAPH ALGORITHM

Master Thesis

Abdurrahman KNIFATI

Supervisor

Assoc. Prof. Dr. Indrit MYDERRIZI

Istanbul – 2021

DECLARATION

I hereby declare that in the preparation of this thesis / project, scientific ethical rules have been followed, the works of other persons have been referenced in accordance with the scientific norms if used, there is no falsification in the used data, any part of the thesis /project has not been submitted to this university or any other university as another thesis/project.

Abdurrahman KNIFATI

…/…/2021

TO ISTANBUL GELISIM UNIVERSITY

THE DIRECTORATE OF INSTITUTE OF GRADUATE STUDIES

The thesis of Abdurrahman KNIFATI titled as "FPGA Implementation of Discrete Cosine Transform using Difference Based Adder Graph Algorithm" has been accepted as MASTER THESIS in the department of ELECTRIC AND ELECTRONIC by our jury.

Signature

Director ______________________________

Assoc. Prof. Dr. Indrit MYDERRIZI

(Supervisor)

Signature

____

Member *Assoc. Prof. Dr. Hacı İLHAN*

Signature

Member ______________________________

Assist. Prof.Dr. AFM Shahen SHAH

APPROVAL

I approve that the signatures above signatures belong to the aforementioned faculty members.... / ... / 2021

Signature

Prof. Dr. İzzet GÜMÜŞ

Director of the Institute

ÖZET

Günümüzde Dijital Sinyal İşleme (DSP), özellikle Alan Programlanabilir Geçit Dizilimi (FPGA) kullanarak devre uygulaması gibi donanım tasarımı ile ilgili konularda aktif bir araştırma alanıdır. Buna ek olarak, çarpıcılar DSP uygulamalarında çok fazla kaynak tükettiği için araştırmacılar, Çoklu Sabit Çarpma (MCM) olarak da bilinen shift / add ağ tasarımını kullanarak çarpmaları uygulamaktadırlar. Hcub, FRYER ve DiffAG dahil olmak üzere MCM problemini verimli bir şekilde çözen birkaç algoritma vardır. Bu çalışmada DiffAG algoritması Matlab kullanılarak geliştirilmiştir. Ardından, MCM çözümünü kullanarak FPGA üzerinde Ayrık Kosinüs Dönüşümü (DCT) tasarımı uygulanır. DCT tasarımı, satır-sütun ayrıştırma yöntemi kullanılarak geliştirilmiştir. Karşılaştırmak için, FREYR'ın MCM çözümü kullanılarak başka bir tasarım geliştirilmiştir. Sonuçlar, DiffAG algoritmasından elde edilen MCM çözümlerini kullanan DCT tasarımının% 18,2 oranında daha fazla işlem hızı elde ettiğini göstermektedir. Bununla birlikte, FREYR'in MCM çözümü kullanılarak elde edilen sonuçlar,% 9,2 oranında daha az alan kullanarak DCT'ye ulaşmaktadır.

Anahtar Kelimeler: 2-D DCT, FPGA tasarımı, MCM algoritması, satır-sütun ayrıştırma, DiffAG, FREYR, DSP uygulaması, görüntü sıkıştırma, Matlab.

SUMMARY

Nowadays, Digital Signal Processing (DSP) is an active research field especially topics related to hardware design such as circuit implementation using Field Programmable Gate Arrays (FPGA). In addition, since multipliers take up a lot of resources in DSP applications, researchers have been implementing multiplications using shift/add network design, also known as Multiple Constant Multiplication (MCM). There are several algorithms that solve the MCM problem efficiently, including Hcub, FRYER and DiffAG. In this work, the DiffAG algorithm has been developed using Matlab. Then, using its MCM solution, a Discrete Cosine Transform (DCT) design is implemented using FPGA. The DCT design is developed using the row-column decomposition method. In order to compare, another design is developed using FREYR's MCM solution. The results suggest that the DCT design using MCM solutions obtained from DiffAG algorithm achieve more processing speed by 18.2%. However, results obtained using FREYR's MCM solution achieve the DCT using less area by 9.2%.

Key Words: 2-D DCT, FPGA design, MCM algorithm, row-column decomposition, DiffAG, FREYR, DSP application, image compression, Matlab.

TABLE OF CONTENTS

ABBREVIATIONS

2D	**:**	Two-Dimension
ASIC	**:**	Application-Specific Integrated Circuit
CLB	**:**	Configurable Logic Block
CSD	**:**	Canonic Signed Digit
DCT	**:**	Discrete Cosine Transform
DiffAG	**:**	Difference-Based Adder Graph
DSP	**:**	Digital Signal Processing
DST	**:**	Discrete Sine Transform
FFT	**:**	Fast Fourier Transform
FIR	**:**	Finite Impulse Response
FPGA	**:**	Field Programmable Gate Array
HDL	**:**	Hardware Description Language
LUT	**:**	Look-up Table
MCM	**:**	Multiple Constant Multiplication
PSNR	**:**	Peak Signal-to-Noise Ratio
RAM	**:**	Random-Access Memory
RTL	**:**	Register-Transfer Level
TU	**:**	Transform Unit

LIST OF TABLES

LIST OF FIGURES

PREFACE

This work was done by the support of many people. I thank my advisor Mr. Indrit MYDERRIZI for his support. İn addition, I thank the support of my family and my friends who were with me in this journey.

CHAPTER ONE

INTRODUCTION

1. PROLOGUE

1.1. Motivation

In this technologically advancing world, communications and data processing are ever expanding and developing at an accelerating rate. Digital Signal Processing (DSP) has become a crucial section in all aspects. That includes industry, health, information, security, transportation, safety, etc. To develop technology and communications towards a better future, it is necessary that scientists innovate and invent methods that increase the rate of data processing while at the same time reduce power consumption and data storage allocation. DSP is one of the most vital fields that research is focusing on. That field is expanding rapidly and with it there are many opportunities to find improvements and optimizations of older technologies. The field of image compression takes place in a wide variety of applications that could use some improvement.

A well-known reliable method for image compression uses Discrete Cosine Transform (DCT). The benefits of transforming the data of images using the discrete cosine transform are several. First, it reduces storage without distorting the images much. Second, it improves information processing rate later on, since less time and power are needed, then.

Field Programmable Gate Arrays (FPGAs) are a common type of chip that is considered a superior solution to generic processors and Application-Specific Integrated Circuits (ASICs) in terms of capabilities in image processing while having design flexibility to reach an optimal solution. FPGAs are considered the best option when it comes to designing new circuits for image processing purposes in general.

Recently, researchers developed algorithms that implement constant multiplications in terms of additions and shift operations. Such methods, via the use of FPGAs, proved superior to generic multipliers in terms of speed, power and silicon area needed, especially in large scale image, video or audio processing. Thus, this work is motivated by making an algorithm that implements image compression using discrete cosine transform built on an FPGA board.

1.2. Organization of this work

Briefly, this is how this thesis is organized, explaining the contributions made to the research field.

In section II, necessary topics are covered as background information. A brief explanation of FPGAs and their features are given. In addition, reasons why FPGAs are suitable for image compression are provided. Next, an understanding of image compression is presented. Furthermore, to implement image compression, it is essential to understand its method as well as the DCT that is used to compress the images. After that, a brief explanation of Multiple Constant Multiplication (MCM) problem is given. It is important for the application of shift/add solutions. Consequently, MCM problems are typically solved by algorithms such as FREYR or DiffAG. Such algorithms are many and rely on different concepts. More details are provided regarding that topic as needed.

In section III, one of the algorithms for solving MCM problems is the Difference Based Adder Graph (DiffAG) algorithm. It is extensively explained since it is considered the heart of the project. The algorithm is provided in a concise form as well. In addition, reasons why this algorithm is picked, how it is developed on Matlab are stated. Then, results obtained by DiffAG are compared to a recent algorithm called FREYR in terms of number of additions and adder depth. Thus, some notes are drawn from the comparison.

In section IV, a DCT code developed using Matlab is presented. The code relies on MCM solutions from the DiffAG algorithm. Then, a simple code is developed for image compression that uses the DCT algorithm. This is the first step to simulate the results as MCM blocks inside a DCT algorithm in image processing. Consequently, a brief illustration of that step is given.

In section V, simulations are developed in preparation for hardware implementation. A Verilog code is written to implement the image compression model. It gets simulated similar to the algorithm previously used in Matlab.

In section VI, once everything is simulated, the code is then turned into a circuit on the FPGA chip. Thus, results and comments are drawn in comparison of other previous works in that field. At the end, sections VII and VIII state the conclusion and tell what the future work will be on.

CHAPTER TWO

BACKGROUND

2. RELATED WORK

2.1. Field Programmable Gate Arrays

A Field Programmable Gate Array (FPGA) is a device that allows the design of digital circuits using a hardware description language (HDL). There are several HDLs such as VHDL or Verilog. Digital design using FPGAs provides the flexibility in design and cuts costs down since reprogramming is easy and fast. FPGAs consist of Configurable Logic Blocks (CLBs), interconnections and I/O pads. Logic blocks are used to make logic gates via reconfiguration. There are look-up tables (LUTs) that are useful for combinational logic implementation. LUTs can sometimes be called RAM units. There are input ports and output ports for entering and extracting data. Via programming the interconnections, it is possible to connect two logic networks throughout the FPGA or to interconnect several CLBs for making a logic design. Figure 1 is a simplistic illustration of the FPGA architecture. It shows logic blocks, I/O cells and the interconnections between them (Patil, 2010).

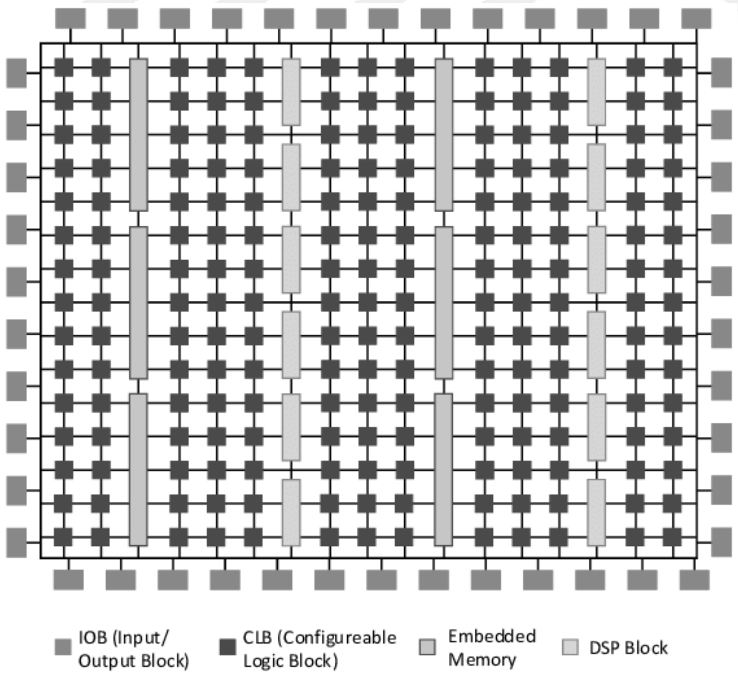


Figure 1: Illustration of FPGA's architecture
Source: (Ronak & Fahmy, 2016).

Details of the configurable logic blocks are presented. They consist of D flip-flops, a look-up table and a multiplexer. Logic functions are implemented via look-up tables. Figure 2 illustrates a 4-input look-up table of one output. The output is a result from the combination of several signals. D flip-flops are memory elements and multiplexers are for selecting which data to forward to the output (Patil, 2010).

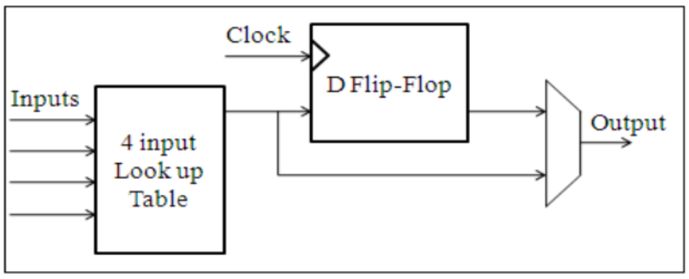


Figure 2: Illustration of a logic block.
Source: (Patil, 2010).

FPGAs are invented based on the idea that logic blocks surrounded by I/O blocks allows for assembling the whole design in various arrangements. Modern FPGAs have high capabilities such as high operation speed, large number of cells, supported protocols, etc. Via the use of FPGAs, it is easy to design custom hardware such as custom accelerators that get embedded in other hardware to implement time consuming computations (Paulitsch et al., 2011). In addition, FPGAs are very useful when it comes to replacing a failed old digital component, such as legacy system components in an old industrial control system (Anghel et al., 2003; Bomar, 2002; Rodriguez-Andina et al., 2007).

2.2. Image Compression

Image compression is still an important application in Digital Signal Processing. Minimizing storage and data transmission, it is necessary to increase the effectiveness of image compression. This reduces bandwidth as well as reduces costs for storage in various applications. Image compression applications span widely from TV, mobile, all the way to professional photography and video recording. Thus, part of research is still interested in developing algorithms and tools to enhance low bit image encoding (Stephen A. Martucci, 1996). Essentially, an image is a signal of two dimensions (2D) which get perceived by the human system. A large proportion of the imagery data is represented in several image types, especially data in the field of biomedical, remote sensing and large-screen digital imagery (Woods, 2008).

Image compression aims to decrease the data used to represent an image. Compression is a process that compacts the image data in order to store or transmit it.

This is done by ridding of data redundancy such as coding, psychovisual, or interpixel redundancies. There are two types of image compression: lossless and lossy image compression. Lossless image compression techniques allow the image to be perfectly recovered. It relies on the idea of entropy coding. It uses decomposition to minimize redundancies. This type of compression is relatively ineffective since compression ratios are low compared to lossy compression. However, it is needed to preserve the quality of the image as is. This is important in medical imaging where quality of the image matters. However, the more common and effective type of image compression is the lossy technique. It provides high ratio of compression, however, the original image is not completely recovered. There are several types of lossy compression depending on the quality of the retrieved image and the application in use (Vrindavanam, 2012).

To compress an image there are two requirements: there should be a significant correlation among nearby pixels within a single image. This correlation is known as spatial correlation. Secondly, for acquired data from several images, there should be a significant correlation among pixels of those images. This correlation is known as spectral correlation. Thus, to compress an image, this correlation must be taken advantage of. By removing most of the correlated pixels before compression, this increases effectivity of the compression in later steps (Vrindavanam, 2012).

Image compression process can be broken down into several steps, see figure 3. At first the image data are reduced. This happens by turning the image into grey level. If necessary, the image could be reduced by spatial quantization or noise removal. Next, the mapping process takes place where image data gets transformed into another mathematical spectrum to make the compression easier. Furthermore, during the encoding process, the quantization step turns the data into a discrete form. Finally, the last step is symbol coding the data for transmission or storage.

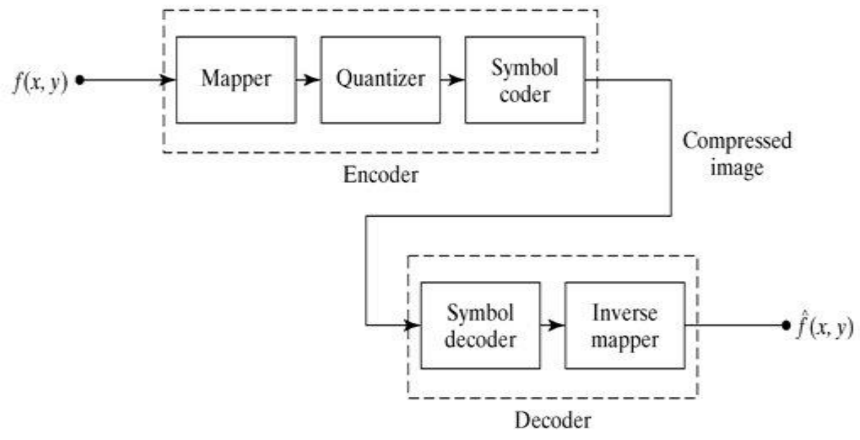


Figure 3: Stages of image compression.

Source: (Tresa & Sundararajan, 2013)

The most important step is the mapping step since data usually are highly correlated. Generally, the data in a certain pixel is quite similar to the neighboring one. To decorrelate data, it is useful to find an appropriate equation that find the redundancy and remove it (Hussain et al., 2018).

2.3. Discrete Cosine Transform:

Discrete cosine transform is an effective coding scheme that is well known and widely used. It is a transformation that is separable, real and orthogonal. It transforms the data from spatial domain into the frequency domain in a more compact representation. DCT is considered the basis of several image compression standards.

