M.S. THESIS

Approximate Calculation of DCT for HEVC and JPEG Hardware Encoders

HEVC와 JPEG 하드웨어 부호화기를 위한 DCT의
Approximate Calculation

BY

ANISH MAHENDRA TAMSE

AUGUST 2015

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Approximate Calculation of DCT for HEVC and JPEG Hardware Encoders

HEVC와 JPEG 하드웨어 부호화기를 위한 DCT의 Approximate Calculation

지도교수 이 혁 재

이 논문을 공학석사 학위논문으로 제출함

2015 년 8 월

서울대학교 대학원

전기 컴퓨터 공학부

아 니 쉬

Anish Mahendra Tamse의 공학석사 학위논문을 인준함

2015 년 8 월

Abstract

Discrete Cosine Transform (DCT) is widely used for various image and video compression applications because of its excellent energy compaction property. DCT is computationally intensive and the calculations are parallelizable. Therefore it is often implemented in hardware for speeding up the calculation. However due to large size of DCT or multiple modules of DCT required for some applications, the hardware area taken up by DCT in image or video encoders become significant. The DCT required in most applications doesn't need to be exact. Taking advantage of this fact, here a novel approach is provided to reduce the hardware area cost of the DCT module. The DCT hardware module consists of combinational logic and memory. Both the components are reduced and the complete implementation is described. The application being aimed at is for HEVC and JPEG, however the idea is applicable to any DCT hardware implementation. Finally the degradation caused to encoded image and video in terms of BDBR is discussed and the gate count results from the synthesis is provided.

Keywords: Discrete Cosine Transform, HEVC, JPEG, Approximate DCT
Student Number: 2013-23847

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Discrete Cosine Transform or DCT is used to express a finite sequence of numbers in terms of components of cosine functions of multiple frequencies. Unlike Fourier transform, DCT doesn't use imaginary numbers for calculations. The DCT is widely used for various signal processing applications and for image and video compression. DCT is a popular choice for these applications because of its excellent signal energy compaction properties [2]. It almost approaches Karhunen-Loève transform in its compaction efficiency. Karhunen-Loève transform is the ideal transform in de-correlation sense.

Here, the focus is on image/video compression application of DCT. DCT plays a prime role in each of these applications. DCT is very computationally intensive. The size of DCT required for newer applications only keep increasing, further adding to its complexity. Therefore DCT computation is generally implemented in hardware to improve the computation time. However because of large size of DCT as in HEVC video compression standard or due to multiple DCT modules required as in JPEG image compression standard, DCT implementation in hardware requires significant area. But the DCTs for most of the applications doesn't need to be accurate. Our goal is to design an approximate DCT architecture which results in reduced area with minimal degradation in encoded image or video quality. The standards chosen here are HEVC and JPEG. HEVC is an upcoming video compression standard which is very promising. JPEG is an extremely popular image compression standard. The type of DCT required in both the cases is two-dimensional. Therefore our focus throughout will be on 2D DCT architecture.

Formally, the discrete cosine transform transform is a linear, invertible function on a set of real numbers. It is calculated as:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \quad k = 0, ..., N-1$$

The two dimensional DCT follows from the one dimensional definition. It is obtained by performing the 1D DCT once along the two dimensions.

1.1 2D DCT Hardware Module

The 2D DCT is computationally intensive. However, the calculations are highly parallizable. Hence it is frequently implemented in hardware. It is used commonly in many video and image compression standards. The calculation of DCT requires multiple independent multiplications and additions. Hence implementing the module as a whole in hardware speeds up the process multifold.

Computing 2D DCT is essentially multiplication of two matrices, an input matrix and a DCT coefficient matrix. However 2D DCT can also be computed equivalently by computing two 1D DCTs as mentioned earlier. The advantage of calculating 2D DCT in this fashion is that it requires fewer multiplication and addition operations. Therefore when it is implemented in software in this manner, it is faster. Moreover, for hardware implementation, using this method

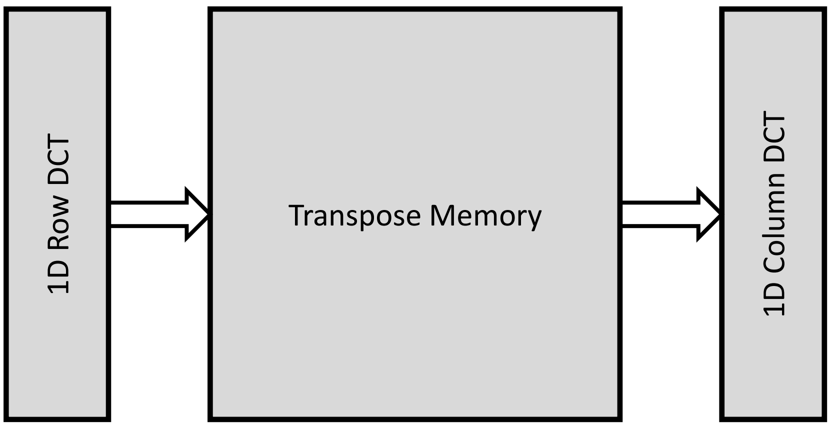

Figure 1.1 2D DCT module implemented as 1D Row DCT followed by transpose memory followed by 1D Column DCT

of computing saves area cost and enables pipelining. Thus significant number of clock cycles as well as area can be saved.

The architecture generally used for implementing a DCT module in hardware is fairly standard. In its basic form, the complete 2D DCT module can be divided into three sub-modules, namely 1D row DCT module, transpose memory and 1D column DCT module.

The 1D row DCT module is completely combinational in nature. Given N input coefficients, it calculates its N point 1D DCT. The 1D column DCT module similarly is combinational in nature too. The transpose memory comprises the non-combinational area of the complete module. It consists of registers to store the result of the row 1D DCT until it is required by the column 1D DCT. The transpose memory also contains multiplexers, the purpose of which will be described later.

Let us suppose that this module calculates the 2D DCT of an $N \times N$ block. A row of the input block is sent every clock cycle to the 1D row DCT module.

The output (1D DCT of the input row) is available at the end of the clock cycle. Since the 2D DCT size is $N \times N$, there will be $N \times 1$ input coefficients and N resulting output coefficients for every clock cycle to the 1D row module. These output coefficients at the end of the clock cycle are pushed into the transpose memory. Since there are N rows in a given block, all the row 1D DCTs will be completed at the end of N^{th} clock cycle. At the end of N clock cycles the transpose memory hence contains all the row DCT coefficients.

In the next N cycles, the column 1D DCT is calculated for the coefficients inside the transpose memory. Initially, the first coefficient of the each row DCT result is sent to the column DCT module. In the next cycle, the second coefficient of each row is sent and so on till all the coefficients are eliminated. The coefficients obtained from the 1D column module are the required 2D DCT coefficients, which can be then forwarded to the appropriate module.

It can be observed that the fashion in which the coefficients are written and read from the transpose memory is unique. While writing into the memory, the rows are written as a whole whereas while reading, one element from each row is read. To enable this form of input and output from the transpose memory, the coefficients within the transpose memory are moved every cycle.

The method of moving the coefficients inside the memory is described below. We will study the case of 4×4 DCT for ease of representation (the size of the DCT required in HEVC varies from 4×4 to 32×32).

The transpose memory shown in Figure 1.2 consists of 4×4 register array. The bit width of the registers depends on the application. DCTs in HEVC require 16 bit registers. The diagram represents the movement of the coefficients within the memory. The first row of the figure shows the result of row DCT output pushed into the memory. At the beginning of next cycle, all the coefficients move to the respective register in the next row. Once all the rows are filled, all

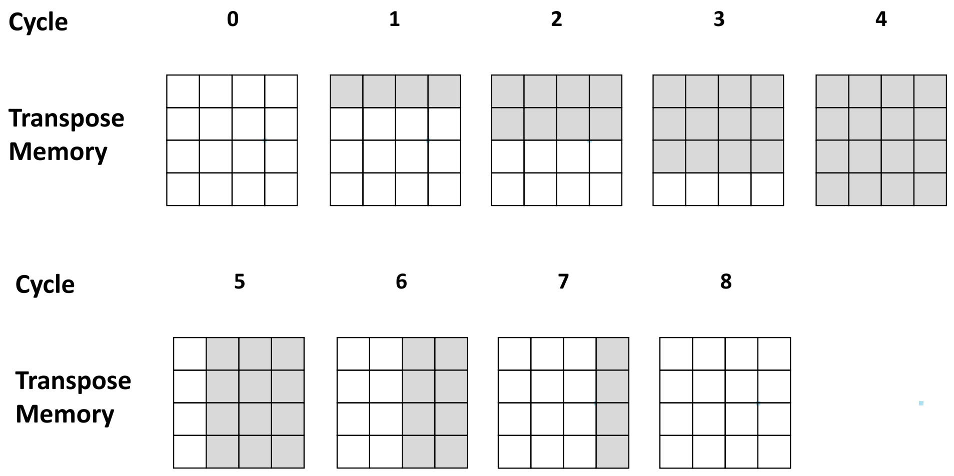

Figure 1.2 Representation of movement of coefficients within the transpose memory. The shaded cells represent the filled memory locations.

the coefficients start to move in the horizontal direction. The coefficients leaving the transpose memory are the inputs to the column 1D transform module.