This formula is used commonly for image processing applications. It is a two dimensional DCT of a square block of pixels, f(x, y) can be defined as:

$$C(u,v) = \alpha(u)\alpha(v)\sum_{x=0}^{N-1}\sum_{x=0}^{N-1} f(x,y)cos\left(\frac{(2x+1)u\pi}{2N}\right)cos\left(\frac{(2y+1)v\pi}{2N}\right) - - - - - (1)$$

Where, f(x, y) is a two dimensional sequence of N x N points, C(u, v) denotes N x N points DCT of the block f(x, y) and x, y = 0,1,…,N-1; u, v = 0,1,…,N-1.

The zero frequency coefficient (C (0, 0)) of the DCT matrix is known as the DC coefficient. The rest of the coefficients are called AC coefficients. They show the amount of variations using the grey level at a particular rate and in a particular direction.

Typically, the image information is concentrated in the DC component and in the low AC frequency data. The data now are ordered from low frequency to high frequency in a zigzag manner. See figure 4 below.

These frequencies are placed in the top left corner of the image whereas the high frequency data are set in the right bottom corner. Thus, it is best to condense the data by keeping the low frequency coefficients while removing the higher ones. This step is called quantization step. It scales the values in the image down depending on the frequency, high components get reduced to almost zero while low components stay large. This is where the high frequencies get removed to free space. And since Huffman coding stores data in a zigzag manner, only the very few beginning values are non-zeros while the rest become zeros. A series of zeros is easily coded using Huffman and does not take up much storage.

DCT has the property of decorrelation for most images. The fundamental vectors that represent the original image block are independent. This property can be seen in the Figure 5 (Hussain et al., 2018).

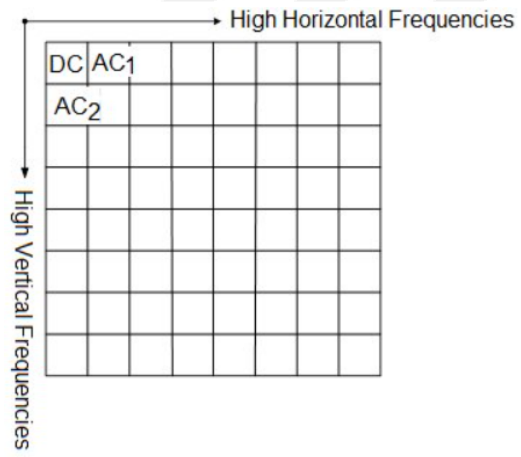


Figure 4: DCT values are ordered by frequency

Source: (Hussain et al., 2018).

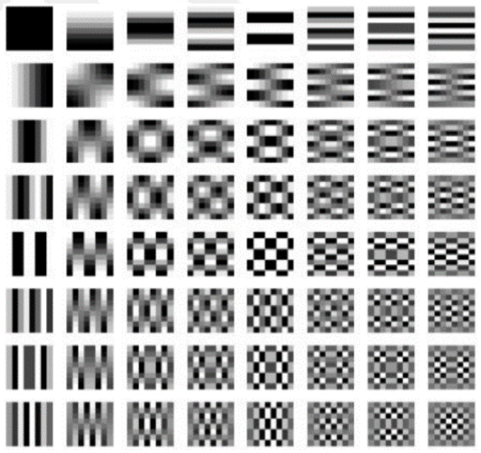


Figure 5: Illustration of DCT components and the property of energy compaction.

DCT possesses the property of energy compaction when dealing with highly correlated data. This is why DCT is a widely used transform for image processing. However, DCT has a disadvantage that appear during high image compression. The decompressed image shows tiling artifacts due to the fact that DCT is implemented in blocks rather than for the complete image (Garrido et al., 2018). Below is an illustration of the effect of tiling, see figure 6.

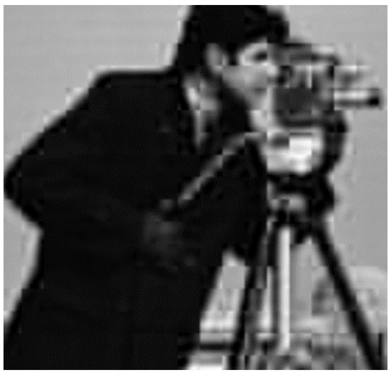


Figure 6: Example of tiling in image compression in DCT.

2.4. Related DCT Implementations

2.4.1. An Efficient Low Area Implementation of 2D DCT on FPGA

The discrete cosine transform has several applications such as MPEG in multimedia, JPEG in image compression and H.261 in teleconferencing. Typically, to compress an image using JPEG, the image data is divided into 8x8 blocks. Each block gets shifted down from unsigned integer range to signed integer range. Then, the 2D DCT is applied for each block.

One of the common effective methods for 2D DCT is called the row-column decomposition method (Agostini et al., 2007; Kusuma & Widodo, 2010; Pradeepthi & Ramesh, 2011). This method has several advantages. First, since this method has the separable property, the 2D DCT can be implemented into two 1D DCT parts, row-wise first then column-wise or vice versa. Also, this method is quite regular and modular which makes the control logic needed for implementation less complex. Furthermore, this method of 2D DCT requires less computations because of the symmetry inherent in the DCT coefficients. Thus, the computational complexity is decreased by a factor of 4 (Doğan, 2016).

2.4.2. The High Efficiency Video Coding standard

A recent standard for video compression called High Efficiency Video Coding (HEVC) has been developed (ITU-T, 2019; Pourazad et al., 2012). Its efficiency achieved 50% more video compression than the previous standard known as H.264. Both standards rely on the discrete cosine transform and its inverse to implement the video compression. However, the H.264 standard has only sizes of 4x4 and 8x8 Transform Units (TU) for DCT while the HEVC extended the sizes to include 16x16 and 32x32 TU sizes. Using TU sizes of 16x16 or 32x32 for DCT increases energy compaction. The catch is, the computational complexity increases exponentially. Also, the HEVC standard involves the Discrete Sine Transform (DST) and its inverse in some cases.

HEVC encoders rely heavily on transformations such as DCT, DST and their inverses (Vanne et al., 2012). Those transforms are computationally complex. They comprise about 11% of the computations in the HEVC encoder.

A new technique for HEVC DCT is developed. It is made for all sizes of transform units and it reduces energy consumption. Once the forward transform and the quantization stages are done, the high frequency coefficients become zero in each TU. Furthermore, low frequency coefficients of low value possess small effect on the later stages. Thus, in this technique, only the more significant values of low frequency are kept while the rest are assumed to be zero.

The method used includes mode decision as well as the coding stage for HEVC encoder design. Note that DCT coefficients must be the same in encoding and decoding stages for HEVC in order to avoid mismatched results. The HEVC decoder does not require any modifications in this method. Consequently, using such method decreases complexity of the DCT significantly while the results have less Peak Signal-to-Noise Ratio (PSNR) and more bit rate. The HEVC HM software encoder (Shen et al., 2013) provided reduction in execution time up to 12.74%. Also, the DCT operations provided a reduction of 37.27% in execution time within the HEVC HM encoding stage.

An implementation of low power HEVC 2D DCT technique has been developed for hardware (Kalali et al., 2016). The design is made for all TU sizes using Verilog HDL. This design can compute 4, 8, 16 and 32 coefficients per cycle for the TU sizes 4x4, 8x8, 16x16 and 32x32 sizes, respectively. This design at least implement 48 frames per second having a specification of quad full HD (3840x2160). Moreover, another hardware design of low power HEVC 2D DCT is made but has higher utilization. The difference is that this design can compute two 8x8 TUs or four 4x4 TUs in parallel. Thus, it computes 16 DCT coefficients per cycle for TU sizes of 4x4, 8x8 and 16x16, and 32 DCT coefficients for 32x32 TU size in each cycle. This at least provides a video processing of 53 frames per second of specification Ultra HD (7680x4320).

To minimize power consumption, clock gating is exploited for the designs. Also, to optimize the design, Hcub is used to find the MCM solution reducing the number and size of adders. The Hcub provides a reduction in power consumption in both lower utilization design and higher utilization design reaching up to 5.9% and 13.1%, respectively. Furthermore, the method used reduces power consumption in both designs. The lower utilization and higher utilization designs both recorded a reduction in power by 17.9% and 18.9, respectively (Kalali et al., 2016).

2.4.3. Efficient MCM Using DSP Blocks in FPGA

In modern FPGAs, there are built-in blocks that compute specific tasks efficiently without taking up much area. Such blocks reduce the need for LUTs and registers. Design implementation using such blocks improves power consumption and executes operation faster than design made onto a network of LUTs. Such built-in blocks perform various tasks in several applications. In modern FPGAs, DSP blocks are used in several applications involving image, video and audio processing since they have customizable multiplier blocks (Azgin et al., 2019).

DSP multiplier blocks execute the MCM operation faster and more efficiently than an equivalent network of shifter and adders. If properly used, a single DSP block can compute several constant multiplications given suitable value assigned to it as a constant.

Using DSP blocks, an efficient method is used for implementing the MCM operation on FPGAs. This method decreases the number of DSP multipliers needed for computing MCM block via handling the constants in that block.

The design is implemented using FPGAs. The design is realized by 2D DCT in the HEVC algorithm. The efficiency of the implementation using DSP blocks compared to shift/add MCM blocks is 35.8% (Azgin et al., 2019).

2.4.4. HEVC 2D DCT

Within an HEVC, the 2D DCT is computed in Transform Units. HEVC has TU sizes of 4x4, 8x8, 16x16 and 32x32 pixels. As is the norm, the 2D DCT is computed via applying the 1D DCT vertically then horizontally. The DCT matrices are calculated using the basis DCT equations. Nevertheless, it is simpler to deal with integer values for hardware implementation.

All TU sizes of HEVC 2D DCT are prepared and implemented. For comparison, three cases are considered. First, the constant multiplications are developed using DSP blocks, tying a single DSP block for each multiplication (baseline). The second design (concatenate) relies on the method presented in (Fu et al., 2017). In the third case, a technique is used for minimizing the number of DSP blocks used by mapping constants to them.

Comparing the three techniques, the third design (proposed) uses less DSP multipliers than the baseline and the concatenate designs by 35.8% and 13%, respectively (Mert et al., 2018).

2.5. Multiple Constant Multiplication Algorithms

Throughout the last two decades, the Multiple Constant Multiplication (MCM) has been attracting a lot of attention in the field of digital signal processing (DSP) area (Aksoy et al., 2011, 2014; Chandra & Chattopadhyay, 2016; Dempster & Macleod, 1995; Gately et al., 2012; Gustafsson, 2007; Johansson et al., 2011; Kumm & Zipf, 2012; Thong & Nicolici, 2010; Voronenko & Püschel, 2007). Most DSP applications such as audio, image, and video processing require multiplications. Multipliers are costly in terms of hardware. An MCM block works as follows: via a set of coefficients known beforehand, the multiplication process gets realized as a network of additions, subtractions and shifts of the input. The MCM problem aims to find the least required number of additions/subtractions to realize the MCM block (Aksoy et al., 2014). This process is sometimes called the multiplierless implementation and it is a bit-parallel arithmetic.

It is well known that this implementation assumes additions and subtractions to cost the same in term of hardware. Shifts are considered free since they can be hardwired into the circuit. The MCM problem finds common coefficients among the required constants so that an implementation is realized using the least number of adders/subtractors. Also since shifts are considered free, all even numbers can be realized by shifting other odd numbers. Moreover, although the MCM typically deals with positive coefficients, it is assumed that negative coefficients are simply found by switching the addition into subtraction in the previous step or vice versa. These assumptions are used in most of the literature (Gustafsson, 2007; Thong & Nicolici, 2010).

The problem of MCM has several applications such as Finite Impulse Response (FIR) filters, Fast Fourier Transform (FFT), Discrete Cosine Transform (DCT), etc. Much research has been done using the FIR filters to validate and compare the results of MCM blocks generated by the various algorithms (Aksoy et al., 2014; Chandra & Chattopadhyay, 2016). Furthermore, FPGAs were successfully used to implement MCM designs. This is due to the fact that FPGAs are programmable and can be simply redesigned with ease (Aksoy et al., 2014).

It is known that to guarantee an optimal solution of an MCM problem, the algorithm used must be an exhaustive search algorithm. Such algorithms are proven to be NP-complete (Thong & Nicolici, 2010). However, many researchers have developed more efficient algorithms to solve the MCM problem in several ways without the need to deal with such complexity (Aksoy et al., 2014).

Adder graph algorithms are one of the best algorithms throughout the literature. They rely on two parts. The first part tries to realize the solution as shifts and additions/subtractions of previously realized coefficients. When all coefficients get realized, the result is optimal. Otherwise, if there are no coefficients realizable using previously realized coefficients, a heuristic finds which additional "useful" coefficient needed to allow for the optimal part to carry on. However, having the heuristic part means that the solution is no longer guaranteed to be optimal (Gustafsson, 2007).

Recently, one of the better adder graph algorithms is called FREYR (Aksoy et al., 2014). Repetitively, the algorithm solves the MCM problem using an efficient graph based algorithm (Voronenko & Püschel, 2007). Using that solution, FREYR chooses which multiplications to get realized using at most a certain number of generic multipliers. This results in a minimum number of additions/subtractions required in the MCM block. FREYR is considered one of the best algorithms in terms of delay, power consumption and area, compared to the state-of-art algorithms (Aksoy et al., 2014; Chandra & Chattopadhyay, 2016).

Nevertheless, there is also the difference based adder graph algorithm (DiffAG) (Gustafsson, 2007). Similar to adder graph algorithms, the difference algorithm relies on the optimal part, realizing a required coefficient using adds/shifts of previously realized coefficients. However, it uses a different heuristic part to pick its new required coefficient. To explain, the difference algorithm computes all differences between any pair of coefficients. Then, it finds the difference that has the minimum realization cost amongst those differences. That difference enables the optimal part to realize more required coefficients using the newly added ones. The difference heuristic provides results better than RAG-n in terms of adder cost while it provides solutions better than Hcub in terms of average adder cost even though some specific solutions could be less ideal in some instances (Gustafsson, 2007).

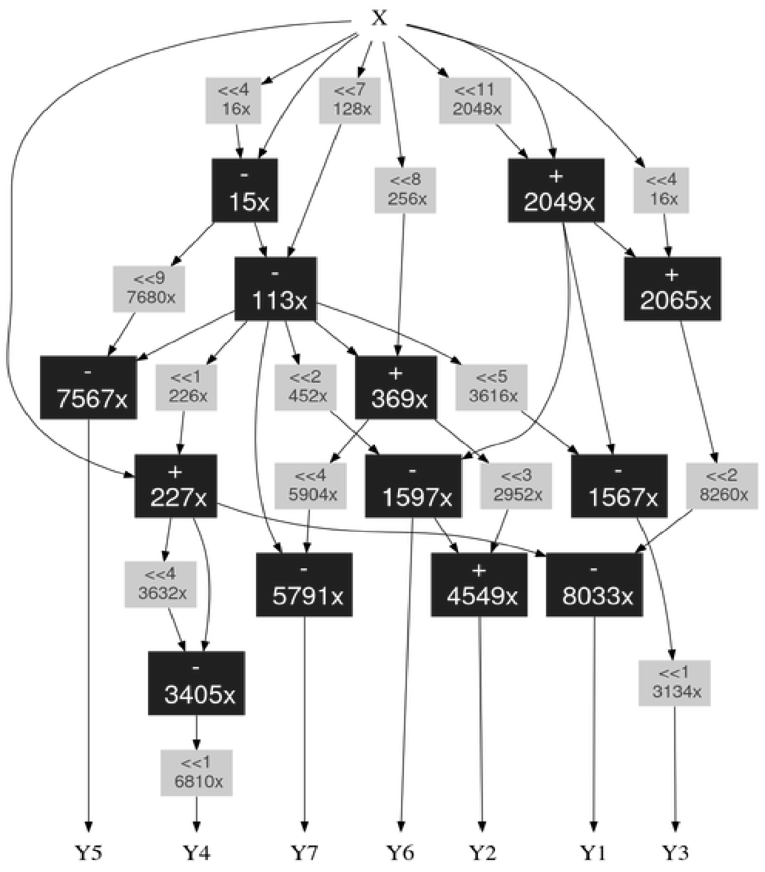


Figure 7: An example of MCM solution using HCUB algorithm
Source: (Voronenko, n.d.).

CHAPTER THREE

THE DIFFERENCE BASED ADDER GRAPH

3. THE DIFFERENCE BASED ADDER GRAPH

3.1. Steps of the algorithm:

The detailed algorithm has been detailed in (Gustafsson, 2007). However, for the purpose of implementing the algorithm using Matlab, some important parts are emphasized.