As described above, the transpose memory consists of registers. The movement of the coefficients is facilitated by using multiplexers. Each register receives the coefficient for the immediate next cycle either from the register above it or to the left of it. Hence all the registers except the ones in the first row or the first column need 2×1 multiplexers.

1.1.1 Pipelining the process

It can be seen that in the process described earlier, the column DCT module is idle while the row DCT is being calculated and vice versa. The transpose memory is also not utilized fully all of the time. These issues can be addressed by pipelining the complete process. For a single sized DCT module, the process can be perfectly pipelined, such that both the 1D DCT modules are used all the time and all of the registers in transpose memory are used all the time.

The pipelining architecture needs a minor modification to be made to the original architecture. It requires additional hardware area but the overhead is negligible. Earlier, the input to the transpose memory was always coming into the first row of the memory. For pipelining however, we need to add a multiplexer such that we can redirect the input to either the first row or the first column of the memory. Similarly, we need another multiplexer at the output to control whether the output is read from the last column (like earlier) or from the last row.

With these modifications, the transpose memory coefficient pattern is as shown in Figure 1.3.

Here, the white block in the cycles 0 to 3 represents empty memory locations. The memory is fully filled in clock cycle 4. From the 5th cycle onward, the 1D row transform coefficients of the next 4x4 input is pushed into the first column of the transpose memory. Hence the row transform module is still engaged and all the memory locations are still being used. Similarly, in the 9th cycle, we can see the output from the row transform of the third input block arriving into the first row of the transpose memory.

1.2 Approximate DCT

For codec applications in image compression and video compression, the DCT doesn't need to be exact. DCT is only used on the encoder side, therefore it has the option to encode the coefficients differently. Furthermore, in HEVC, the standard reference implementation specifies an integer DCT matrix, which is already an approximation of the original DCT. In JPEG image compression standard, even though the encoder can make use of a floating point DCT,

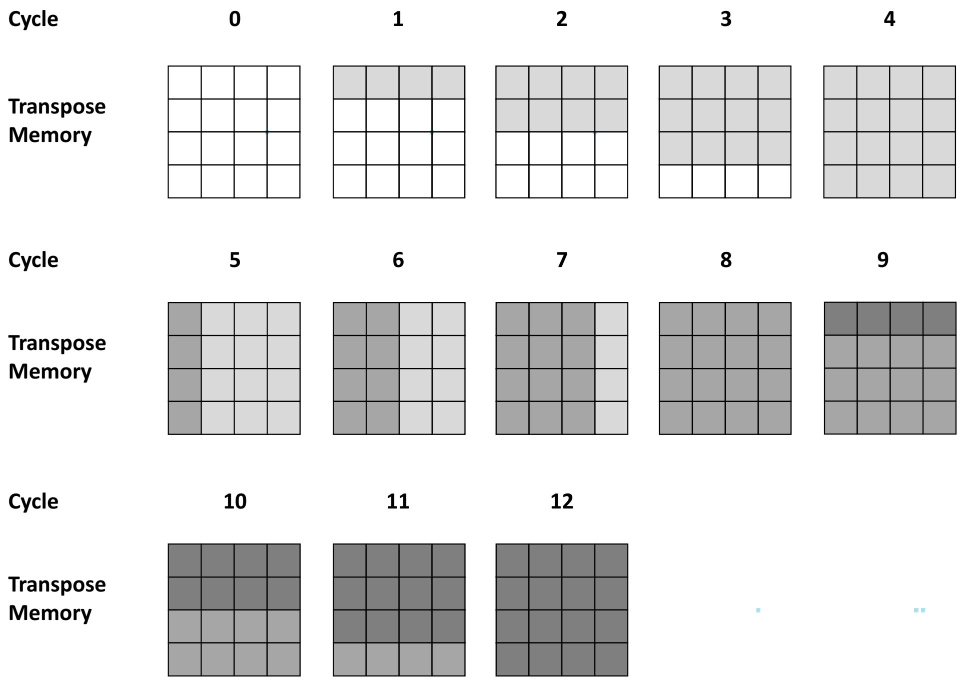

Figure 1.3 Pattern of movement of coefficients within transpose memory when pipelined. The different shades of grey represent coefficient from different input blocks.

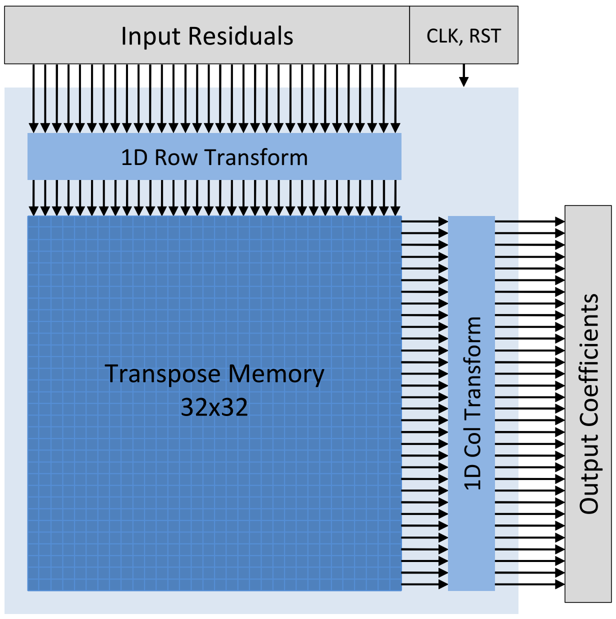

Figure 1.4 The complete 32×32 DCT module architecture.

most encoders use integer/fixed point DCT which is an approximation of the original DCT. These approximate fixed point DCTs are used because they avoid floating point operations completely at negligible loss in final encoding quality. In JPEG there is an option to use the floating point DCT but the difference between the image compressed with fixed point DCT and floating point DCT is barely perceivable.

If we further loosen the criteria of exact integer DCT requirement, we can modify the architecture of the DCT module so as to save area cost of the hardware. We must however make sure that the effect it has on the resulting encoded image/video quality is minimal.

Chapter 2

Related Works

Discrete cosine transform has been studied for a long time. Some of the earliest works on discrete cosine transform are [3, 4, 5, 6, 7, 8]. Most of the early works were related to studying the properties of DCT and reducing the complexity of calculating the DCT. Once its high energy compaction efficiency was studied [2], it became a popular choice for many applications. The main target applications were speech signal processing and image/video compression.

Because of the computationally intensive and parallelizable nature of DCT, it is implemented in hardware. [9, 10] and [11] discuss efficient architecture for implementing it in hardware. [9] shows an implementation for 8×8 DCT aimed for realtime image and video compression applications. Row-parallel architecture is used. They save area by reducing the number of 1D DCT modules required at the cost of increased result latency. [10] removes all the multipliers from the architecture and uses distributed arithmetic architecture. They minimize the number of additions required by exploiting the timing property of the DCT. The focus of [11] is for realtime image processing applications.

They describe an associative processor architecture. The architecture uses an associative memory and multiple processors, one for each pixel.

The architectures are further optimized specifically for target application in H.264 in [12, 13]. This is possible since the H.264 standard specifies an integer coefficient matrix for inverse DCT. Therefore ideally, the forward DCT should be the inverse of the provided transform, hence the exact integer coefficients can be known. Since the values of the integer constants are known, the combinational logic of the generic architecture can be further optimized. [12] discusses two architectures, one optimized for hardware area and the other one for result throughput. Both the architectures implement the exact DCT for the IDCT specified in the standard. The two architectures differ in the number of 1D DCT modules used for the implementation. The FPGA and ASIC implementation of the architectures are discussed. [13] describes an architecture wherein the different sizes of DCT required are combined into a single architecture. Similarly the DCT architectures specific to HEVC are discussed in [14, 15, 16]. The architectures of [14, 15] are similar, differing in their implementation of the multiplication logic. Both avoid the use of complete multipliers and use adders and shifters instead. The architecture of [16] combines the HEVC DCTs of different sizes into a single architecture.

Our focus is on approximate computation of DCT. Works related to approximate DCT can be classified into two types.

- *Algorithmic Modification*: Here, the algorithm for the computation of DCT is modified, the main aim being reducing the total arithmetic operations required. They are mainly aimed to speed up software computatation of DCT, although some benefit may translate to hardware too due to reduced number of adders/multipliers required. Two main approaches have been used to achieve algorithmic modification of DCT, namely, frequency selection and accuracy selection. Frequency selection approach tries to eliminate the computation of the less significant frequencies and the accuracy selection approach tries to reduce the accuracy with which each coefficient is computed. [17] uses the first approach to calculate select few components. It also shows analysis for the error introduced with the method. [18] is an example of the second approach. [19] combines the two approaches. The idea is to mask the error of each coefficient of the DCT output by making sure that the error introduced in the quantizaiton step is larger. Hence depending on the quantizaiton parameter, a balance of the two approaches is provided.

- *Hardware Modification*: The emphasis here is on modifying the architecture directly. The changes may not have any significant benefit if this modified DCT is implemented in software. [20] discusses low power approximate architecture. They perform analysis of experimental results of lowering power while losing accuracy. A point is then chosen for optimal tradeoff between the two.[21] achieves reduction in hardware area by changing the partial butterfly structure in hardware so that it uses only 14 adders for $8\times$ 8 DCT. [22] aims at reducing DCT hardware for H.264, the focus here too being on combinational logic reduction. Their approach for the reduction is by decomposing the 8 point DCT to 4 point DCT.

Our work belongs to the second type, i.e. aimed at hardware modification. Our aim is to provide an approximate DCT implementation for HEVC/JPEG with main focus on the non-combinational area (transpose memory) reduction. We try to reduce the amount of transpose memory required for the complete module. To the best of our knowledge, there hasn't been any other work which aims at reducing the transpose memory in a DCT architecture.

Chapter 3

The Moving Window Idea for Bit-Width Reduction

An integer when stored in binary format is usually stored in 2's complement format. For the discussion example, consider 16 bit binary integer with the most significant bit to the left and the least significant bit to the right. Our aim is to approximately represent the same integer with fewer bits, eight in this case, and hence effectively save memory.

Let us consider the case of positive integer. A 16 bit positive integer will have n prefix 0s, where $1 \leq n \leq 16$ and the remaining bits will be 1s and 0s. $n = 1$ represents the case where the number is greater than 2^{14} and $n = 16$ when the number is 0. Therefore if we know the location of the most significant bit which is 1 in the binary representation of the number, we automatically know all the bits to the left of it are zeroes. Let the most significant bit which is a 1 be the k^{th} bit in the n bit integer. Then, if we know the value of k and all the bits from k to LSB, we can fully reconstruct the original integer.

As mentioned, we can recover the original number by storing all the bits

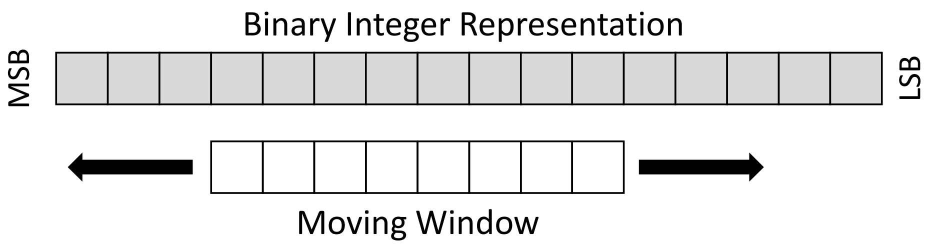

Figure 3.1 Example of an 8 bit moving window and 16 bit integer. The moving window can be aligned in 8 different positions and can store one of the corresponding eight possible binary sequence.

from the k^{th} bit to the LSB along with the position of the most significant 1. The number of bits between the k^{th} bit to the LSB can vary, hence the number of bits to be stored can vary. Next we impose an additional constraint that the total number of bits we can store is W. For the discussion, let $W = 8$. We can store any eight consecutive bits occurring in the binary representation of the number. Since we store consecutive bits and the starting point of these eight bits can be varied, this is called a moving window. If we know the contents of the window and the position of the window, we can approximately recover the original integer. To recover the original number from the contents of the window and its position, the method followed is as follows:

- We know the continuous sequence of eight bits and their positions (deduced from the starting position of the window). Hence the corresponding 8 bits in the number to be recovered are filled with the bits from the window.
- The remaining unknown bits to the left side if any, are filled with 0s.
- The remaining unknown bits to the right if any, are fill with 1 followed

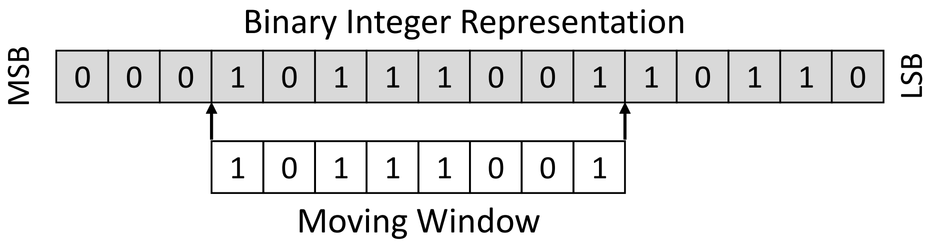

Figure 3.2 The ideal window alignment for the shown binary integer. The recovered binary sequence when the window position is known will be 0001011100110000.

by 0s to fill completely.

Eight bit window is not always enough to completely cover the original 16 bits. If the magnitude of the integer is less than 2^8, it can be fully recovered with the correct window position (since the number can be represented within 8 bits). However, whenever the number exceeds 2^8, we have to decide which bits will be accommodated in the window and which ones will be discarded. Varying the position of the window, we can either discard some bits towards the MSB side or the LSB side or both. When we recover the number, the recovered number will be a good approximation if the bits lost were on the LSB side rather than the MSB side.

If the bits (non-zero) are discarded on the MSB side, the recovered number will be vastly different than the original number (at least by an order of magnitude of 2)

However if the bits discarded (non-zero) were on the LSB side, the error is contained. The original number and the recovered number in this case follow the following inequality.

$$|Integer_{recovered} - Integer_{original}| \leq 2^{(LSB\, bits\, ignored)}$$

Table 3.1 The relation between the window size and the maximum error possible when an integer is encoded using the ideal window position.

Window Size	Error Percentage
2	50.00%
3	25.00%
4	12.5%
5	6.25%
6	3.13%
7	1.56%
8	0.78%

Thus for the ideal window position, it should be made sure that the most significant 1 bit is included in the window and the number of LSB bits ignored is minimized. This happens when the window starts at the position of the first bit which is 1 from the MSB side.

Furthermore, for ideal window position,

$$\frac{|Integer_{recovered} - Integer_{original}|}{|Integer_{original}|} \leq \left(\frac{1}{2}\right)^{(window\,size)}$$

i.e. the error decreases exponentially with the size of the window. The maximum case errors for some of the window sizes is summarized in Table 3.1.

In case of negative integers, everything remains the same except that instead of prefix 0s, we have prefix 1s. Thus the changes are as follows:

- For recovering the number from the window and its position, we prefix 1s to the unknown locations to the left of the window instead of 0s.
- The ideal window position starts from first 0 from MSB side instead of first 1.

Checking the MSB of the original binary representation of the integer tells us if the integer is positive or negative. The appropriate window position can then be chosen as described earlier and the position and window values can be encoded. Similarly in the recovery stage, the sign bit can be checked to determine whether the original integer was positive or negative. Once determined, the appropriate recovery procedure can be followed.

3.1 ML Recovery for Moving Window

Earlier, the method to recover the original integer from the window and its position was mentioned. We will now look at the rationale behind the logic of recovery. When the window position is such that the first 1 from MSB for positive integers or first 0 from MSB for the negative integers is lost, the number recovered will not be an approximation of the original number hence it doesn't make sense to talk about error in this case. If this is not the case, then the number would have lost bits only on the LSB side. Therefore,

$$Int_{orig} - 2^{LSB\,bits\,lost-1} \leq Int_{recovered} \leq Int_{orig} + 2^{LSB\,bits\,lost-1}.$$

The error is contained and the recovery logic tries to minimize the error on average. For all the unknown positions to the left of the window, 1s were filled if the integer was negative and 0s if it was positive. This is to comply with the 2's complement representation of the integers.

For all the unknown positions to the right of the window, we are free to choose the bits since those bits in the original number could range from 0 (all bits being 0) to $2^{LSB\,bits\,lost} - 1$ (all bits are 1). Assuming the original number to be distributed uniformly between the lowest possible number and the largest possible number, we can conclude all the trailing sequences of 1s and 0s are equally likely. And error is defined as the difference between the magnitude

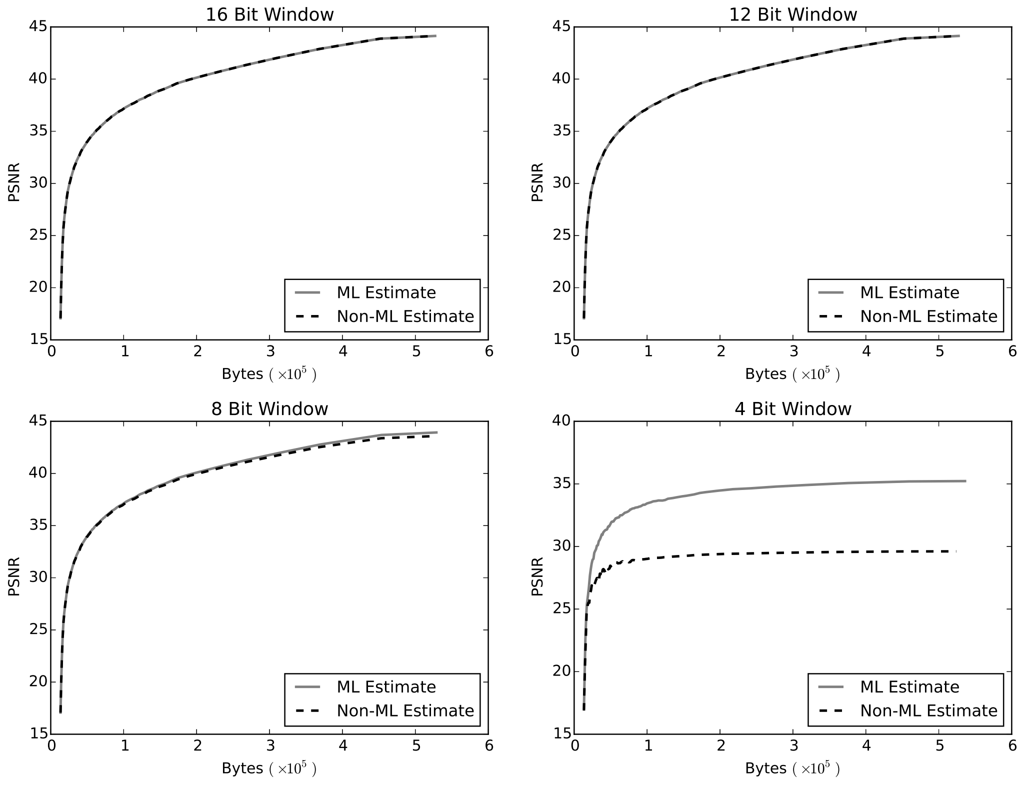

Figure 3.3 PSNR vs bytes curves for Hydrangeas image encoded using JPEG for different sized bit windows. Using ML estimate improves PSNR as is especially evident from the curve for 4 bit window.