1) Create a matrix for the realized coefficients. Each row has a coefficient along with how it was realized. Also, include an extra column for keeping the (cost/adder depth) of realized coefficients. At first, store the realized coefficient 1. For example, the coefficient 1 can be realized using 1 = $2^0(1) + 2^0(0)$. The first row of realized matrix is (1, 1, 1, 0, 0). Moreover, create the required coefficients vector.

2) Now, if there are no required coefficients the code terminates. Otherwise, make all possible realizable values using an adder/subtractor with inputs from realized coefficients, i.e.

   $C = \left|2^i v_1 \pm 2^j v_j\right| ------(2)$

   v1, v2 are realized coefficients, i, j, are integers inferring that the realized values can be shifted in both directions left and right.

Note that when a number gets realized, its components – i.e. how it was realized – and its cost/adder depth should also be stored. It is important to store the realizable list of coefficients, their components and costs/adder depths. This is crucial when choose the most “useful” difference in next steps (step 7 or in step 8), since all of these components are ready to be used.

3) Whenever a required coefficient gets realized, return to step 2. This is because the realized matrix has been expanded, meaning it is possible to realize other required coefficients using newly realized ones or their shifted values. Note that sometimes this could result in having higher adder depth solutions depending on what coefficients get realized first.

Those three steps are known as the optimal part of RAG-n and Hcub. If, using only that part, all the required coefficients got realized, then, the solution is optimal. There is one adder for each unique odd integer coefficient.

Otherwise, the difference based adder graph heuristic takes place to choose an additional value in order to realize the required coefficients. To explain, the new value gets used as a realized value to build upon and eventually realize the required set using the same equation in step 2.

4) Create a node for each coefficient. Also, create a single node that has all of the realized coefficients. Connect all nodes with each other such that there is a single line between any two nodes. Calculate all realizable values, known as differences, using the c_equation with inputs taken from any two nodes. Assign those differences to the line connecting those two nodes. Note that if one of the differences is already realized, one of the nodes becomes realizable whenever the other one has been realized. For the realized node, all realized coefficients must be considered.

This heuristic helps choose which extra coefficient to get realized since it is assured to be used in the coming iterations realizing the required coefficients.

5) If there were a realized coefficient on an edge, merge the two nodes. By merging those two nodes, their related edges and differences get merged. Alternatively, this could be done by merging the two nodes then regenerating the differences again as in step 4 and the c_equation.

6) Keep track of the frequency of occurrence for all differences within all difference sets.

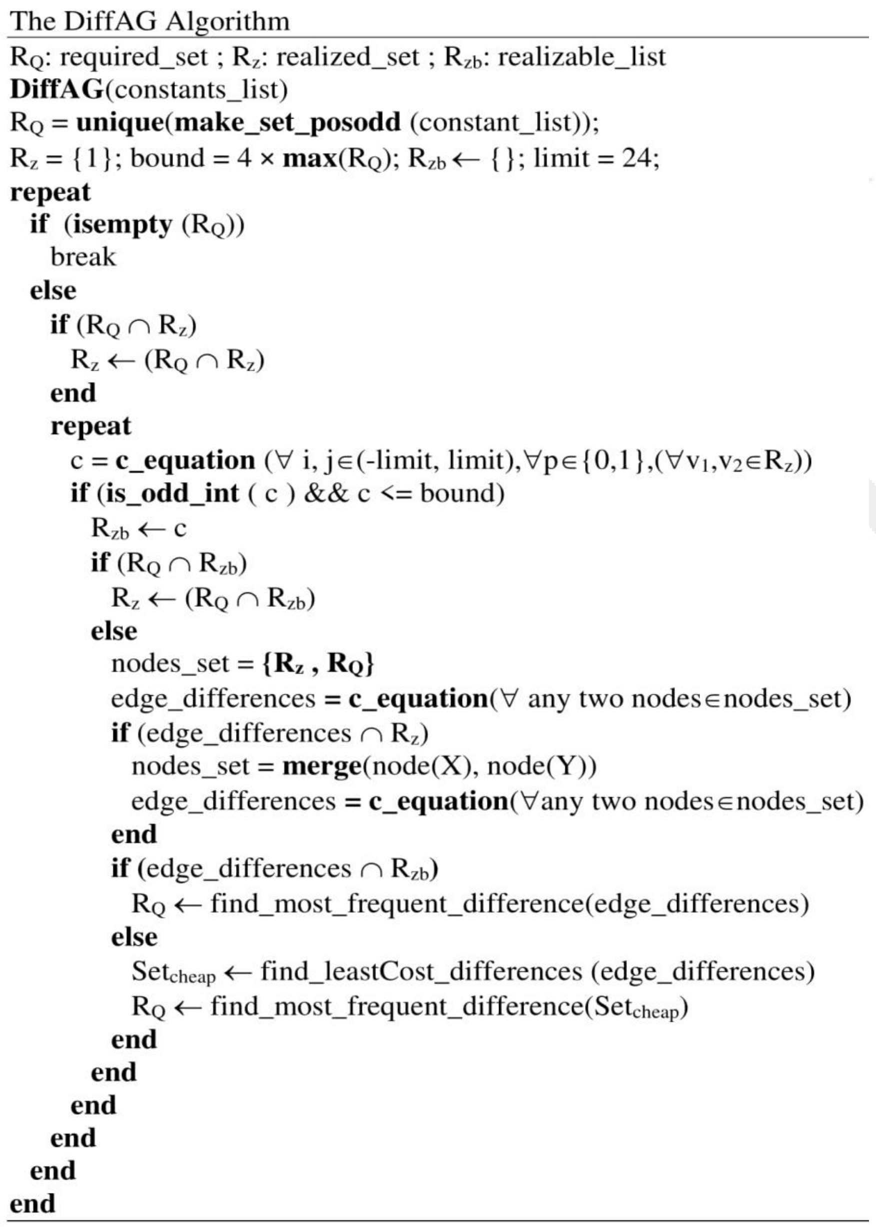
7) Find any differences that are also realizable as obtained in step 2. If so, include the difference that has the most frequency to the required set. Then, return to step 2.

8) Using a viable cost measure, e.g. the canonic signed digit (CSD) representation, calculate the costs for realizing all possible differences. Then, choose the difference that has the most frequency among the difference subset having the least cost. Include that chosen difference in the required set and return to step 2.

It is important to know that the difference that gets picked in step 8 cannot get realized right away. However, it is useful for realizing other required coefficients. Basically, the heuristic is trying to pick the most useful value realizing as many required coefficients as possible.

3.2. The Formulated Algorithm

For the purpose of implementation of the algorithm using Matlab, it is formulated into a concise form. This is also useful to become a reference later on.



```
The DiffAG Algorithm
$R_Q$: required_set ; $R_z$: realized_set ; $R_{zb}$: realizable_list
DiffAG(constants_list)
$R_Q$ = unique(make_set_posodd (constant_list));
$R_z = \{1\}$; bound = $4 \times$ max($R_Q$); $R_{zb} \leftarrow \{\}$; limit = 24;
repeat
  if  (isempty ($R_Q$))
    break
  else
    if ($R_Q \cap R_z$)
      $R_z \leftarrow (R_Q \cap R_z)$
    end
    repeat
      c = c_equation ($\forall$ i, j$\in$(-limit, limit),$\forall p \in \{0,1\}$,($\forall v_1, v_2 \in R_z$))
      if (is_odd_int ( c ) && c <= bound)
        $R_{zb} \leftarrow$ c
        if ($R_Q \cap R_{zb}$)
          $R_z \leftarrow (R_Q \cap R_{zb})$
        else
          nodes_set = {$R_z$ , $R_Q$}
          edge_differences = c_equation($\forall$ any two nodes$\in$nodes_set)
          if (edge_differences $\cap R_z$)
            nodes_set = merge(node(X), node(Y))
            edge_differences = c_equation($\forall$any two nodes$\in$nodes_set)
          end
          if (edge_differences $\cap R_{zb}$)
            $R_Q \leftarrow$ find_most_frequent_difference(edge_differences)
          else
            $Set_{cheap} \leftarrow$ find_leastCost_differences (edge_differences)
            $R_Q \leftarrow$ find_most_frequent_difference($Set_{cheap}$)
          end
        end
      end
    end
  end
end
```

Figure 8: The difference based adder graph in concise form.

3.3. Results

The performance of the developed algorithm is simulated in two experiments. Each has its own conditions in order to illustrate some characteristics of the algorithm.

In experiment 1, several instances are generated randomly. This is useful to know the averages, norms, exceptions, etc. For example, 10 instances each having 10 constants of 16 bit distributed randomly, are generated as in Figure 9. Similarly, 10 instances of 20 constants of 16 bit are considered, as in Figure 10. Then, 10 instances of 30 constants of 16 bit are considered, as in Figure 11. Finally, 10 instances of 40 constants of 16 bit are considered, as in Figure 12.

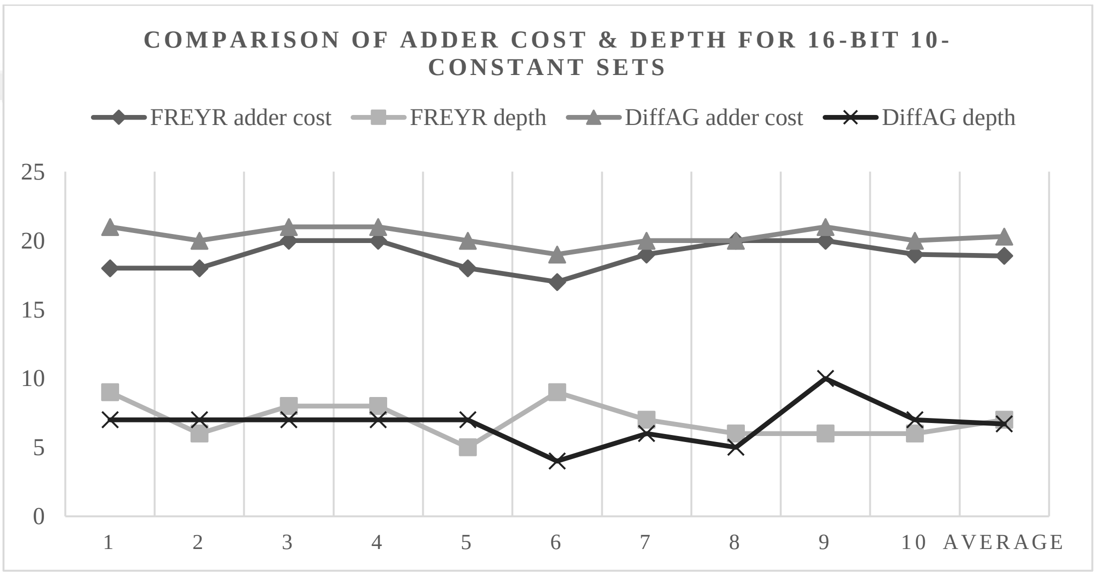


Figure 9: Comparison in terms of adder cost and depth. 16-bit 10-constant sets.

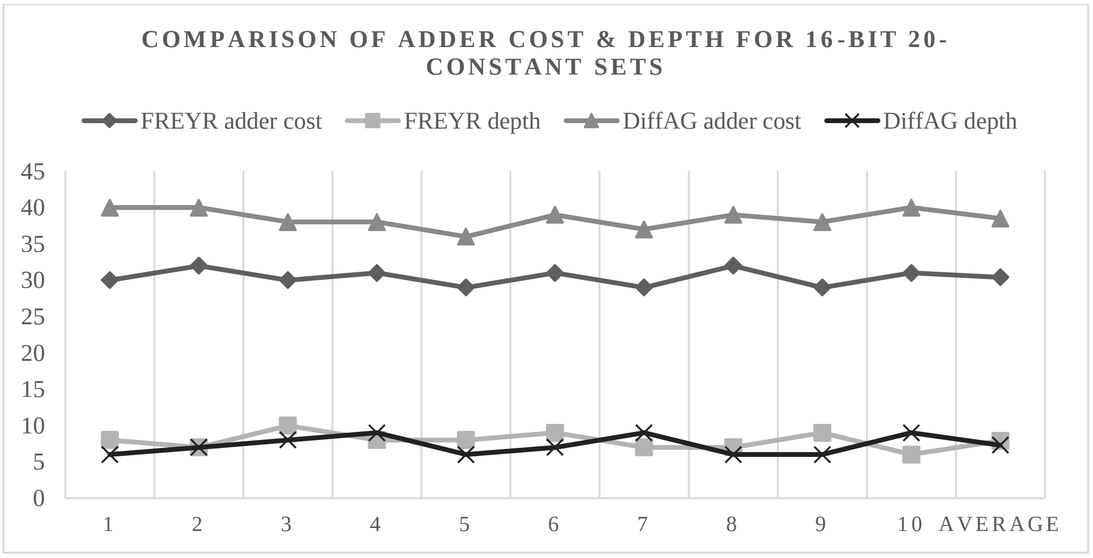


Figure 10: Comparison in terms of adder cost and depth. 16-bit 20-constant sets.

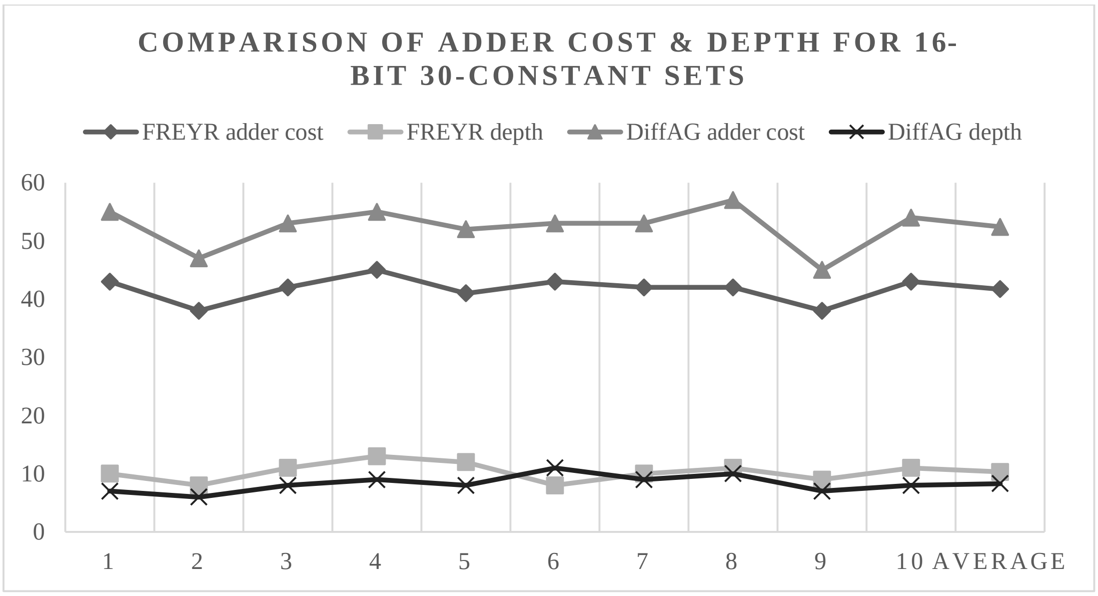


Figure 11: Comparison in terms of adder cost and depth. 16-bit 30-constant sets.

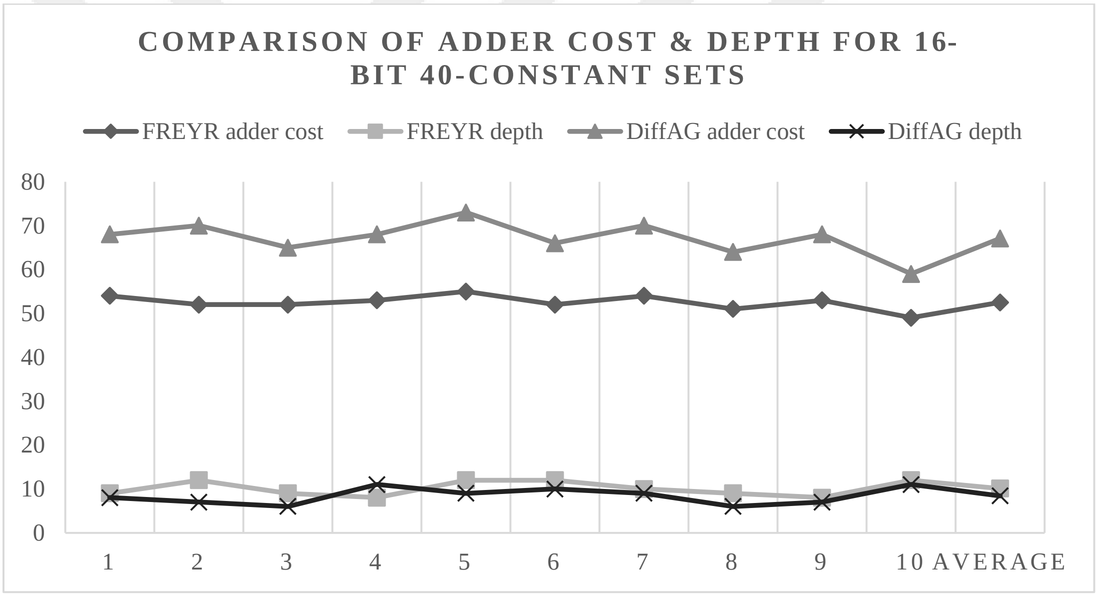


Figure 12: Comparison in terms of adder cost and depth. 16-bit 40-constant sets

In experiment 2, 10-constant sets 8–24 bits are shown in Figure 13. Then, 20-constant sets 8–24 bits gets considered, Figure 14. Also, sets of 30-constant 8–24 bits displayed in Figure 15. Finally, 40-constant sets of 8–24 bit width are generated as illustrated in Figure 16. This experiment finds where algorithm becomes less efficient or gives unwanted results.