of the trailing sequence and magnitude of the trailing sequence in the original integer. Magnitude of the trailing sequence is a constant once the recovery logic is decided. Thus for the estimate to be maximum likelihood (ML) estimate for minimizing error for an average case, we choose the middle value for recovering the trailing sequence. Since the number of possibilities is an even number, we have two equally valid candidates 1 followed by 0s or 0 followed by 1s. The first case is then decided to be the recovery logic at random.

The plot in Figure 3.3 shows two cases of moving window encoding applied to the transpose memory of DCT while JPEG compression. The first case has recovery method which doesn't use ML estimate (all the trailing positions are filled with 0s) and the second case has the recovery method which uses ML estimate. In the first case all the pixels will show an overall shift to the darker side compared to the original image. The brightness of the image encoded using ML estimate won't have this issue and will be a truer replication of the original image. This is because when not using ML estimate, all the recovered values have inherent bias to move towards one direction (lower in this case), whereas in the second case, the variations are canceled out on the average. This shows it is important that ML estimate is used for lower error.

The memory overhead for moving window implementation is the bits required to store the window position. As long as the bits required is smaller than the bits saved due to the small window size, we save bits overall. The example discussed earlier had a window of size 8 bits. This can be generalized to other sizes as follows.

Let

$$total = Total\,original\,bit\,width$$

$$window = size\,of\,the\,moving\,window\epsilon[1, total]$$

then,

$$Total\,Bits\,Required = min\{ceil(log(total - window)) + window, ceil(log(total)) + window - 1\}$$

$$Maximum\,Error\,Fraction < 0.5^{window}$$

Chapter 4

Approximate DCT for HEVC

4.1 HEVC Overview

HEVC or High Efficiency Video Codec is also known as H.265 for being the successor to the earlier video encoding/decoding standard H.264. HEVC as a video compression standard has taken over its predecessor H.264. HEVC promises up to 50% more compression efficiency, which means that HEVC can essentially deliver essentially the same quality of video at up to half the bit rate. This is especially true for higher resolution videos. This large amount of saving comes at the cost of increased encoding and decoding complexity. For example, the intra prediction in HEVC has 33 directional modes as opposed to 8 of H.264. HEVC has higher bit depth for each of the vertical and horizontal motion vectors compared to H.264, in addition to new modes for predicting the motion vector, AMVP (Advanced Motion Vector Prediction) and Merge Mode. The DCT which plays a major role in compression is also made more complex for HEVC. The DCT sizes supported for HEVC are 4×4, 8×8, 16×16 and 32×32

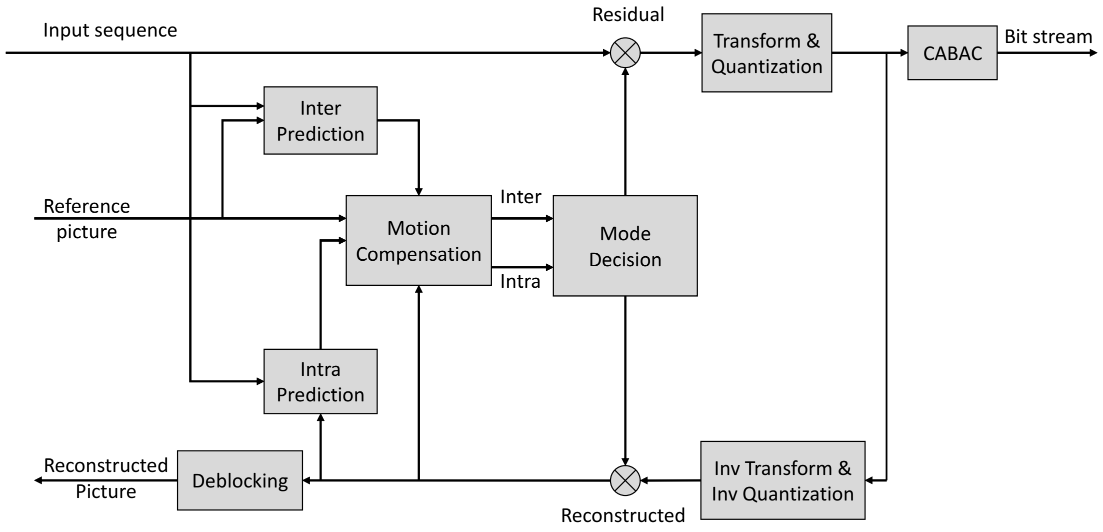

Figure 4.1 Overview of HEVC encoder structure

whereas the sizes supported for H.264 were only 4×4 and 8×8.

4.2 HEVC Encoder

The encoder structure in HEVC is as shown in Figure 4.1. The reference frame is available in the memory for encoding. The mode decision module decides whether to use inter prediction or intra prediction for every macro-block or CU (compression unit, as it is referred to in HEVC) of every frame. For intra prediction, various directional modes are tried based on the settings to find out the best one. For the inter prediction mode, various reference frames are tried and motion estimation is performed to obtain the ideal motion vector. The best mode is then sent as the output.

To decide which mode is better, the mode decision module needs to quantify a cost associated with each mode and for each of the sub-modes (directions in case of intra and motion vector in case of inter) within the mode. This job is performed by RDO (Rate Distortion Optimization) module within the mode decision module. Each of the prediction modes is associated with a rate, i.e. the bits required to encode the block using this mode and a distortion i.e. the total distortion caused if this block is used as the prediction mode. The rate distortion optimization module uses the two values to calculate the cost for each of the modes. The mode with the least cost is used as the final mode to be encoded and is given as the output of the mode decision module.

The previous reconstructed frames are kept in the memory. The reconstructed frames are used for inter prediction. Not using actual frames and using reconstructed frames makes sure that the encoder always calculates the predicted blocks using what the decoder will be using for prediction. Hence there will not be any mismatch between the encoder and decoder. Otherwise the mismatch would keep adding up with every inter-predicted frame and will result in error accumulation.

In the next step, the predicted macro-block is used to calculate the residual i.e. the difference between the values of the predicted pixels and the actual pixels. The residual is then sent as input to the transform and quantization module where the actual lossy compression takes place. Once quantized, the values of the residual can be encoded with fewer bits than earlier but the exact values of the residual pixels cannot be obtained back. The quantized values are then written into the bit-stream using lossless entropy encoding technique called CABAC (Context-Adaptive Binary Arithmetic Coding).

4.3 DCT in HEVC Encoder

Transform calculation plays a significant role in compressing videos. However, it requires a large amount of multiplication and addition operations. The DCT calculations in HEVC can take up to 30% of the total encoding time. Also, these arithmetic operations required for the transform can be highly parallelized. Because of the parallel nature and large amount of calculations, it is often implemented in hardware.

In the HEVC encoder described earlier, the DCT calculation occurs at two places.

1. *Rate Distortion Optimization module* : This module exists within the mode decision module. It is used to decide which of the various available prediction modes for the given macro-block will be optimal. Since the decoder just needs to know the prediction mode from the bit stream and not how it is calculated, HEVC doesn't specify any constraints on how this is decided. Hence the architecture of the RDO module varies with the implementation. The most ideal implementation will be the one where transform and quantization is performed which is followed by inverse quantization and inverse transform. The distortion is then calculated by SSE and rate is calculated by CABAC and then finally the cost is computed using these two values. Since this has to be repeated for every mode possible, it takes a significant amount of time. Hence various research has been performed to find methods with much less complexity which give near optimal results [23, 24, 25]. Most of the alternative methods make use of DCT and quantization but not inverse DCT and inverse quantization.

2. *Transform and Quantization module* : This module receives the macro-block residual from the mode decision module. The residual is then transformed using DCT and then quantized. The output of the quantization is forwarded to the CABAC module for encoding into the bit stream.

4.4 Approximate DCT in HEVC

As we saw earlier, DCT is used in two modules. If we analyze the role of DCT in each of the two modules, it can be noticed that the DCT need not be exact for either of the cases. The DCT module can be replaced with an approximate DCT calculation without causing any errors with the encoding. We will need to make sure the degradation resulted by the approximate nature is minimized to keep the approximate DCT usable. We look at both of the instances of DCT separately and study as to why an approximate DCT is fine in each case.