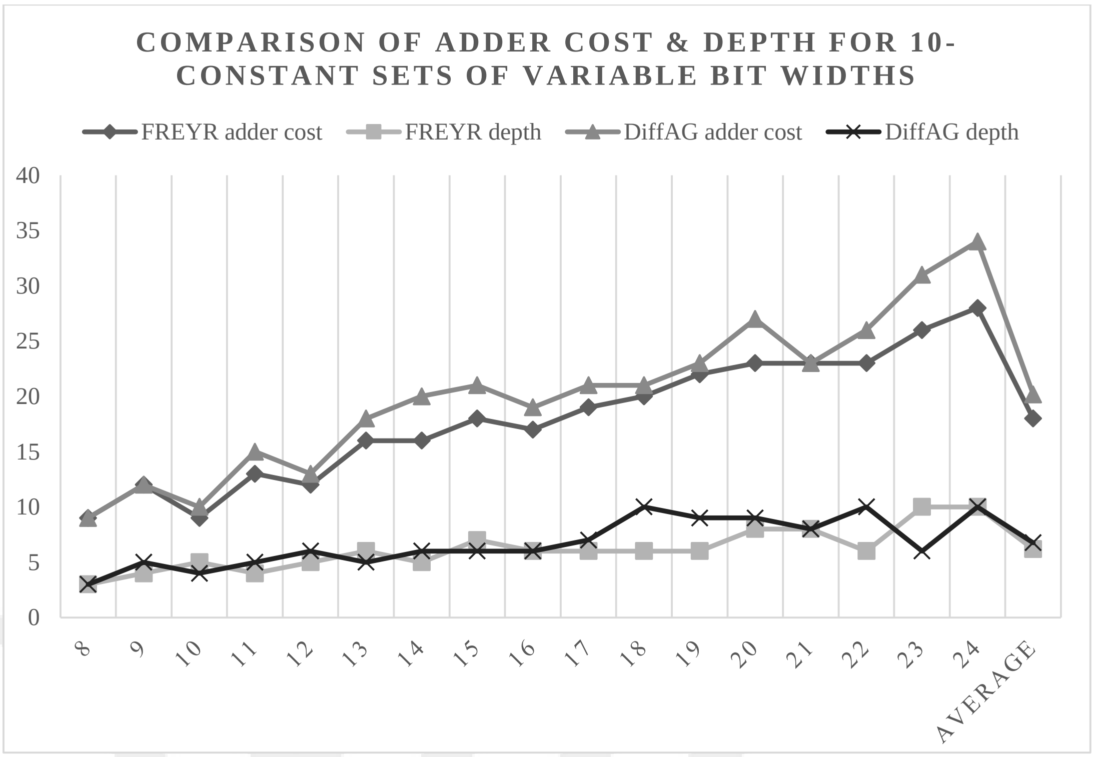


Figure 13: Comparison in terms of adder cost and depth. Variable width 10-constant sets

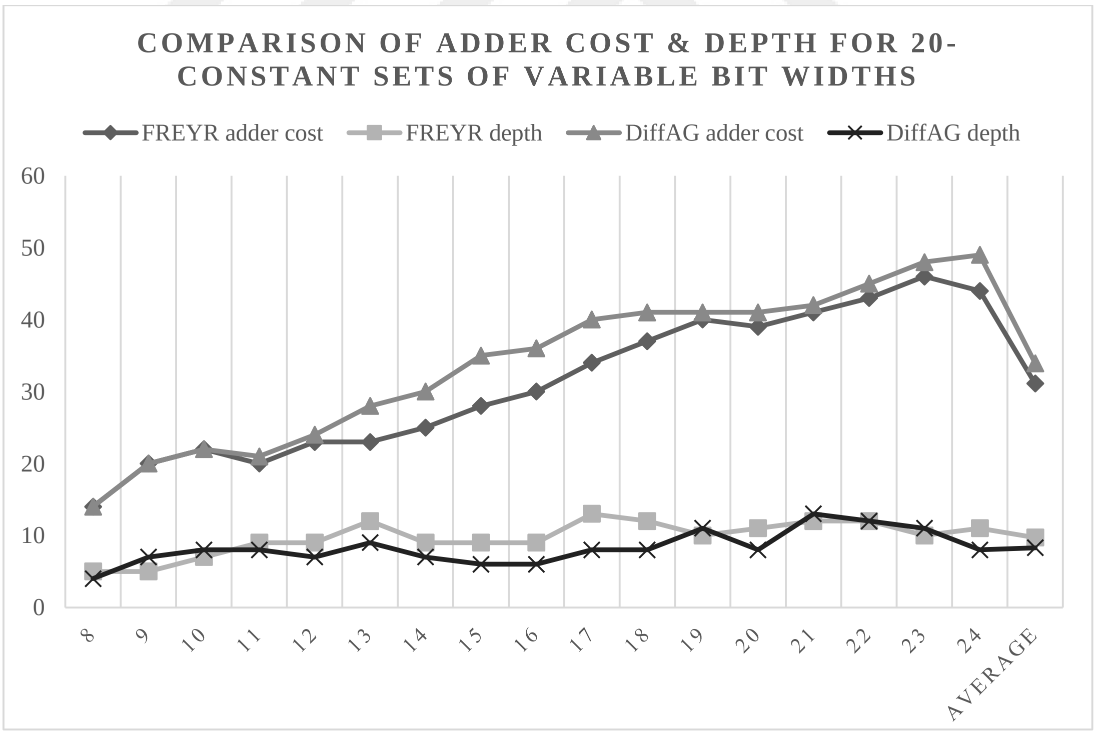


Figure 14: Comparison in terms of adder cost and depth. Variable width 20-constant sets

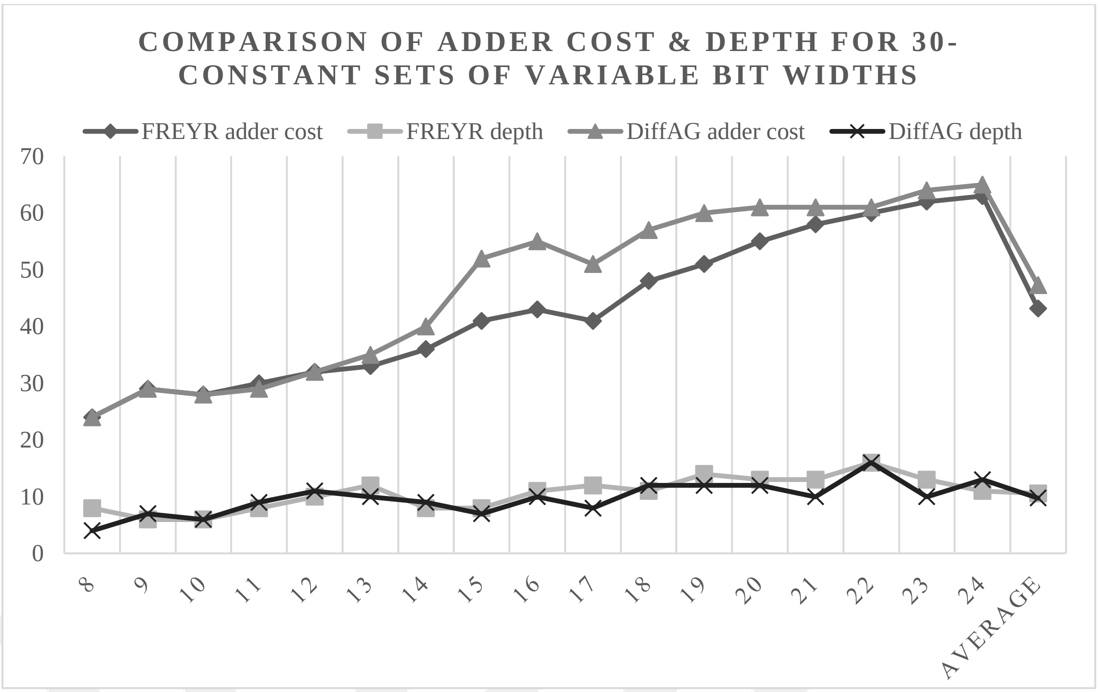


Figure 15: Comparison in terms of adder cost and depth. Variable width 30-constant sets

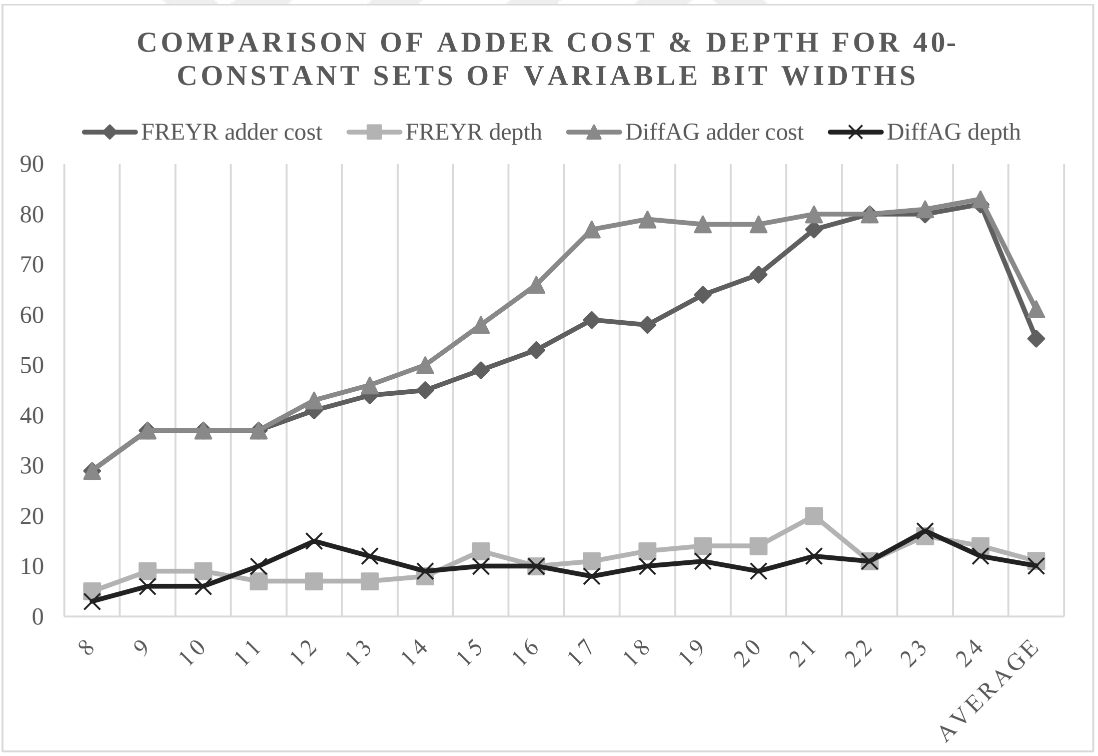


Figure 16: Comparison in terms of adder cost and depth. Variable width 40-constant sets

Overall, the FREYR code aims to solve the MCM problem using the least adder cost while the DiffAG provides solutions that have lower adder depth. As for experiment 1, there are several observations. First, having more required coefficients helps DiffAG work better and find solutions of less adder depth. This can be observed where the average of adder depth decreases as the number of required coefficients increases, compared to FREYR’s results. However, the adder cost of these instances are more than FREYR’s results. This is because DiffAG does not rely on searching several solutions for that MCM instance. To illustrate, in either step 7 or step 8, it would be best to generate several solutions for the same set of required constants. However, the DiffAG has to pick a new required constant during either step 7 or step 8 in an exhaustive manner in order to pick the best solution amongst them.

This idea is especially important in step 8 since it could change the results drastically. The current code relies on finding the minimum cost constant in its Canonic Signed Digit (CSD) representation.

As for experiment 2, when testing both algorithms over different bit widths, it is observed that the FREYR performs better over the range 12 – 19 bits whereas the DiffAG performs better of 19 bit widths and more. This is especially clear in the cases of 30-, 40-constant sets where having more bits and large set sizes gives the DiffAG more options to find a solution using already realized constants. See figures (15–16). However, sometimes the solution becomes chained, using few constants to realize most of the required constants, which results in a solution that has more depth.

DiffAG works in a manner that gives results of low depth. This is inherent in its core idea. However, its results require more adders on average than FREYR’s.

It is noticed that when the heuristic chooses a new required constant, the solution sometimes ends up worse than expected. Choosing another constant may result in a better solution in terms of adder cost or depth. Basically, steps 7 and 8 necessitate a method to explore all possible required coefficients then decide the best one depending on the end results.

CHAPTER FOUR
IMAGE COMPRESSION

4. IMAGE COMPRESSION USING DISCRETE COSINE TRANSFORM

4.1. The Discrete Cosine Transform Using Multiple Constant Multiplications:

A simple method for implementing 2D DCT is in matrix multiplication form. Typically, the transformation is done block by block. Several applications determined to fix the matrix to be 8x8 pixel blocks. This is useful since the transformation matrices are constant for one size. In addition, this size is practical and well known in various standards such as jpeg.

Furthermore, since the DCT coefficients are constant for a certain TU size, this operation can be turned into a multiple constant multiplication problem that can be solved by an algorithm such as DiffAG. Thus, the DCT can be done without the use of multipliers. It is known that multipliers take up space in hardware design. MCM solutions implement the DCT using shifts and additions of the data input. This saves area on the hardware as well as reduces power and increases speed. The only necessity is that one matrix must be constant in order to develop the shift/add network for the MCM block.

However, to implement the DCT using multiplierless matrix multiplications, coefficients must be calculated according to equation 1, previously discussed in section II. This results in an 8x8 matrix of constants to be multiplied with the image data. DCT is computed as below:

$$[\mathrm{Z}_N] = [\mathrm{C_N}][\mathrm{D_N}][\mathrm{C_N}]^{\mathrm{T}} \qquad (3.a)$$

Where $[C_N]$ and $[D_N]$ are the coefficient and input image data matrixes, respectively.

Note that in DCT, the image must firstly be changed into grey level. This represents the useful information in the image without the colors.

4.2. The Multiple Constant Multiplication Solution:

In our work, a DCT block of 8x8 has been chosen similar to works in other research papers. Consequently, 8x8 matrix get constructed to implement the DCT.

```
[0.3535   0.3535  0.3535  0.3535  0.3535  0.3535  0.3535  0.3535
 0.4903   0.4157  0.2777  0.0975 -0.0975 -0.2777 -0.4157 -0.4903
 0.4619   0.1913 -0.1913 -0.4619 -0.4619 -0.1913  0.1913  0.4619
 0.4157  -0.0975 -0.4903 -0.2777  0.2777  0.4903  0.0975 -0.4157
 0.3535  -0.3535 -0.3535  0.3535  0.3535 -0.3535 -0.3535  0.3535
 0.2777  -0.4903  0.0975  0.4157 -0.4157 -0.0975  0.4903 -0.2777
 0.1913  -0.4619  0.4619 -0.1913 -0.1913  0.4619 -0.4619  0.1913
 0.0975  -0.2777  0.4157 -0.4903  0.4903 -0.4157  0.2777 -0.0975]
```

Figure 17: DCT values before manipulation.

However, to solve the MCM problem, this matrix must be entered as an integer matrix to the DiffAG algorithm. A simple solution is to shift the matrix by 10 to the left, or multiplying by 2^{10}. Thus, the integer matrix becomes:

```
[5791     5791    5791    5791    5791    5791    5791    5791
 8033     6810    4549    1597   -1597   -4549   -6810   -8033
 7567     3134   -3134   -7567   -7567   -3134    3134    7567
 6810    -1597   -8033   -4549    4549    8033    1597   -6810
 5791    -5791   -5791    5791    5791   -5791   -5791    5791
 4549    -8033    1597    6810   -6810   -1597    8033   -4549
 3134    -7567    7567   -3134   -3134    7567   -7567    3134
 1597    -4549    6810   -8033    8033   -6810    4549   -1597]
```

Figure 18: DCT values after shifting by 2^{10} to the left.