- The first module where DCT is used in is the Rate Distortion Optimization (RDO) module. The result of the DCT here need not be accurate because the RDO module doesn't need to provide exact result. In fact, most of the implementations of the RDO module don't provide the exact result as to which mode is the most optimal. The exactness is generally traded off for a large reduction in the complexity. Thus using an approximate DCT does not result in erroneously encoded video.

- The second module where the DCT is used in is the transform and quantization module, just before the bit stream encoding takes place. The result of this module is critical since its output coefficients are directly encoded into the bit stream of the final output. However, the DCT used here can still differ from the DCT used in other encoders without causing errors in encoding. As long as the inverse DCT used is the same throughout (in the encoder and all the decoders), there will not be any mismatch between the reconstructed frames. Hence there will not be any error accumulation. Another way to think of this is to say that originally, the loss incurred in the encoding was only due to quantization whereas now there will be some information lost in the forward DCT step too. In fact, the HEVC

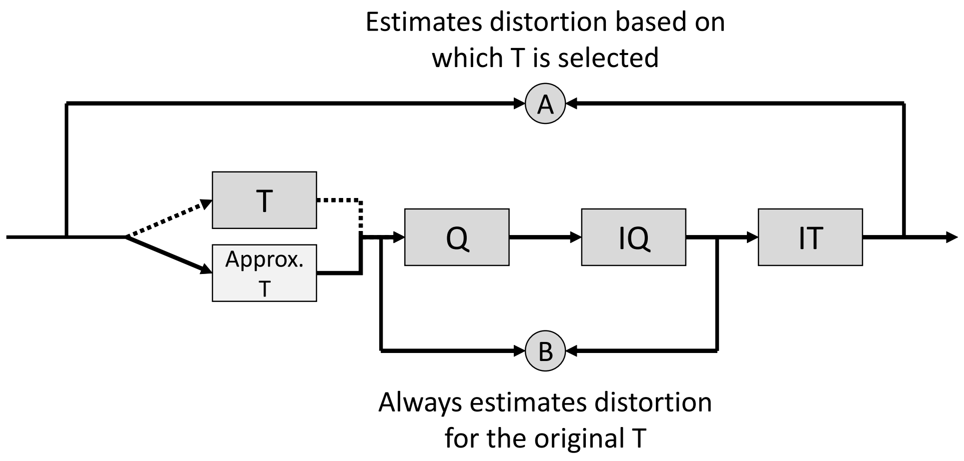
Estimates distortion based on which T is selected
A
T
Approx. T
Q
IQ
IT
B
Always estimates distortion for the original T

Figure 4.2 Distortion calculation in complete RDO (at A) vs simplified RDO (at B). Simplified RDO doesn't use inverse transform for calculating the distortion.

standard specifies just the inverse DCT matrix. The forward DCT used need not be the exact inverse of the inverse DCT matrix.

The two DCT modules, one in RDO and the one in transform and quantization module are not functionally independent of each other. The result of the RDO module affects the final prediction mode and hence the residual input given to the transform and quantization module. Therefore it is necessary that the RDO and the transform and quantization module are in sync, i.e. the RDO correctly approximates the distortion and rate of what will be encoded. This is studied in more detail.

The RDO module calculates the cost from rate and distortion. The rate calculation is based on the quantized coefficients which will always provide correct approximation of the final encoded rate. The distortion calculation however is generally implemented in two different ways. The first method calculates transform followed by quantization followed by inverse quantization and finally inverse transform. The distortion is then calculated by calculating as the sum of squared errors between the result and the original residuals. The second method uses only the inverse quantization result to calculate the distortion. This provides an approximate result of the distortion but hardware cost is reduced since inverse transform is not required.

The schematic in Figure 4.2 shows the distortion calculation part of the RDO module. Initially let us consider the transform module used is the exact one. A and B are the two methods of calculating the distortion. At both A and B, the sum of squared errors are calculated. The distortion calculated at A is the exact distortion and B is an approximate method of calculating the same distortion (called as simplified distortion henceforth). The advantage with simplified distortion is that another inverse transform module is not required.

Now we replace the forward transform module with an approximate forward transform module. Now, the inverse transform module in the final stage no longer corresponds to the inverse of the forward transform used. Therefore the distortion becomes worse because there is distortion added not only by quantization but also by approximate nature of the transform. The distortion calculation at A correctly reflects this change. However, the distortion calculated at B still estimates the old distortion (for the exact forward transform) albeit with lesser accuracy than before. Thus we have two cases for using the approximate transform module:

Case 1: Use approximate transform module only for RDO. The encoding process will be using the exact transform.

Case 2: Use approximate transform module for both RDO and for encoding process. Include the inverse transform module for distortion calculation, i.e. RDO will not use simplified distortion.

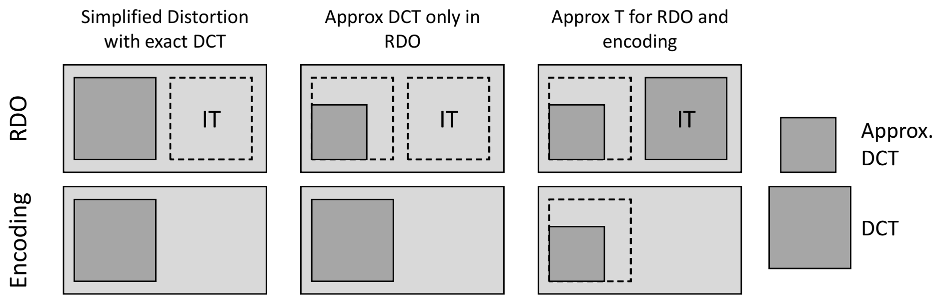
Simplified Distortion with exact DCT
Approx DCT only in RDO
Approx T for RDO and encoding
RDO
Encoding
IT
IT
IT
Approx. DCT
DCT

Figure 4.3 The three feasible cases of RDO transform module and encoding transform module combinations. All the cases try to minimize area compared to the original implementation where RDO contains both transform and inverse transform module and encoding transform is exact.

The two cases along with a third case of using exact forward transform at both places are shown in Figure 4.3.

The first column in Figure 4.3 is the reference case for comparison; the second column is the Case 1 and the third column is the Case 2. Just from this, it cannot be concluded which of the two cases will have lesser area and/or which case will perform better. It will depend on the accuracy of the approximate transform and the area saving it imparts. Suppose the approximate transform estimates the actual transform without any error, then case 2 will not cause any degradation. And the degradation in case 1 will be that of the reference case.

Here we will always assume the case of RDO using the inverse transform module. Therefore, the approximate transform module designed will be used in both the places where DCT occurs, i.e. the RDO and in the transform and quantization module just before the bit stream encoding. This case is chosen to maintain generality, i.e. the approximate DCT can be used on unmodified

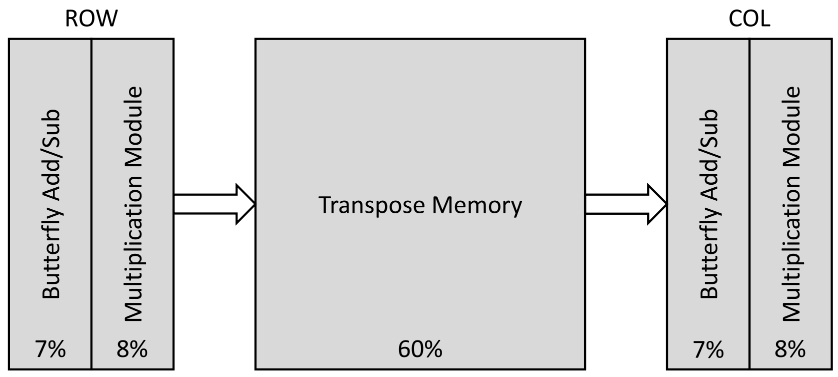

Figure 4.4 Approximate area composition of the components for 32×32 DCT in HEVC.

HEVC encoder. Otherwise we need to tailor the modifications separately for different types of RDO.

4.4.1 The three components of the DCT module

The DCT module can be roughly divided into three components:

- *Butterfly Adders / Subtractors:* The calculation of 1D DCTs for rows/columns require each row/column to be multiplied with a coefficient array of equal dimension. Therefore for calculating 32 point 1D DCT, we require a total of 32×32 multiplications. In case of HEVC, the integer DCT coefficients are so chosen that the number of these multiplications can be greatly reduced by using partial butterfly algorithm. Butterfly adders/subtractors shown in Figure 4.4 is the component which implements this logic. This component is used at two places, once for row 1D DCT and once for column 1D DCT. The area contributed by the two is roughly 14% of the total area.

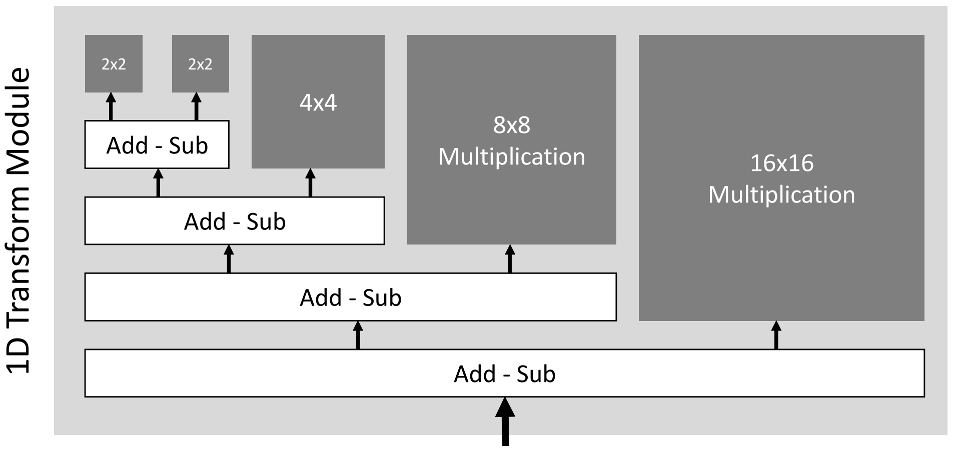

Figure 4.5 Internal architecture of the 32 point 1D DCT module in HEVC.

- *Multiplication Module:* The result of the partial butterfly addition/subtraction is sent to the multiplication module as shown in the Figure 4.5. Each of the coefficient which is sent to the multiplication module needs to be multiplied with a set of pre-defined coefficients. Using algorithm for multiple constant multiplication (MCM), the multipliers can be completely eliminated and the whole logic can be implemented with just adders and shifters. After this optimization the area contributed by the two multiplication modules (for row and column DCT) to the total area is roughly 16%.

- *Transpose Memory:* This composes the non-combinational area for the complete module. It also contributes to the combinational area because of the multiplexers attached to each register to facilitate coefficient movement within the memory. Transpose memory stores the result of row 1D DCT before forwarding it to the column 1D DCT module. The area contributed by this transpose memory towards the total area is roughly 60%.

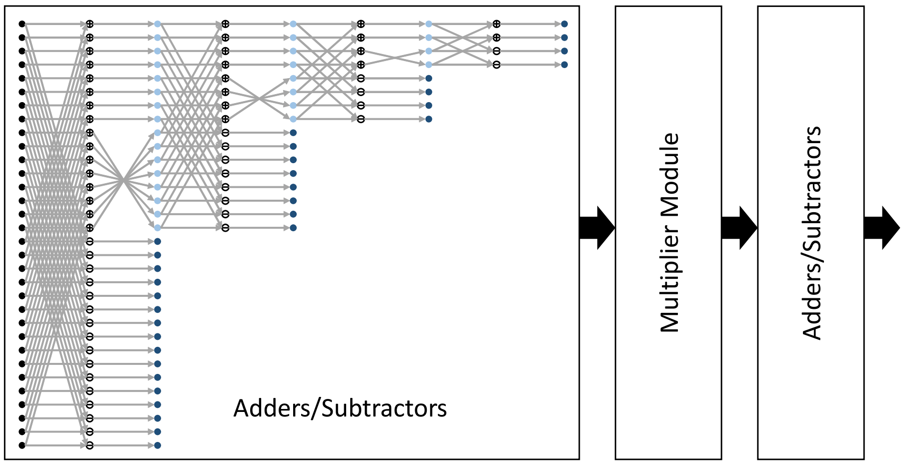

Figure 4.6 Internal structure of 1D row/column DCT module. The partial butterfly structure is shown in detail.

We observe that none of the components is negligible in terms of area, and therefore for overall reduction in the hardware cost, we optimize each of the components.

4.4.2 Optimizing Partial Butterfly Adder/Subtractors

A more detailed view of the 1D row/column DCT module is shown in Figure 4.6. The adders/subtractors occurs before and after the multiplier module. The adders and subtractors before the multiplier module compose the partial butterfly logic and the ones after the multiplier are used to calculate the final transformed coefficients from the multiplier result.

Any modification to the partial butterfly structure causes a ripple effect. The resulting approximation errors flow through the multiplier module and the latter adders and subtractors. The errors get amplified in the multiplier module resulting in large degradation. Therefore approximations can be performed only for the latter set of adders and subtractors. The latter set of adders and subtractors implement the logic shown below for computing the first coefficient of the row DCT result. The other coefficients involve similar computations.

$X_{temp1} = (90\times O_0+90\times O_1+88\times O_2+85\times O_3+82\times O_4+78\times O_5+73\times O_6+67\times O_7+61\times O_8+54\times O_9+46\times O_{10}+38\times O_{11}+31\times O_{12}+22\times O_{13}+13\times O_{14}+4\times O_{15})$

$X_1 = (X_{temp1} + add)/2^{shift}$

The multiplication results required for computing X_{temp1} are obtained from the multiplication module. The results of the multiplication module is added among themselves as detailed and X_{temp} is then rounded. The rounding is performed by adding a constant value (*add* term) and right shifting by another constant amount. The right shifting doesn't require any hardware cost. However rounding adders contribute significant area since the result of multiplication is a 16 bit integer. This addition of the *add* term is eliminated and the rounding is performed just by shifting at the cost of negligible loss in the encoded video quality.

4.4.3 Optimizing the multiplication module

As seen in Figure 4.5, there are four different sizes of multiplication modules in the DCT module, namely 16 point, 8 point, 4 point and 2 point. There are 16, 8, 4 and 2 inputs to the four module respectively. Since the result for these inputs need to be computed simultaneously, each of the multiplication modules consists of multiple identical sub-modules. The 16 point module consists of sixteen identical sub-modules. Each of those sub-modules, given a variable input provides the product of that input with 16 predefined coefficients. Similarly the other sub-modules of smaller sizes provide product of the input with respective number of predefined coefficients. The number of gates required by each type of sub-modules is summarized in Table 4.1.

There are 16, 8, 4 and 2 identical sub-modules within the 16 point, 8 point, 4

Table 4.1 Gate count for sub-modules of the multiplier block

Multiplication Sub-Module Size	Gate Count
16 point	1077
8 point	633
4 point	350
2 point	278

Table 4.2 Gate count for different modules of the multiplier block

Multiplication Sub-Module Size	Number of Sub-Modules	Effective Gate Count	Percentage of Total Area
16 point	16	17232	71.1%
8 point	8	5064	20.9%
4 point	4	1400	5.8%
2 point	2	556	2.2%

point and 2 point multiplication modules respectively. Hence the effective areas of each multiplication module is as shown in the Table 4.2.

We observe that the 4 point and 2 point multiplication sub-modules contribute negligible areas. Therefore we focus the optimization on the first two. The set of coefficients for which we need to provide the products of the input with for 16 point multiplication module are as follows:

90, 90, 88, 85, 82, 78, 73, 67, 61, 54, 46, 38, 31, 22, 13 and 4.

The set of coefficients for which we need to provide the products of the input with for 8 point multiplication module are as follows:

90, 87, 80, 70, 57, 43, 25 and 9.

4.4.3.1 Multiple Constant Multiplication (MCM)

Given a constant, its product with a variable can be implemented with just shifting and addition operations. A full multiplier is not required. The Multiple Constant Multiplication or MCM is the problem of having given a variable and a set of constants, providing the product of the variable with each of the constants with least number of adders. The problem as such is NP complete and various heuristic algorithms have been.

Here we have an instance of MCM problem where we are given a variable input and we have to provide the product with sixteen coefficients for 16-point module and similarly for 8 point module. The algorithm used for our purpose is [1]. Using the algorithm, the 16-point multiplication module can be implemented with thirteen adders/subtractors and twenty three shift operations. The flow of the additions/subtractions and shifting is shown in Figure 4.7.

The 8-point multiplication module can be similarly implemented with eight add/subtract and eleven shift operations as shown in Figure 4.8.

4.4.3.2 Approximate MCM

In our case, exact result of the multiplication is not needed. Hence we extend the original MCM problem to approximate MCM problem. The idea is to reduce the number of adders even further at the cost of getting an approximate result. Our aim is to modify the set of constants for which the products have to be found in such a manner that the number of adders required is fewer than before and the degradation caused is minimized.

When we have two sets of coefficients which result in same number of adders, we will need to determine which set causes less degradation. This can be performed by encoding a set of benchmark videos with the modified coefficient

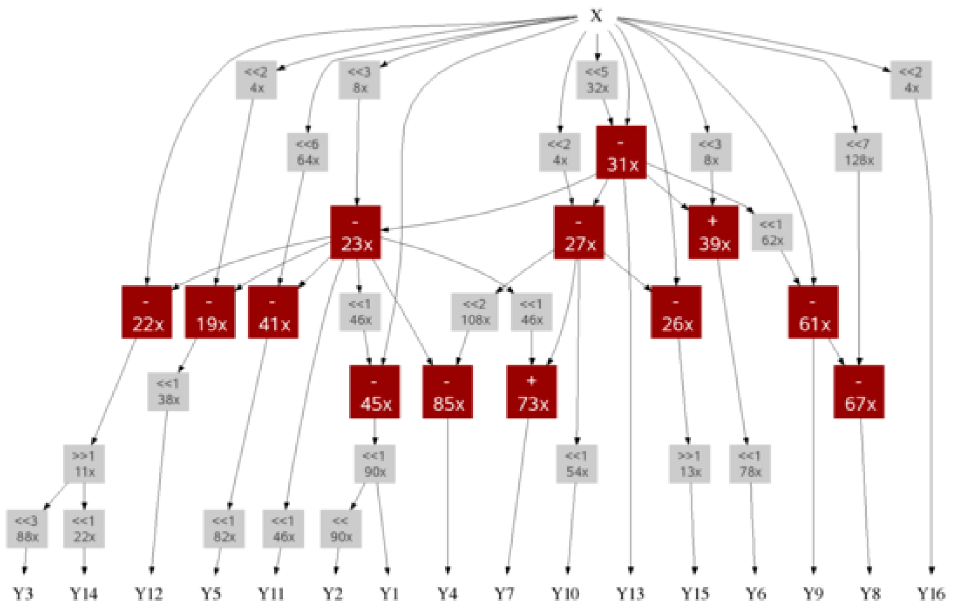

Figure 4.7 The exact implementation of 16 point multiplication sub-module using MCM algorithm [1]. There are 13 add/subtract operations and 23 shift operations.