Note that these values do not affect the final output since the data in the compressed image will be shifted back after processing. Thus, the integer matrix gets entered to the algorithm and generates an MCM solution. Only 7 unique constants are used to implement the matrix multiplications. Each of these constants are implemented using a network of shifts and additions using input values. Below are the illustrations:

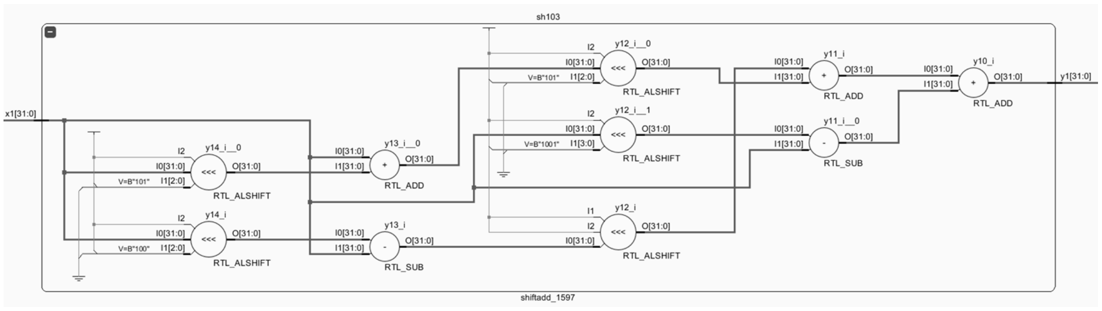


Figure 19: Shift/add network for 1597 multiplier

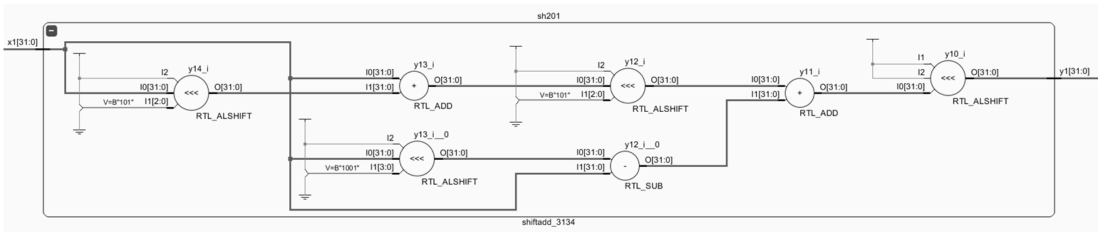


Figure 20: Shift/add network for 3134 multiplier

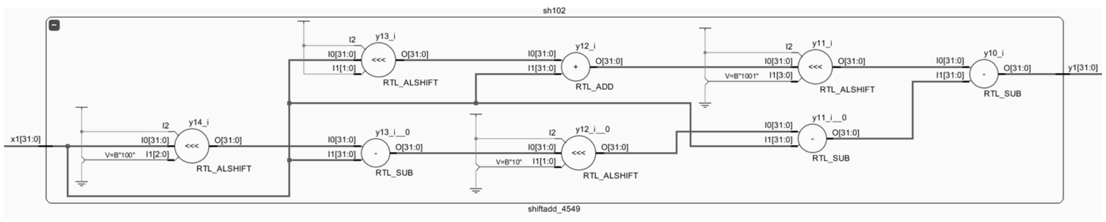


Figure 21: Shift/add network for 4549 multiplier

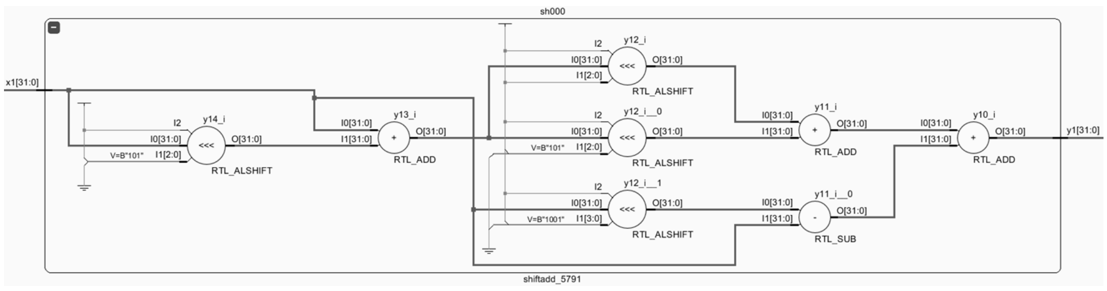


Figure 22: Shift/add network for 5791 multiplier

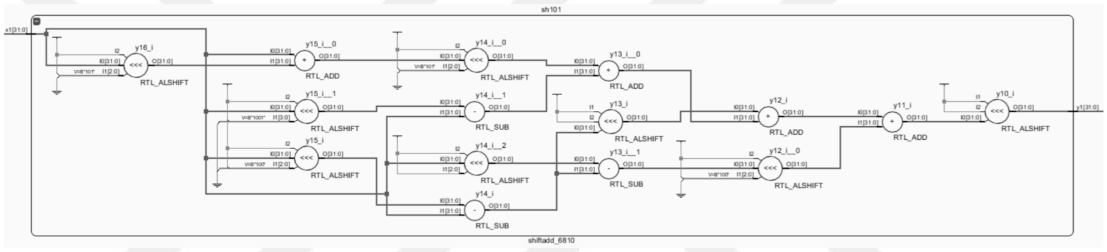


Figure 23: Shift/add network for 6810 multiplier.

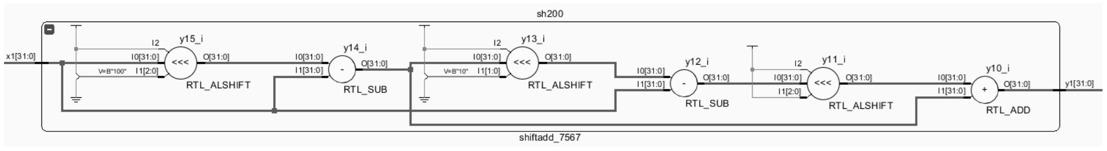


Figure 24: Shift/add network for 7567 multiplier.

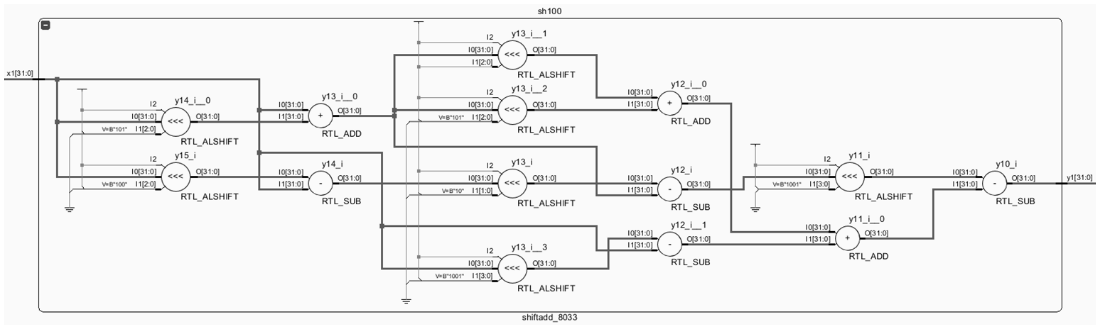


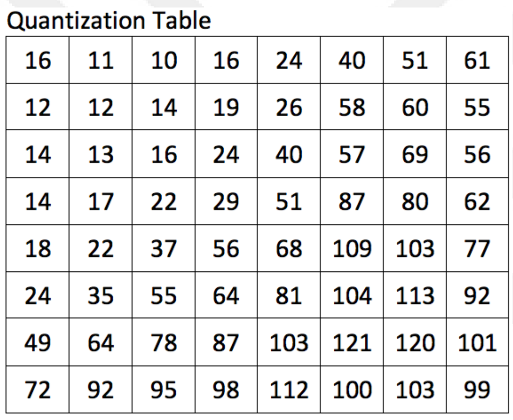
Figure 25: Shift/add network for 8033 multiplier.

4.3. The Block Processing

It is worth noting that the DCT operation must be implemented on all data blocks. This could be processed by one of two ways. Either implement the blocks via a Matlab function called blockproc. Otherwise, it can simply be done via a for loop that determines the limits for each block, operates the DCT for that block and then moves to the next block.

4.4. Image Compression

One of the steps of image compression is quantization process. It is known as multiplying the data obtained after DCT by a quantization table. Below is an example of a quantization table for the JPEG compression standard in figure 26. This table scales that data down by a factor depending on the frequency. To illustrate, traditionally, looking at the results of the DCT operation, high frequency data are scaled down by a factor of 100 while low frequency data are scaled down by 20. In that way, after rounding to the nearest integer, most of the high frequency values are rounded to zero whereas the low frequency are kept.



Quantization Table

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 26: A JPEG quantization table.

After using the quantization table and removing the high frequencies, data is ready for storage using Huffman coding technique. This technique register the data in a zigzag manner giving priority to the lowest frequency values first. More importantly, it stores data that end with a series of zeros in a very efficient way.

However, since low frequency values are kept, the image can essentially be retrieved without noticeable difference. Ofcourse quantization tables differ vastly depending on the quality needed or compression ratio desired.

To retrieve the image from its compressed version, inverse DCT process is required. Simply, the inverse DCT is computed as below:

$$[\mathrm{D}_N] = [\mathrm{C_N}]^{\mathrm{T}}[\mathrm{Z_N}][\mathrm{C_N}] \qquad (3.b)$$

Again MCM blocks can be implemented instead of multipliers. Resulted image needs shifting to the right. Note that the image data must be written into an appropriate type of image in order to display properly.

4.5. Image Compression Algorithm (concise version)

The algorithm developed for image compression process is summarized in the flow chart of Figure 27.

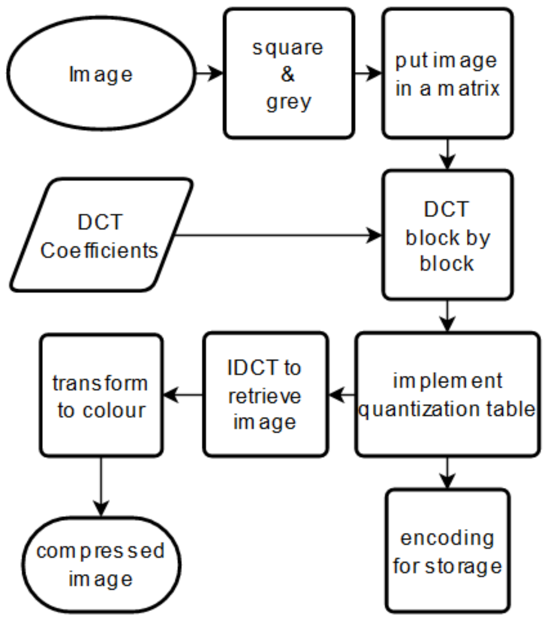


Figure 27: Flow chart of image compression

The input image is turned into grey level as a square image and the image data is placed in a matrix. Using (1), DCT coefficients are obtained (in this case 8x8 size matrix) and DCT is applied to the image data (block by block). A quantization table is applied to the DCT image for all blocks. The resulting data is encoded for storage/transmission or retrieved by applying IDCT to the compressed image. Finally, an appropriate image type is assigned after returning the colors to the image.

4.6. Simulation Results

At first, the image compression algorithm is tested on Matlab. The tested picture is the cameraman.tiff in grey level having the size of 256x256 pixels. Two methods are used: one that involves regular matrix multiplications and one that relies on multiplierless MCM blocks.

There are several achievements from implementing the idea on Matlab. First, the MCM solution had to be verified. By using the 7 shift and add networks generated above, the DCT has been implemented block by block via for loops. The reached values are practically the same, compared to the regular multiplication. In addition, a comparison between the regular multiplication script and the multiplierless multiplication script are made. The multiplierless method is twice as fast as the regular multiplication method, see table 1.

Table 1: Simulation time for DCT in regular multiplication vs. MCM blocks.

	Regular Multiplication	**Multiplierless MCM Blocks**
Time in sec	0.408 sec	0.1775 sec

To reach a quick verification of how the results will perform, it is much easier to implement the whole algorithm on Matlab than on Verilog HDL. Below an example of image compression of the picture cameraman.tiff. See table 2 and figure 28.

Table 2: Compression difference in size for 256x256 cameraman image.

	Before compression	**After compression**	**Compression Ratio**
Picture size	44 KB	19.3 KB	2.28

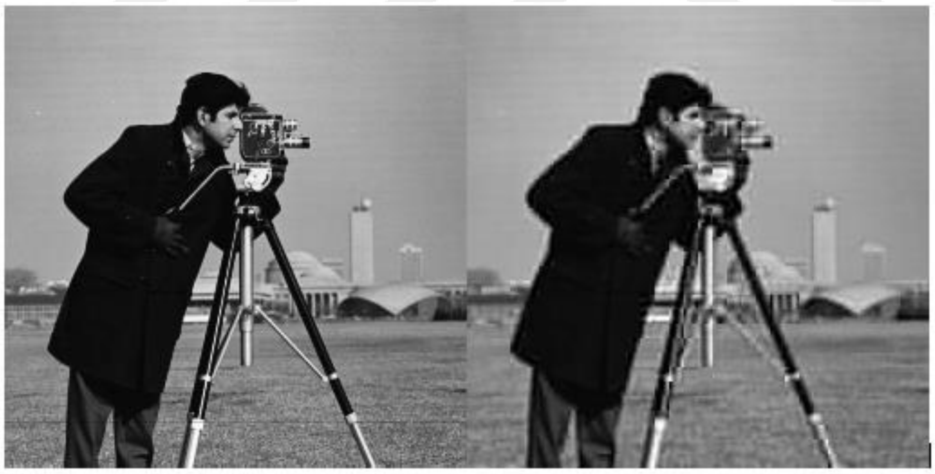


Figure 28: (a) Original image and (b) compressed image of the cameraman.

CHAPTER FIVE

SIMULATION

5. DESIGN OVERVIEW

5.1. Design Method

In this stage, development of 2D DCT application using an HDL is necessary. Verilog language has been picked for developing the algorithm, since it is simpler and easier to read. The 2D DCT code is developed on the basis of row column decomposition method. There are several works that have implemented this method (Atani et al., 2008; Pradeepthi & Ramesh, 2011). However, for the purpose of comparing two MCM solutions, no multipliers are used. The first MCM block is obtained from our DiffAG algorithm presented in section III while the other MCM block is obtained from using FREYR algorithm (Aksoy et al., 2014). The overview of the code is presented below:

This DCT method has been detailed in (Doğan, 2016; Sanjeevannanavar & Nagamani, 2011). According to their works, the 2D DCT can be implemented using 1D DCT twice. This method has been proven more efficient. Using the below equations (4.a – 4.h), a single 1D DCT is computed as follows:

$$Z_0 \;=\; d(x_0+x_7)+d(x_1+x_6)+d(x_2+x_5)+d(x_3+x_4) ----(4.a)$$

$$Z_2 \;=\; b(x_0+x_7)+f(x_1+x_6)-f(x_2+x_5)+b(x_3+x_4) ----(4.b)$$

$$Z_4 \;=\; d(x_0+x_7)-d(x_1+x_6)-d(x_2+x_5)+d(x_3+x_4) ----(4.c)$$

$$Z_6 \;=\; f(x_0+x_7)-b(x_1+x_6)+b(x_2+x_5)-f(x_3+x_4) ----(4.d)$$

$$Z_1 \;=\; a(x_0-x_7)+c(x_1-x_6)+e(x_2-x_5)+g(x_3-x_4) ----(4.e)$$

$$Z_3 \;=\; c(x_0-x_7)-g(x_1-x_6)-a(x_2-x_5)-e(x_3-x_4) ----(4.f)$$

$$Z_5 \;=\; e(x_0-x_7)-a(x_1-x_6)+g(x_2-x_5)+c(x_3-x_4) ----(4.g)$$

$$Z_7 \;=\; g(x_0-x_7)-e(x_1-x_6)+c(x_2-x_5)-a(x_3-x_4) ----(4.h)$$

The main module is used for implementing DCT using MCM blocks. Traditionally, DCT coefficients are multiplied by the input. However, since they are constant, multiplication can be replaced by shift/add operations of the input data. For each 1D DCT vector, the shift/add networks constitute the MCM block that implement the multiplications.

In order to generate the MCM solution, coefficients are entered into the DiffAG algorithm. This provides shift/add networks that represent the coefficients. Thus, for each input from the image matrix that value gets connected to the appropriate network that implement the shift/add operation. In each iteration, a vector of DCT coefficients is computed until 1D DCT is executed in one direction. Then, another 1D DCT is implemented in the other direction using obtained results.