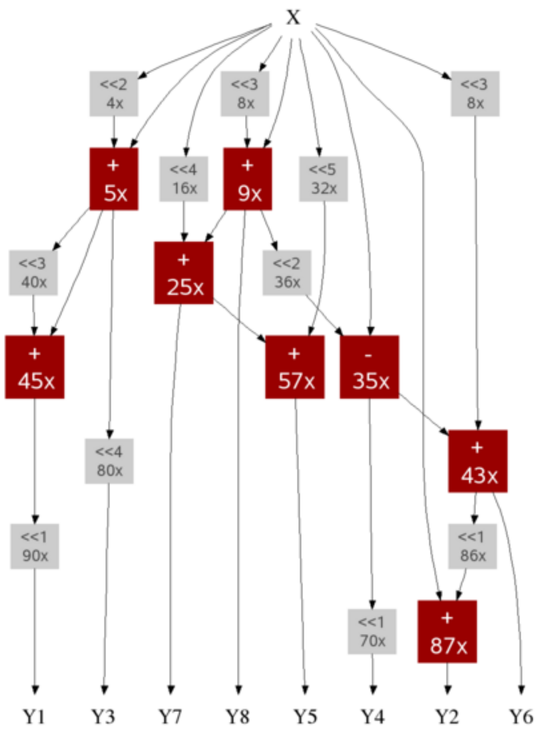

Figure 4.8 The exact implementation of 8 point multiplication sub-module using MCM algorithm [1]. There are 8 add/subtract operations and 11 shift operations.

DCT and comparing the encoded result. However the process becomes very intensive and time consuming once the number of candidates are large. We therefore developed a set of heuristics which can approximately quantify the degradation. The following are the various heuristics which act as cost function for a given set of coefficients.

- Sum of absolute difference between the original and the modified coefficient sets
- Sum of absolute percentage changes to the original set of coefficients
- Comparing diagonal matrix with the result of its modified forward transform followed by inverse transform
- Comparing impulse matrix with the result of its modified forward transform followed by inverse transform
- Check correlation coefficient between DCT of original and modified coefficients

These five heuristics were tested against actual degradation by adding random noise to the original set of coefficients. The last three options perform very closely and are well correlated with the actual degradation result. For our case, we have proceeded with the last heuristic.

We follow the following guideline roughly to obtain at a new set of coefficients with a reduced number of adders.

- Rate all the adders with a importance value. Importance value of an adder is the number of edges coming out of the adder. A high importance value implies the result of the adder is used at many places.

- Choose the adder with the lowest importance value and eliminate its requirement by changing the coefficient(s) affected by it appropriately. The coefficients are changed to the closest value which is already occurring in the map as a result of some other adder. Therefore the original adder will not be required anymore.

- If two or more adders have similar importance values, check the degradation caused by each set of resulting coefficients using the heuristic mentioned earlier.

By repeating the process, we obtain a new set of coefficients at each iteration which requires fewer adders to implement using the MCM algorithm [1] than the previous set. The summary of the different set of coefficients and the corresponding degradation in the final encoding is given in Table 4.3.

4.4.4 Optimizing the transpose memory

The transpose memory contributes 60% of the total area and is therefore a critical component of the area optimization process. Each of the coefficients in the transpose memory is represented by 16 bits. Most of the time the non-DC coefficients in the transpose memory are not very large in magnitude. Therefore they don't use all the 16 bits for in their binary format. Therefore we can effectively apply the idea of moving window to the DCT transpoe memory as not many bits will be lost.

In Chapter 3 we studied the idea of moving window. A 16 bit binary integer can be approximately represented with a window and its position. The idea is extended to transpose memory of the DCT by representing each of the coefficients in that manner. For every coefficient, we encode the moving window in along with the bits describing the position. By our convention, the first bit

Table 4.3 Summary of the various sets of modified coefficients. Also shown are the add/subtract operations required to realize them and the degradation caused by the modification.

Coefficient Set	Add/Sub Operations	BDBR Degradation	
		Class D	Class B
90, 90, 88, 85, 82, 78, 73, 67, 61, 54, 46, 38, 31, 22, 13, 4	13	0.00%	0.00%
92, 92, 88, 88, 82, 78, 73, 62, 62, 54, 46, 41, 31, 22, 13, 4	8	0.00%	0.00%
92, 92, 88, 88, 80, 80, 72, 64, 64, 54, 46, 40, 32, 22, 13, 4	6	0.00%	0.02%
88, 88, 88, 88, 80, 80, 72, 64, 64, 52, 44, 40, 32, 22, 13, 4	4	0.01%	0.08%
88, 88, 88, 88, 80, 80, 64, 64, 64, 52, 44, 40, 32, 22, 13, 4	3	0.11%	0.19%
88, 88, 88, 88, 80, 80, 72, 64, 64, 64, 44, 40, 32, 22, 16, 4	2	0.86%	1.37%

describes the sign of the integer being encoded, the next three bits describe the position and the remaining the magnitude. Experiments are performed for various window sizes and 6 bit window (including the sign bit) and 3 bit position marker is used for final implementation. The results for different bit sizes are summarized in Table 6.1 in Chapter 6.

The original transpose memory consisted of 32×32×16 bits. The modified transpose memory is composed of 32×32×6 bits for the windows and 32×32×3 bits for the positions. Thus each of the coefficient is effectively reduced to 9 bits from 16 bits. The overhead cost will be in the form of combinational logic required to compute the contents of the window and the position bits.

Chapter 5

Approximate DCT for JPEG

5.1 JPEG Overview

"JPEG" stands for Joint Photographic Experts Group, the name of the committee that created the JPEG standard. The JPEG standard specifies the codec, which defines how an image is compressed into a stream of bytes and decompressed back into an image, but not the file format used to contain that stream.

Although a JPEG file can be encoded in various ways, most commonly it is done with JFIF encoding. The encoding process consists of several steps.

1. The representation of the colors in the image is converted from RGB to YCBCR, consisting of one luma component (Y), representing brightness, and two chroma components, (Cb and Cr), representing color.

2. The resolution of the chroma data is reduced, usually by a factor of 2 or 3. This reflects the fact that the eye is less sensitive to fine color details than to fine brightness details.

3. The image is split into blocks of 8×8 pixels, and for each block, each of the Y, Cb and Cr data undergoes DCT.

4. The amplitudes of the frequency components are quantized. Human vision is much more sensitive to small variations in color or brightness over large areas than to the strength of high-frequency brightness variations. Therefore, the magnitudes of the high-frequency components are stored with a lower accuracy than the low-frequency components. The quality setting of the encoder (for example 50 or 95 on a scale of 0–100 in the Independent JPEG Group's library) affects to what extent the resolution of each frequency component is reduced. If an excessively low quality setting is used, the high-frequency components are discarded altogether.

5. The resulting data for all 8×8 blocks is further compressed with a lossless algorithm.

The decoding process reverses these steps, except the quantization because it is irreversible.

As can be seen from the summary of the steps involved, DCT plays a prime role in compressing an image in JPEG. It is also the most computationally intensive step performed for each macro-block. A difference between the DCT used in HEVC and JPEG apart from size variation is the input coefficients. The DCT in HEVC gets residuals as its input whereas the DCT for JPEG receives actual image pixels as input. For calculating the residuals, information about the surrounding macro-blocks is generally required, whereas the pixel information of one macro-block is independent of the surrounding ones. Therefore in JPEG each macro-block of 8×8 pixels can be processed independently. What this implies in terms of hardware implementation is that there can be multiple processing units in a JPEG encoder each capable of handling an 8×8 matrix of pixels improving the encoding speed. Since each unit has an 8×8 DCT, there are multiple DCT modules in a complete encoder. Thus saving area cost using approximate DCT causes multiple times the saving overall due to replication making the savings more significant.

5.2 Approximate DCT

As mentioned in earlier chapter, DCT hardware is essentially composed of combinational logic implementing the row and column 1D DCT and a non-combinational transpose memory to store the 1D transformed coefficients. For the DCT for JPEG encoder application, we perform the modifications on its transpose memory component. The fraction of the hardware area composed by the transpose memory varies with the size of DCT but is significant. The percentage of the area used by transpose memory for 8×8 DCT (which is the size of the macro-block used by JPEG) is ~53%. Henceforth in this chapter, by DCT, we refer to 8 point DCT unless specified otherwise.