In order to make a comparison between the two MCM solutions, another 2D DCT module is developed using FREYR's solution, see table 3. Note the difference in number of adders and depths. Having less adders results in less area on the chip. However, the less adder depth a design has, the faster and less power consumption it brings about.

Table 3: Comparison between FREYR and DiffAG MCM solutions

MCM Solution	**FREYR algorithm**	**DiffAG algorithm**
Number of Adders	12	14
Adder Depth	5	4

To transform the whole image using DCT, the same process must be repeated block by block in each iteration. For example, an image of 256x256 pixels requires to be segmented into 32 segments of size 8x8. Depending on the complexity of MCM blocks used, the performance of the DCT varies. In addition, the resources used in each cycle differs in terms of area and power.

5.2. Simulation

In this subsection, an explanation of the simulation is briefed. First, all modules are connected to a clock. On each positive edge of that clock the operations execute. The clock is initialized and runs continuously. A data vector gets entered to compute a vector of 1D DCT coefficients. This operation takes 9 clock cycles to compute and the resulted vector is registered. Eight vectors get computed and registered. Then, using the results obtained, another iteration starts. In the same manner, the second 1D DCT is computed and the 2D DCT coefficients are calculated for a block of 8x8. In order to compute DCT for the whole image, another 8x8 block of data is entered and the process repeats.

For simulating the DCT algorithm, a short test bench is prepared using Xilinx Vivado. The design schematics are presented below, see figure 29 and figure 30.

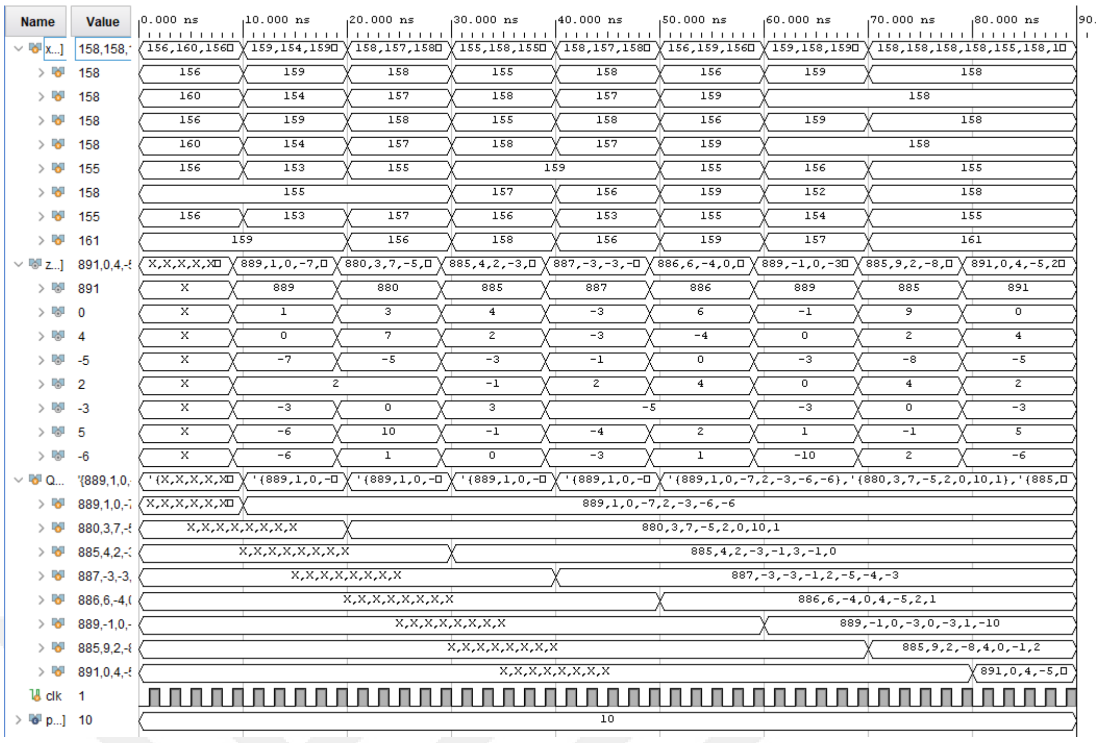


Figure 29: 1D DCT simulation using data from an image.

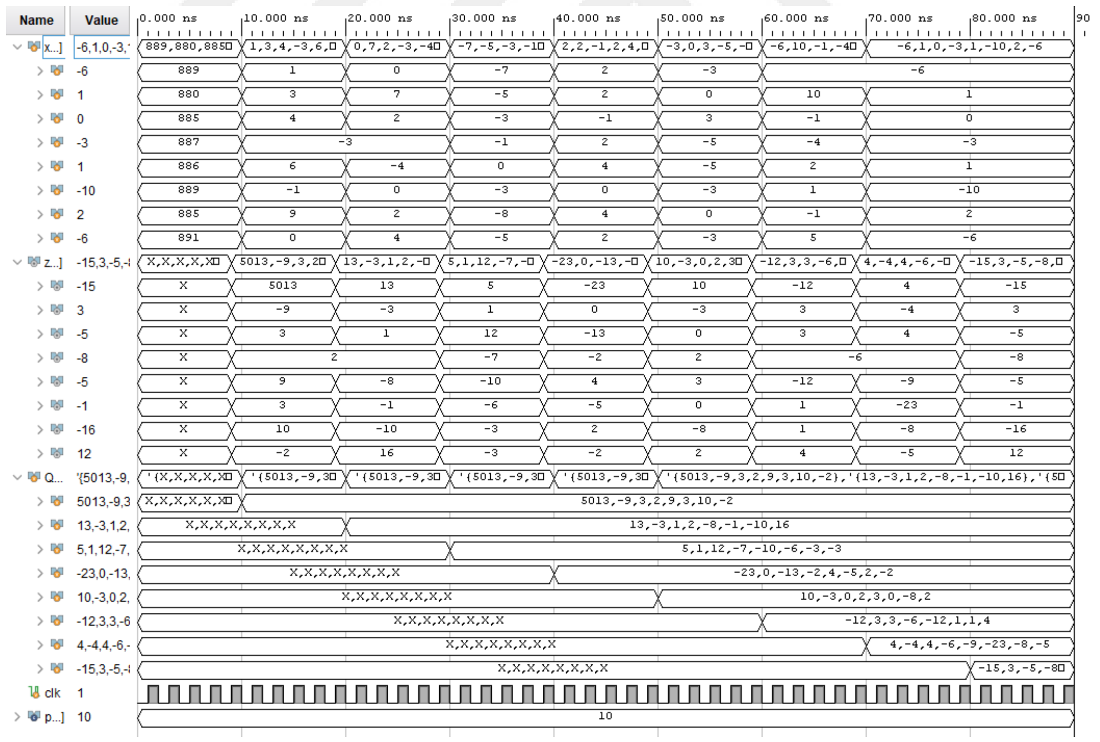


Figure 30: 1D DCT simulation using results from the previous step.

5.3. Register Transfer Level (RTL) Schematics

In figure 31, the addition/subtraction of the input data is performed (called butterfly). Figure 32 shows an example for a 1D DCT coefficient network. Similarly, the rest of DCT vectors are designed using shift/add network modules by changing the first stage modules as needed according to equations (4.a – 4.h) shown previously. The shift/add networks are shown in figures 19–25 in part IV.

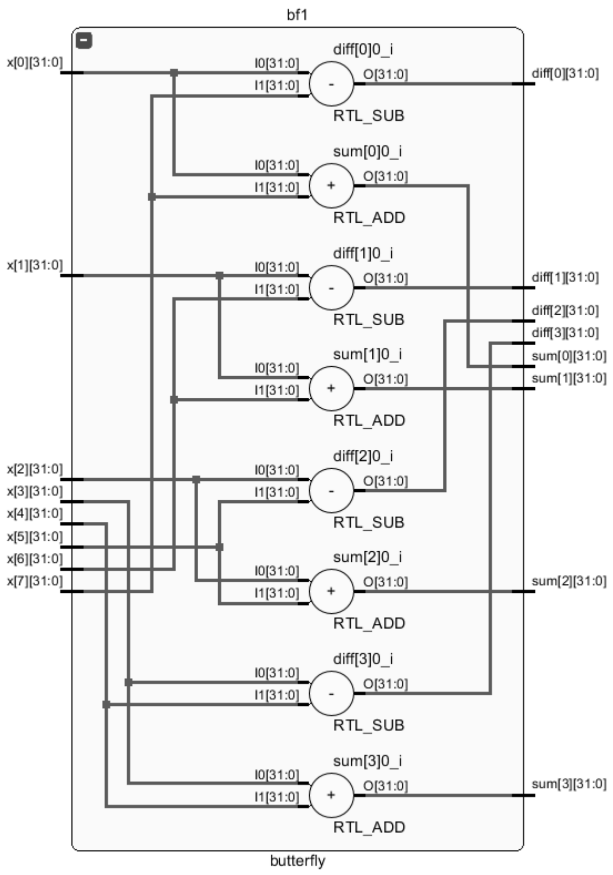


Figure 31: RTL schematic of the butterfly block.

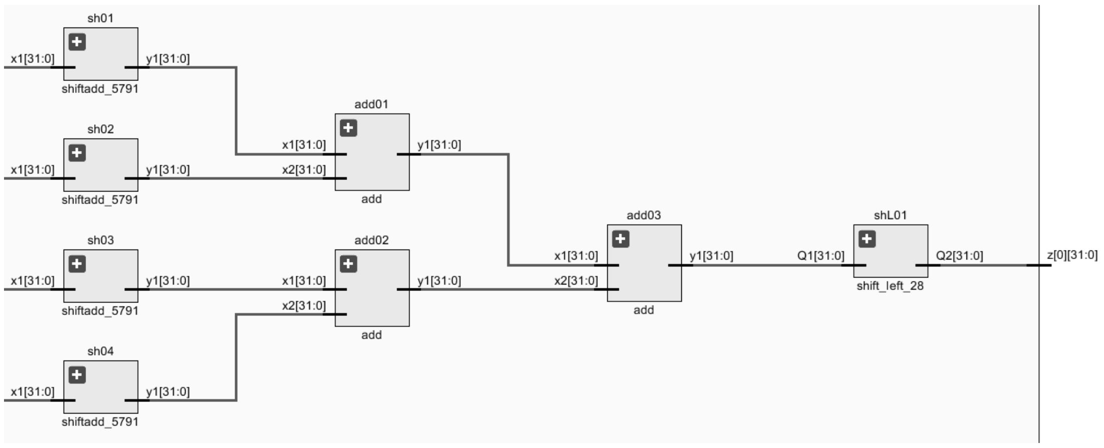


Figure 32: Schematic of a single vector from 1D DCT.

CHAPTER SIX

IMPLEMENTATION

6. HARDWARE IMPLEMENTATION

To implement the 2D DCT design using hardware, Genesys 2 Kintex-7 FPGA (XC7K325T) development board is used. The design has been simulated and synthesized using Xilinx Vivado. A comparison is made between two implementations of 2D DCT designed using FREYR and DiffAG MCM solutions.

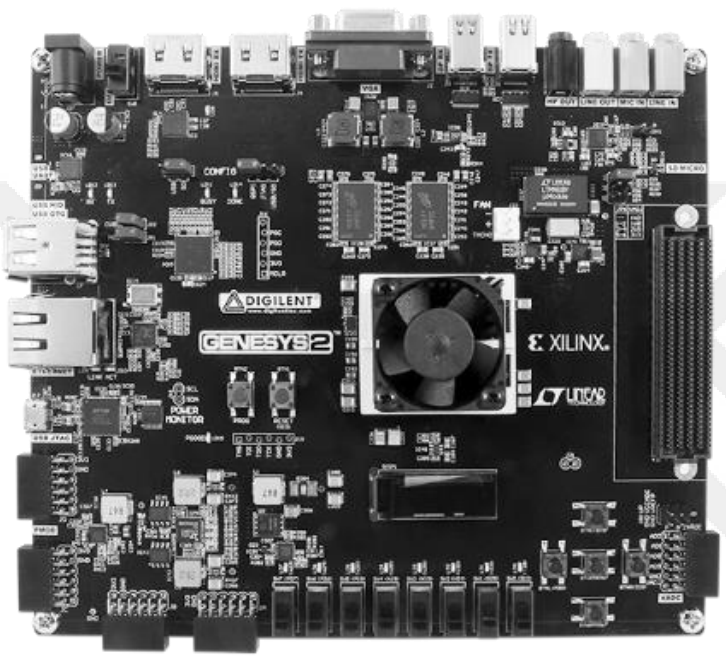


Figure 33: Kintex 7 FPGA board used for hardware implementation.

6.1. Timing Analysis

The DiffAG design has 9 stages that execute serially. Each stage takes a single clock cycle to compute. For a vector to execute, it has to pass through 9 clock cycles. However, an extra clock cycle is used to register the results into RAM. Then, since there are 8 vectors, 80 clock cycles are necessary to compute 1D DCT. An extra buffering stage is included to retrieve the 8 just-registered vectors to be used as input in the next stage. Thus, 80 cycles are necessary to compute the other 1D DCT. Then, an extra buffering clock cycle is necessary to enter the next block of data for processing. Thus, the total number of clock cycles used in this design is 80+1+80+1 = 162 clock cycles.

Similarly, the FREYR design requires 10 stages executed serially. Thus, a 1D DCT requires 88 clock cycles to execute. Then, the total number of clock cycles needed for this design is 178 clock cycles. Table 4 illustrates the timing differences between the two designs.

Table 4: Timing comparison between the two MCM designs using HD spec.

Processing Speed	FREYR	DiffAG	Difference
# Cycles	178 clock cycles	162 clock cycles	-16 clocks
# FPS	35.70 fps	39.22 fps	+3.52 fps
# Speed	0.865 µs	0.787 µs	+9.01% faster

6.2. Logic Area Analysis

The two designs also vary on how much area they takes up on the chip. Table 5 shows the details of how much resources are used implementing 2D DCT in both DiffAG and FREYR MCM solutions.

Table 5: Logic area comparison between the two MCM designs.

Logic Utilization	FREYR	DiffAG	Difference
# LUTs	3588	4341	-20.99%
# Flip-Flops	1944	1952	-0.41%
# Registers	1614	1513	+6.25%
Total %	7146	7806	-9.24% less area

According to table 4 and table 5, since FREYR design is more optimized in terms of number of adders, it takes up less area in the FPGA. However, since DiffAG solution is more optimized in terms of adder depth, it computes DCT in less time than FREYR design. Depending on the application at hand, speed could be considered more important than area, especially since FPGAs nowadays have a plenty of resources to use.

For our design, the execution of 2D DCT of size 8x8 pixel requires 162 clock cycles. When mapped to Kintex 7 FPGA, it takes about 0.787µs for the 8x8 DCT block to compute. This process still can be optimized further. Nevertheless, this design can be used to apply image processing of size 1920x1080 pixel at rate of 39 fps which could be suitable for HD image processing applications.

CONCLUSION

CONCLUSION

The purpose of this thesis is to develop an algorithm that implements 2D DCT using MCM blocks generated by the DiffAG algorithm. This is done in order to test the performance of the DiffAG algorithm and realize its solution on image processing applications. The DiffAG algorithm is first developed and verified using Matlab. Then, its solution has been implemented in image compression. Furthermore, another 2D DCT code is developed using Verilog to simulate and implement the DiffAG solution on FPGA.

Regarding DiffAG, efficiency of the algorithm is validated through its Matlab implementation. The obtained results are compared to FREYR algorithm used in MCM. The DiffAG provides results that have low depth but at the expense of more adder cost. It is worth noting that DiffAG provides better performance when dealing with many constants of low bit width or when dealing with large constants. When compared, DiffAG gives results of adder depth less than FREYR by 17% on average, while FREYR gives results that have adder cost less than DiffAG by 11% on average.

Finally, in order to verify the results obtained from the algorithms, two designs are developed using 2D DCT. Hardware design has been implemented on Kintex 7 FPGA using Xilinx Vivado software. Two designs are used to compare the MCM solutions obtained from DiffAG and FREYR algorithms. Results show that design obtained using FREYR uses more logic area by about 9.2%. However, the design obtained using DiffAG achieves processing rate by about 9.01%. Nevertheless, this design can be used to apply image processing of size 1920x1080 pixel at rate of 39 fps which could be suitable for HD image processing applications.

ACKNOWLEDGEMENTS

This work would not be possible without the support and help received from my professor Mr. Indrit Myderrizi. Also, the support and resources received from Mr. Levent Aksoy is highly appreciated. Finally, this work was possible due to the support of my college Gelişim University.

REFERENCES

Agostini, L. V., Silva, I. S., & Bampi, S. (2007). Multiplierless and fully pipelined JPEG compression soft IP targeting FPGAs. *Microprocessors and Microsystems*, *31*(8), 487–497. https://doi.org/10.1016/j.micpro.2006.02.002

Aksoy, L., Costa, E., Flores, P., & Monteiro, J. (2011). A hybrid algorithm for the optimization of area and delay in linear DSP transforms. *2011 IEEE/IFIP 19th International Conference on VLSI and System-on-Chip, VLSI-SoC 2011*, *1*(c), 148–153. https://doi.org/10.1109/VLSISoC.2011.6081637

Aksoy, L., Flores, P., & Monteiro, J. (2014). Efficient design of FIR filters using hybrid multiple constant multiplications on FPGA. *2014 32nd IEEE International Conference on Computer Design, ICCD 2014*, 42–47. https://doi.org/10.1109/ICCD.2014.6974660

Anghel, L., Velazco, R., Saleh, S., Deswaertes, S., & El Moucary, A. (2003). Preliminary validation of an approach dealing with processor obsolescence. *Proceedings - IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, *2003-Janua*, 493–501. https://doi.org/10.1109/TSM.2005.1250148

Atani, R. E., Baboli, M., Mirzakuchaki, S., Atani, S. E., & Zamanlooy, B. (2008). Design and implementation of a 118 MHz 2D DCT processor. *IEEE International Symposium on Industrial Electronics*, *i*, 1076–1081. https://doi.org/10.1109/ISIE.2008.4677274

Azgin, H., Kalali, E., & Hamzaoglu, I. (2019). An Efficient FPGA Implementation of Versatile Video Coding Intra Prediction. *Proceedings - Euromicro Conference on Digital System Design, DSD 2019*, 194–199. https://doi.org/10.1109/DSD.2019.00037

Bomar, B. W. (2002). Implementation of microprogrammed control in FPGAs. *IEEE Transactions on Industrial Electronics*, *49*(2), 415–422. https://doi.org/10.1109/41.993275

Chandra, A., & Chattopadhyay, S. (2016). Design of hardware efficient FIR filter: A review of the state-of-the-art approaches. *Engineering Science and Technology, an International Journal*, *19*(1), 212–226. https://doi.org/10.1016/j.jestch.2015.06.006

Dempster, A. G., & Macleod, M. D. (1995). Use of Minimum-Adder Multiplier Blocks in FIR Digital Filters. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, *42*(9), 569–577. https://doi.org/10.1109/82.466647

Doğan, A. (2016). An efficient low area implementation of 2-D DCT on FPGA. *ELECO 2015 - 9th International Conference on Electrical and Electronics Engineering*, 771–775. https://doi.org/10.1109/ELECO.2015.7394534

Fu, B. Y., Wu, E., Sirasao, A., Attia, S., Khan, K., & Wittig, R. (2017). *Deep Learning with INT8 Optimization on Xilinx Devices INT8 for Deep Learning INT8 Deep Learning on Xilinx DSP Slices*. *486*, 1–11.

Garrido, M. J., Pescador, F., Chavarrias, M., Lobo, P. J., & Sanz, C. (2018). A High Performance FPGA-Based Architecture for the Future Video Coding Adaptive Multiple Core Transform. *IEEE Transactions on Consumer Electronics*, *64*(1), 53–60. https://doi.org/10.1109/TCE.2018.2812459

Gately, M. B., Yeary, M. B., & Tang, C. Y. (2012). Multiple real-constant multiplication with improved cost model and greedy and optimal searches. *ISCAS 2012 - 2012 IEEE International Symposium on Circuits and Systems*, 588–591. https://doi.org/10.1109/ISCAS.2012.6272099

Gustafsson, O. (2007). A difference based adder graph heuristic for multiple constant multiplication problems. *Proceedings - IEEE International Symposium on Circuits and Systems*, 1097–1100. https://doi.org/10.1109/iscas.2007.378201

Hussain, A. J., Al-Fayadh, A., & Radi, N. (2018). Image compression techniques: A survey in lossless and lossy algorithms. In *Neurocomputing* (Vol. 300). Elsevier B.V. https://doi.org/10.1016/j.neucom.2018.02.094

ITU-T. (2019). High Efficiency Video Coding. Recommendation ITU-T H.265. *ITU Telecommunication Standardization Sector (ITU-T), 265,* 1. https://www.itu.int/rec/T-REC-H.265-201911-I/en

Johansson, K., Gustafsson, O., Debrunner, L. S., & Wanhammar, L. (2011). Minimum adder depth multiple constant multiplication algorithm for low power FIR filters. *Proceedings - IEEE International Symposium on Circuits and Systems*, *1,* 1439–1442. https://doi.org/10.1109/ISCAS.2011.5937844

Kalali, E., Mert, A. C., & Hamzaoglu, I. (2016). A Computation and Energy Reduction Technique for HEVC Discrete Cosine Transform. *IEEE Transactions on Consumer Electronics*, *62*, 166–174.

Kumm, M., & Zipf, P. (2012). Hybrid multiple constant multiplication for FPGAs. *2012 19th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2012*, *2*, 556–559. https://doi.org/10.1109/ICECS.2012.6463686

Kusuma, E. D., & Widodo, T. S. (2010). FPGA implementation of pipelined 2D-DCT and quantization architecture for JPEG image compression. *Proceedings 2010 International Symposium on Information Technology - Visual Informatics, ITSim'10*, *1*. https://doi.org/10.1109/ITSIM.2010.5561411

Mert, A. C., Azgin, H., Kalali, E., & Hamzaoglu, I. (2018). Efficient multiple constant multiplication using DSP blocks in FPGA. *Proceedings - 2018 International Conference on Field-Programmable Logic and Applications, FPL 2018*, 331–334. https://doi.org/10.1109/FPL.2018.00063

Patil, A. (2010). *Fpga Implementation of Digital Filters Using Mcm* [The Florida State University]. http://fsu.digital.flvc.org/islandora/object/fsu%3A180368

Paulitsch, M., Schmidt, E., Scherrer, C., & Kantz, H. (2011). Industrial applications. *Time-Triggered Communication*, *7*(2), 303–359. https://doi.org/10.1533/9781845698843.31

Pourazad, M. T., Doutre, C., Azimi, M., & Nasiopoulos, P. (2012). HEVC: The New Gold Standard for Video Compression: How Does HEVC Compare with H.264/AVC? *IEEE Consumer Electronics Magazine*, *1*(3), 36–46. https://doi.org/10.1109/MCE.2012.2192754

Pradeepthi, & Ramesh, A. P. (2011). Pipelined Architecture of 2D-DCT, Quantization and ZigZag Process for JPEG Image Compression Using VHDL. *International Journal of VLSI Design & Communication Systems*, *2*(3), 99–110. https://doi.org/10.5121/vlsic.2011.2308

Rodriguez-Andina, J. J., Moure, M. J., & Valdes, M. D. (2007). Features, design tools, and application domains of FPGAs. *IEEE Transactions on Industrial Electronics*, *54*(4), 1810–1823. https://doi.org/10.1109/TIE.2007.898279

Ronak, B., & Fahmy, S. A. (2016). Mapping for maximum performance on FPGA DSP blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *35*(4), 573–585. https://doi.org/10.1109/TCAD.2015.2474363

Sanjeevannanavar, S., & Nagamani, A. N. (2011). Efficient design and FPGA implementation of JPEG encoder using verilog HDL. *Proceedings of the International Conference on Nanoscience, Engineering and Technology, ICONSET 2011*, 584–588. https://doi.org/10.1109/ICONSET.2011.6168038

Shen, Y. F., Li, J. T., Zhu, Z. M., & Zhang, Y. D. (2013). High efficiency video coding. In *Jisuanji Xuebao/Chinese Journal of Computers* (Vol. 36, Issue 11). https://doi.org/10.3724/SP.J.1016.2013.02340

Stephen A. Martucci. (1996). ZEROTREE ENTROPY CODING OF WAVELET COEFFICIENTS FOR VERY LOW BIT RATE VIDEO Stephen A . Martucci and Iraj Sodagar Subsidiary of SRI International Princeton , NJ 08543--5300. *Communications*, 533–536.

Thong, J., & Nicolici, N. (2010). Combined optimal and heuristic approaches for multiple constant multiplication. *Proceedings - IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 266–273. https://doi.org/10.1109/ICCD.2010.5647750

Tresa, L. E., & Sundararajan, M. (2013). Video Compression Using Discrete Wavelet Transform. *I-Manager's Journal on Digital Signal Processing*, *1*(3), 1–7. https://doi.org/10.26634/jdp.1.3.2436

Vanne, J., Viitanen, M., Hamalainen, T. D., & Hallapuro, A. (2012). Comparative rate-distortion-complexity analysis of HEVC and AVC video codecs. *IEEE Transactions on Circuits and Systems for Video Technology*, *22*(12), 1885–1898. https://doi.org/10.1109/TCSVT.2012.2223013

Voronenko, Y. (n.d.). *Spiral*. Carnegie Mellon University. http://spiral.ece.cmu.edu/mcm/gen.html

Voronenko, Y., & Püschel, M. (2007). Multiplierless multiple constant multiplication. *ACM Transactions on Algorithms*, *3*(2).

https://doi.org/10.1145/1240233.1240234

Vrindavanam, J. (2012). *A Survey of Image Compression Methods*. *February 2012*, 12–17.

Woods, R. C. (2008). *Digital Image processing* (Third Edit). Pearson Pentice Hall.

Abdurrahman, K., & Indrit, M. (2021). Multiple Constant Multiplication using Difference based Adder Graph. *The 10th International Symposium on Signal, Image, Video and Communications. Accepted for Presentation.*

APPENDICES

A. DIFFAG MATLAB CODE

```
clc; clear all;
% echo find_edge_differences on
% pause
%% 0) get constants and make em posodd
[file_name,file_coef,constants_list] = read_constants ();
tic;
constants_list = fix(constants_list*16384)';
constants_list_posodd = make_set_posodd(constants_list);
time(1) = toc;
%% 1) form two sets: required and realized
tic;
% required_set = constants_list(:,4)';
required_set = unique(constants_list_posodd(:,4))';
original_set = required_set;
realized_set = [1,1,1,0,0,0];
bound = 4 * max(required_set);
cycle = 0; total_time = 0;
time(2) = toc;
while 1
    %% 2 & 3) try and realize any numbers using equation 1
    tic;
    [o,~]=size(realized_set);
    for h = 1:o
        if ismember(realized_set(h,1),required_set)
            required_set(realized_set(h,1) == required_set) = []
            fprintf('found a required constant that was realized already
\n');
            pause
        end
    end
    [required_set,realized_set,realizable_list] =
Hcub_optimal(required_set,realized_set,bound);
%          realized_set
    if isempty(required_set)
        realized_set(1,:)=[]
        fprintf('well done GG \n');
        break;
    end
    cycle = cycle + 1 ;
    fprintf('cycle number %d \n',cycle);
    time(3) = toc;
    %% 4.a) form the nodes graph
    tic;
    [nodes_set] = form_nodes_set(required_set,realized_set);
    time(4) = toc;
    %% 4.b) find edge difference sets
    tic;
    [differences,field_names] = find_edge_differences (nodes_set,bound);
    time(5) = toc;
    %% 5) check for merging and merge if possible
    tic;
    [nodes_set,differences,field_names] =
merge_nodes(realized_set,nodes_set,differences,field_names,bound);
    [~,n] = size(nodes_set);
%     required_set
    time(6) = toc;
    %% 6) statistics on difference occurance
    tic;
    [differences] = difference_stats(differences,field_names);
    time(7) = toc;
    %% 7) find most frequent realizable difference and add it to required
set
    tic;
    [freq_diff_list,max_occurance] =
frequent_difference(differences,field_names,realizable_list);
    if ~isempty(freq_diff_list)
        [freq_diff_list,~] = lowest_cost_differences_ver2(freq_diff_list);
% note we could choose the coefficient differently using random
        required_set = [freq_diff_list(1,1),required_set];
% method/seed to find several solutions and pick the best one
        time(8) = toc;
    else
        %% 8) find most frequent difference with least cost and add it to
required set
        tic;
        [cheap_differences,cost] =
lowest_cost_differences(differences,field_names);
        [most_freq_diff] =
find_freq_diff_among_all(differences,field_names,cheap_differences);
        required_set = [most_freq_diff,required_set];
        time(9) = toc;
    end
    temp_time = sum(time(:));
    total_time = temp_time + total_time;
end
%% 9) write results
file_name2=[file_name,'-1'];
write_results(file_name2,file_coef,original_set,cycle,realized_set,total_tim
e);
[m,~] = size(realized_set);
fprintf('\n');
fprintf('* Summary of results \n');
fprintf('* Number of required constants %d \n',length(original_set));
fprintf('* Number of cycles: %d \n',cycle);
fprintf('* Number of operations: %d \n',m);
fprintf('* Number of adder-steps : %d \n',max(realized_set(:,6)));
fprintf('* Total CPU time : %.2f \n',total_time);
```

```
function [c] = c_equation (v1,c1,v2,c2,bound)
w = 1; c = zeros(1,6);
dynamic_limit = ceil(log2(bound));
limit = dynamic_limit;
% for limit = 1:dynamic_limit
for i = -limit:limit
    for j = -limit:limit
        for p = 0:1
            mult_i = 2^i; mult_j = ((-1)^p)*(2^j);
            temp = abs(mult_i*v1 + mult_j*v2);
            if (temp ~= v1 && temp ~= v2)
                if temp > bound
                    continue
                elseif ((fix(temp)==temp) && (mod(temp,2)))
                    if c1 >= c2
                        cost = 1+c1;
                    else
                        cost = 1+c2;
                    end
                    if temp > 1
                        cond_i = (temp==c(:,1));
                        if (sum(cond_i)>1)
                            ind = find(cond_i);
                            for e = 1:length(ind)
                                if  cost<c(ind(e),6)
                                    c(ind(e),:) =
[temp,mult_i,v1,mult_j,v2,cost];
                                end
                            end
                        else
                            c(w,:) = [temp,mult_i,v1,mult_j,v2,cost];
                            w = w+1;
                        end
                    end
                end
            end
        end
    end
end
% end
end
```

```
function [merge_ind]  =
check_to_merge_nodes(difference_set,realized_constants)

ind_occurance = ismember(difference_set,realized_constants);
if sum(ind_occurance)
    merge_ind = 1;
else
    merge_ind = 0;
end
end
```

```
function [differences] = difference_stats(differences,field_names)
all_elements = [];
for z = 1:length(field_names)
    var_name = char(field_names(z));
    temp = differences.(var_name)(1:end,1)';
    all_elements = [all_elements,temp];
end

for i = 1:length(field_names)
    var_name1 = char(field_names(i));
    set_1 = differences.(var_name1)(1:end,1)';
    for k = 1:length(set_1)
        count = sum(ismember(set_1(k),all_elements));
        differences.(var_name1)(k,6) = count;
    end
end
end
```

```
function [cheapest_difference] = find_cheapest_difference(difference_set)
%
difference_set=[725,7;1387,7;1899,7;1907,8;2103,8;14879,3;6969,9;9033,6;5143
,3]
min_cost = min(difference_set(:,2));
ind = difference_set(:,2) == min_cost;
temp = difference_set(ind,1);
temp = order_cheapest_first (temp);
cheapest_difference = temp(1,1);
end
```

```
function [differences,field_names] = find_edge_differences (nodes_set,bound)
[~,n] = size(nodes_set);
for i= 1:n
    for j= 1:n
        if (i~=j && j>i)
            c = [];
            node_X = cell2mat(nodes_set(i));
            node_Y = cell2mat(nodes_set(j));
            for x = 1:length(node_X)
                for y=1:length(node_Y)
                    temp = c_equation(node_X(x),0,node_Y(y),0,bound);
                    c = [c;temp];
                    c = unique(c,'rows');
                end
            end
            %% just register results in a structure
            var_name = assign_struct_name(i,j);
            differences.(var_name) = c ;
            differences.(var_name)(1,7:length(node_X)+6) = node_X;
            differences.(var_name)(2,7:length(node_Y)+6) = node_Y;
        end
    end
```

```
end
field_names = fieldnames(differences);
end
```

```
function [most_freq_diff] = find_freq_diff_among_all
(differences,field_names,cheap_differences)
most_freq_diff = [];
old_freq = 0;
for t = 1:length(cheap_differences)
    new_freq =
find_freq_difference(differences,field_names,cheap_differences(t));
    if new_freq > old_freq
        old_freq = new_freq;
        most_freq_diff = cheap_differences(t);
    elseif new_freq == old_freq
        old_freq = new_freq;
        most_freq_diff = union(most_freq_diff,cheap_differences(t));
    end
end
most_freq_diff = min(most_freq_diff);
end
```

```
function [frequency] =
find_freq_difference(differences,field_names,difference)
frequency = 0 ;
for t = 1:length(field_names)
    var_name = char(field_names(t));
    difference_list= transpose(differences.(var_name)(1:end,1));
    if ismember(difference,difference_list)
        frequency = frequency +1 ;
    end
end
end
```

```
function [my_brain_hurts,cost] = find_min_cost_element(set)
set = sortrows(set,6);
my_brain_hurts = set(1,1);
cost = set(1,6);
end
```

```
function [optimal_differences,old_cost] =
find_optimal_difference(difference_set)
% difference_set = randi([1 100],18);
optimal_differences = [];
old_cost = 100;
for i = 1:length(difference_set)
    if (difference_set(i) > 1)  %%%%%%%%%%%%
        [new_cost,~] = find_csd_representation(difference_set(i));
        if (new_cost < old_cost)
            old_cost = new_cost;
            optimal_differences = difference_set(i);
        elseif (new_cost == old_cost)
            old_cost = new_cost;
            optimal_differences =
union(optimal_differences,difference_set(i));
        end
    end
end
end
```

```
function [nodes_set] = form_nodes_set(required_set,realized_set)
nodes_set = cell(1,length(required_set)+1);
nodes_set{1} = realized_set(:,1)';
for i = 2:length(required_set)+1
    nodes_set{i} = required_set(i-1);
end
end
```

```
function [freq_diff_full_list,max_occurance] =
frequent_difference(differences,field_names,realizable_list)
max_occurance = 0; difference_full_list = zeros(1,2);temp = zeros(1,2);
difference_full_list(1,:)=[];
freq_diff_full_list = [];
```

```
for t = 1:length(field_names)
    var_name = char(field_names(t));
    temp = differences.(var_name)(1:end,1);
    occurance_list = differences.(var_name)(1:end,6);
    combination = [temp,occurance_list];
    difference_full_list = [difference_full_list;combination];
end
difference_full_list = unique(difference_full_list,'rows');
[~,ia,~] = intersect(difference_full_list(:,1),realizable_list(:,1));
if isempty(ia)
    freq_diff_full_list = [];
    return;
else
    difference_full_list = difference_full_list(ia,:);
    max_occurance = max(difference_full_list(:,2));
    ind = (difference_full_list(:,2) == max_occurance);
    if ~isempty(ind)
        freq_diff_full_list = difference_full_list(ind,:);
    end
end
end
```

```
function [required_set,realized_set,realizable_full_list] =
Hcub_optimal(required_set,realized_set,bound)
realizable_full_list = [];
[o,~]=size(realized_set);
i = 1;
while i <= o
    j = 1;
    while j <= o
        [realizable_list] =
c_equation(realized_set(i,1),realized_set(i,6),realized_set(j,1),realized_se
t(j,6),bound);
        realizable_full_list = [realizable_full_list;realizable_list];
        [m,~]=size(realizable_list);
        for t = 1:m
            if ismember(realizable_list(t,1),required_set)
                realized_set = [realized_set;realizable_list(t,:)];
                [o,~]=size(realized_set);
                required_set(realizable_list(t,1)==required_set) = [];
                i = 1; j = 0;
                break;
            end
        end
        j = j+1;
    end
    i = i+1;
end
realizable_full_list = unique(realizable_full_list,'rows');
end
```

```
function [old_differences,old_cost] =
lowest_cost_differences(differences,field_names)
old_cost = 100; old_differences = [];
for t = 1:length(field_names)
    var_name = char(field_names(t));
    difference_set = transpose(differences.(var_name)(1:end,1));
    [new_differences,new_cost] = find_optimal_difference(difference_set);
    if new_cost<old_cost
        old_cost = new_cost;
        old_differences = new_differences;
    elseif new_cost == old_cost
        old_cost = new_cost;
        old_differences = union(old_differences,new_differences);
    end
end
end
```

```
function [old_differences,old_cost] = lowest_cost_differences_ver2(set)
[old_differences,old_cost] = find_optimal_difference(set);
```

```
end
```

```
function [nodes_set,differences,field_names] =
merge_nodes(realized_set,nodes_set,differences,field_names,bound)
realized_constants = realized_set(1:end,1);
i = 1;
while i <= length(field_names)
    var_name = char(field_names(i));
    difference_list = transpose(differences.(var_name)(1:end,1));
    merge_ind = check_to_merge_nodes(difference_list,realized_constants);
    if merge_ind
        node_X = differences.(var_name)(1,7:end);
        node_Y = differences.(var_name)(2,7:end);
        node_X = nonzeros(node_X)';
        node_Y = nonzeros(node_Y)';
        index_x = [];
        index_y = [];
        for k = 1:length(nodes_set)
            if isequal(nodes_set{k}, node_X)
                index_x = k;
            elseif isequal(nodes_set{k}, node_Y)
                index_y = k;
            end
        end
        nodes_set{index_x} = [node_X,node_Y];
        nodes_set(index_y) = [];
        %%
        [~,n] = size(nodes_set);
        if n == 1
            fprintf('the last two nodes were merged \n');
            break ;
        else
        [differences,field_names] =find_edge_differences (nodes_set,bound) ;
        i = 0;
        end
    end
    i = i+1;
end
end
```

```
function [required_set] = order_cheapest_first (required_set)

% required_set = [7,15,16,33,58,59,61,32,24,27,589];
for i = 1:length(required_set)
    [new_cost,~] = find_csd_representation(required_set(i));
    set_with_costs(i,:) = [required_set(i),new_cost];
end
set_with_costs = sortrows(set_with_costs,2);
% set_with_costs = sortrows(set_with_costs,2,'descend');
required_set = set_with_costs(:,1)';
end
```

```
function [constant_set] = random_matrix_generator(say_constants,bit_width)
% bitwidth = randi(16);
% say_constants = randi(100);
constant_set = randi([-(2^bit_width) 2^bit_width],1,say_constants);
end
```

B. 2D DCT VERILOG CODE

```
module dct(input wire clk,
   input signed[31:0]x[0:7],
   output reg signed[31:0]z[0:7]);
   wire signed[31:0] sum[0:3],diff[0:3];
   wire signed[31:0] temp[0:7];
   butterfly bf1(clk,x,sum,diff);
   dct_1d    dct1(clk,sum,diff,z);
endmodule
```

```
module dct_1d(input wire clk,
   input signed[31:0]sum[0:3],diff[0:3],
   output reg signed[31:0]z[0:7]);
   wire signed[31:0]temp0[0:6],temp2[0:6],temp4[0:6],temp6[0:6];
   wire signed[31:0]temp1[0:6],temp3[0:6],temp5[0:6],temp7[0:6];
   //z0
   shiftadd_5791 sh01(clk,sum[0],temp0[0]);
   shiftadd_5791 sh02(clk,sum[1],temp0[1]);
   shiftadd_5791 sh03(clk,sum[2],temp0[2]);
   shiftadd_5791 sh04(clk,sum[3],temp0[3]);
   add          add01(clk,temp0[0],temp0[1],temp0[4]);
   add          add02(clk,temp0[2],temp0[3],temp0[5]);
   add          add03(clk,temp0[4],temp0[5],temp0[6]);
   shift_left_28 shL01(clk,temp0[6],z[0]);
   //z2
   shiftadd_7567 sh21(clk,sum[0],temp2[0]);
   shiftadd_3134 sh22(clk,sum[1],temp2[1]);
   shiftadd_3134 sh23(clk,sum[2],temp2[2]);
   shiftadd_7567 sh24(clk,sum[3],temp2[3]);
   add          add21(clk,temp2[0],temp2[1],temp2[4]);
   add          add22(clk,temp2[2],temp2[3],temp2[5]);  //
   sub          add23(clk,temp2[4],temp2[5],temp2[6]);
   shift_left_28 shL21(clk,temp2[6],z[2]);
   //z4
   shiftadd_5791 sh41(clk,sum[0],temp4[0]);
   shiftadd_5791 sh42(clk,sum[1],temp4[1]);
   shiftadd_5791 sh43(clk,sum[2],temp4[2]);
   shiftadd_5791 sh44(clk,sum[3],temp4[3]);
   sub          add41(clk,temp4[0],temp4[1],temp4[4]);
   sub          add42(clk,temp4[2],temp4[3],temp4[5]); //
   sub          add43(clk,temp4[4],temp4[5],temp4[6]);
```

```
shift_left_28 shL41(clk,temp4[6],z[4]);
//z6
shiftadd_3134 sh61(clk,sum[0],temp6[0]);
shiftadd_7567 sh62(clk,sum[1],temp6[1]);
shiftadd_7567 sh63(clk,sum[2],temp6[2]);
shiftadd_3134 sh64(clk,sum[3],temp6[3]);
sub          add61(clk,temp6[0],temp6[1],temp6[4]);
sub          add62(clk,temp6[2],temp6[3],temp6[5]);
add          add63(clk,temp6[4],temp6[5],temp6[6]);
shift_left_28 shL61(clk,temp6[6],z[6]);
//z1
shiftadd_8033 sh11(clk,diff[0],temp1[0]);
shiftadd_6810 sh12(clk,diff[1],temp1[1]);
shiftadd_4549 sh13(clk,diff[2],temp1[2]);
shiftadd_1597 sh14(clk,diff[3],temp1[3]);
add          add11(clk,temp1[0],temp1[1],temp1[4]);
add          add12(clk,temp1[2],temp1[3],temp1[5]);
add          add13(clk,temp1[4],temp1[5],temp1[6]);
shift_left_28 shL11(clk,temp1[6],z[1]);
//z3
shiftadd_6810 sh31(clk,diff[0],temp3[0]);
shiftadd_1597 sh32(clk,diff[1],temp3[1]);
shiftadd_8033 sh33(clk,diff[2],temp3[2]);
shiftadd_4549 sh34(clk,diff[3],temp3[3]);
sub          add31(clk,temp3[0],temp3[1],temp3[4]);
add          add32(clk,temp3[2],temp3[3],temp3[5]); //
sub          add33(clk,temp3[4],temp3[5],temp3[6]);
shift_left_28 shL31(clk,temp3[6],z[3]);
//z5
shiftadd_4549 sh51(clk,diff[0],temp5[0]);
shiftadd_8033 sh52(clk,diff[1],temp5[1]);
shiftadd_1597 sh53(clk,diff[2],temp5[2]);
shiftadd_6810 sh54(clk,diff[3],temp5[3]);
sub          add51(clk,temp5[0],temp5[1],temp5[4]);
add          add52(clk,temp5[2],temp5[3],temp5[5]); //
add          add53(clk,temp5[4],temp5[5],temp5[6]);
shift_left_28 shL51(clk,temp5[6],z[5]);
//z7
shiftadd_1597 sh71(clk,diff[0],temp7[0]);
shiftadd_4549 sh72(clk,diff[1],temp7[1]);
```

```
   shiftadd_6810 sh73(clk,diff[2],temp7[2]);
   shiftadd_8033 sh74(clk,diff[3],temp7[3]);
   sub          add71(clk,temp7[0],temp7[1],temp7[4]);
   sub          add72(clk,temp7[2],temp7[3],temp7[5]); //
   add          add73(clk,temp7[4],temp7[5],temp7[6]);
   shift_left_28 shL71(clk,temp7[6],z[7]);
endmodule
```

```
module butterfly(input wire clk,
   input signed[31:0] x[0:7],
   output reg signed[31:0] sum[0:3],diff[0:3]);
always @(posedge clk)
begin
  sum[0]=x[0]+x[7];
  sum[1]=x[1]+x[6];
  sum[2]=x[2]+x[5];
  sum[3]=x[3]+x[4];
  diff[0]=x[0]-x[7];
  diff[1]=x[1]-x[6];
  diff[2]=x[2]-x[5];
  diff[3]=x[3]-x[4];
end
endmodule
```

```
module add(input wire clk,
   input signed[31:0]x1,x2,
   output reg signed[31:0]y1);
always @(posedge clk)
y1 = x1+x2;
endmodule
```

```
module sub(
   input wire clk,
   input signed[31:0]x1,x2,
   output reg signed[31:0]y1);
always @(posedge clk)
y1 = x1-x2;
endmodule
```

```
module shiftadd_1597(input wire clk,
   input signed [31:0] x1,
   output reg signed [31:0] y1);
always @(posedge clk)
```

```
   y1=((((x1<<<4)-x1)<<<1)+((x1+(x1<<<5))<<<5)+((x1<<<9)-x1)) ;
endmodule
```

```
module shiftadd_3134(input wire clk,
   input signed [31:0] x1,
   output reg signed [31:0] y1);
always @(posedge clk)
   y1=((((x1+(x1<<<5))<<<5)+((x1<<<9)-x1))<<<1) ;
endmodule
```

```
module shiftadd_4549(input wire clk,
   input signed [31:0] x1,
   output reg signed [31:0] y1);
always @(posedge clk)
   y1=((((x1<<<3)+x1)<<<9)-((((x1<<<4)-x1)<<<2)-x1)) ;
endmodule
```

```
module shiftadd_5791(input wire clk,
   input signed [31:0] x1,
   output reg signed [31:0] y1);
always @(posedge clk)
  y1=(((x1+(x1<<<5))<<<7)+((x1+(x1<<<5))<<<5)+((x1<<<9)-x1));
endmodule
```

```
module shiftadd_6810(input wire clk,
   input signed [31:0] x1,
   output reg signed [31:0] y1);
always @(posedge clk)
   y1=((((((x1<<<4)-x1)<<<1)+(((x1+(x1<<<5))<<<5)+((x1<<<9)-x1)))+(((x1<<<7)-((x1<<<4)-
x1))<<<4))<<<1) ;
endmodule
```

```
module shiftadd_7567(input wire clk,
   input signed [31:0] x1,
   output reg signed [31:0] y1);
always @(posedge clk)
   y1=((((((x1<<<4)-x1)<<<2)-x1)<<<7)+((x1<<<4)-x1)) ;
endmodule
```

```
module shiftadd_8033(input wire clk,
   input signed [31:0] x1,
   output reg signed [31:0] y1);
always @(posedge clk)
   y1=((((((x1<<<4)-x1)<<<2)-((x1+(x1<<<5))))<<<9)-
(((x1+(x1<<<5))<<<7)+((x1+(x1<<<5))<<<5)+((x1<<<9)-x1)));
endmodule
```

```
module shift_left_28(input wire clk,
   input signed[31:0]Q1,
   output reg signed[31:0]Q2);
always @(posedge clk)
   Q2 = Q1>>>13 ;
endmodule
```

```
module dct_tb2;
reg signed[31:0]x[0:7];
wire signed[31:0]z[0:7];
reg signed[31:0]QQ1[0:7][0:7];
reg clk;
localparam period = 10;
dct tb1(clk,x,z);
initial #90 $finish;
initial begin clk = 0;forever #1 clk = ~clk; end
initial fork
begin
x <= {156,160,156,160,156,155,156,159};
#period QQ1[0] <= z;
x <= {159,154,159,154,153,155,153,159};
#period QQ1[1] <= z;
x <= {158,157,158,157,155,155,157,156};
#period QQ1[2] <= z;
x <= {155,158,155,158,159,157,156,158};
#period QQ1[3] <= z;
x <= {158,157,158,157,159,156,153,156};
#period QQ1[4] <= z;
x <= {156,159,156,159,155,159,155,159};
#period QQ1[5] <= z;
x <= {159,158,159,158,156,152,154,157};
#period QQ1[6] <= z;
x <= {158,158,158,158,155,158,155,161};
#period QQ1[7] <= z;
end
join
endmodule
```

```
module dct_tb4;
reg signed[31:0]x[0:7];
wire signed[31:0]z[0:7];
reg signed[31:0]QQ1[0:7][0:7];
reg clk;
localparam period = 10;
dct tb1(clk,x,z);
initial #90 $finish;
initial begin clk = 0;forever #1 clk = ~clk; end
initial fork
begin
x <= {889,880,885,887,886,889,885,891};
#period QQ1[0] <= z;
x <= {1,3,4,-3,6,-1,9,0};
#period QQ1[1] <= z;
x <= {0,7,2,-3,-4,0,2,4};
#period QQ1[2] <= z;
x <= {-7,-5,-3,-1,0,-3,-8,-5};
#period QQ1[3] <= z;
x <= {2,2,-1,2,4,0,4,2};
#period QQ1[4] <= z;
x <= {-3,0,3,-5,-5,-3,0,-3};
#period QQ1[5] <= z;
x <= {-6,10,-1,-4,2,1,-1,5};
#period QQ1[6] <= z;
x <= {-6,1,0,-3,1,-10,2,-6};
#period QQ1[7] <= z;
end
join
endmodule
```