The transpose memory when fully filled, contains the result of 1D row transform of each row of the input pixel matrix.

In the actual implementation, the position of which coefficient is stored in which memory location can vary and is immaterial. But for ease of discussion, let us follow a convention as shown in Figure 5.1. By our convention, let us assume that the result of each row 1D transform is stored row wise in the transpose memory with the lower frequency coefficients to the left. Then all the coefficients in a given column will have the same frequency components. The left most column will all be DC components and the right most will be the highest frequency components.

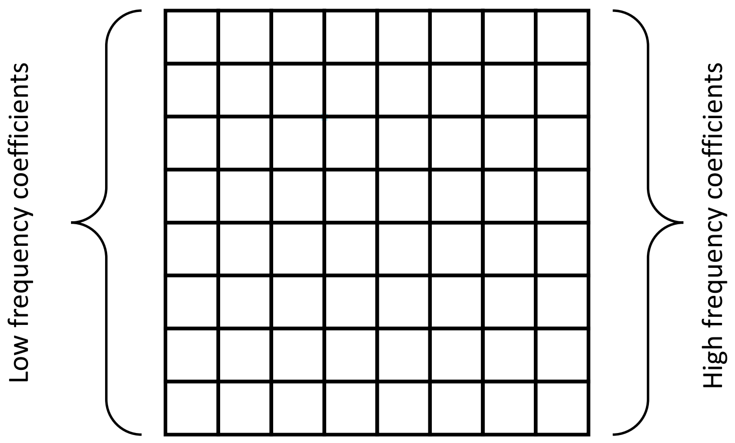

Figure 5.1 The convention followed for the transpose memory alignment. The left most column contains all the DC coefficients and right most column the highest frequency coefficients.

5.3 Application of Moving Window to DCT transpose memory

Due to the good energy compaction property of DCT, most of the non-DC coefficients have small values. Hence the whole range of possible values of coefficients is seldom utilized. Most of the bits of non-DC elements remain unused.Therefore it is possible to use moving window implementation for saving memory without much loss in the final image quality.

The transpose memory in JPEG DCT is an 8×8 coefficient matrix with each coefficient being 16 bits wide. Hence the total memory is 8×8×16 bits. We modify the transpose memory so that instead of storing 16 bits for each coefficient, we only store the moving window. Thus

$$total\,memory = moving\,window\,size \times 8 \times 8.$$

We will be however encoding the positions of the windows which will result in some overhead cost. Three ways for storing the positions are discussed along with their limitations. The final implementation uses the hybrid implementation which combines the advantages of these method.

5.3.1 Ideal implementation

In this case the ideal window position is calculated for each of the coefficients in the transpose memory separately. Since each coefficient has its own unique window position, window position of each of the coefficients needs to be stored. Therefore the overhead in this case is the bits required store all the positions.

For an 8 bit window for representing a 16 bit integer, eight different positions are possible. Hence 3 bits will be required for denoting the position of the 8 bit window. For a 7 bit window, the total positions possible increase to nine and the bits required to 4. But if we can discard one of the positions, we can still continue to use 3 bits and limit ourselves to 8 positions. In case of 7 bit window, we choose to discard the right most position, thus bringing down the total possible positions to 8 and we continue to use 3 bits for denoting the window position.

Hence now the overhead will be $8 \times 8 \times 3$ bits, which is effectively another matrix of size 8×8 and each cell of width 3 bits. Instead of having another separate matrix of registers, we increase the width of the first register matrix to accommodate bits for encoding positions. Each cell of transpose memory now has the structure as shown in Figure 5.2.

5.3.2 Window position based on first row

As mentioned previously, every row of DCT transpose memory consists the result of 1D DCT of one row of pixels according to our convention. Another observation which can be made is that since the image being encoded is mostly

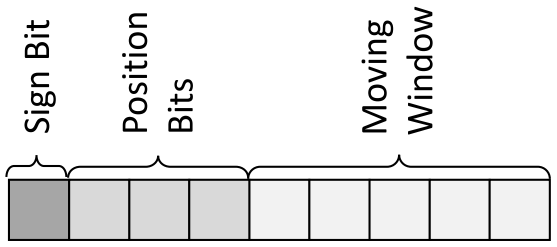

Figure 5.2 Breakup of each memory location in the transpose memory. One bit is used to denote the sign of the number, three bits for the window position and five bits for the magnitude.

smooth at pixel level and doesn't vary abruptly, the adjacent pixel rows are likely to be similar. Hence in a macro-block of 8×8 pixels, the eight rows will have pixel values similar to each other. This translates to the result of their 1D DCT as well, i.e. the 1D row DCT result of each row will be similar in value to each other. Furthermore, if the values are similar, the ideal starting positions of the windows will be similar too (since the window position directly relates to the magnitude of the integer, larger the magnitude, more to the MSB side is the window position). Hence the window positions for every coefficient in the transpose memory need not be stored. Only the position for the first row can be stored and the positions for other rows can be assumed to be same as the first row.

The overhead is greatly reduced in this case as the positions are stored only for the first row. Since position for each coefficient requires 3 bits and there are 8 coefficients in total in the first row, we require 24 bits in addition to the transpose memory. It can be merged within the memory for the first row and make the window size smaller by 3 bits. However since the overhead is very small and to maintain consistency for all the rows, we add an additional array

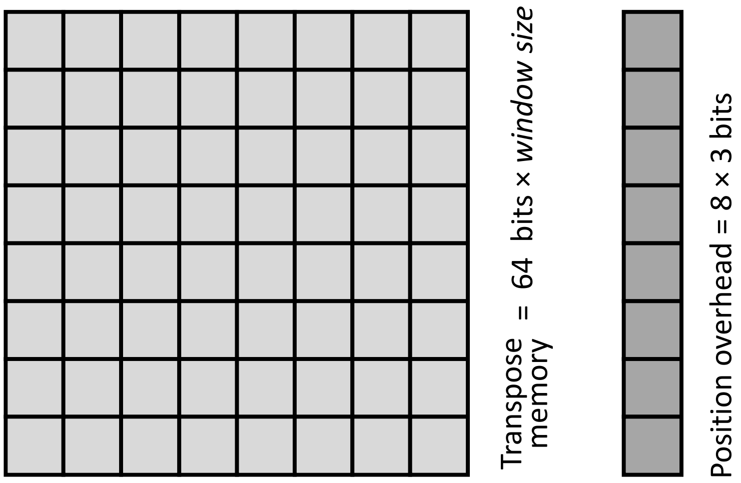
Transpose memory = 64 bits × *window size*
Position overhead = 8 × 3 bits

Figure 5.3 Memory overhead for moving window implementation based on first row. Additional 24 bits are required to store the positions.

of registers for storing the positions.

The idea here as mentioned is that the coefficients in every row will be similar to other rows and hence have similar window position values. But in case a value in the first row is close to a power of 2 (2^x), then a small increase in this value causes the ideal window position to jump by 1 which will cause the most significant 1 (and most significant 0 for negative numbers) to be lost. This causes significant degradation for that particular row when the image is reconstructed. To avoid this, we provide some leeway by actually choosing the window position as the ideal position + 1.

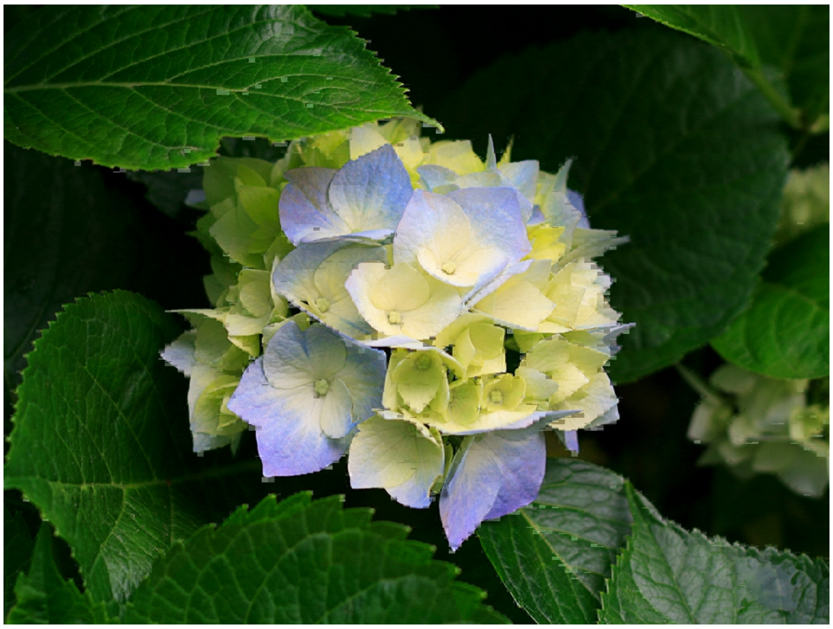

Figure 5.4 JPEG encoding result of row based moving window implementation. Artefacts can be seen for blocks near the edges especially at the edges between flower and leaves.

5.3.2.1 Cases of failure

The prime assumption in this method was that all the rows of a given block are similar in values. In case there is a horizontal edge in the image passing through the block, this is not the case anymore. Therefore in this case the rows which are not similar to the first row will be reconstructed wrongly. Figure 5.4 shows the result of encoding using this method and the artifacts which arise where the horizontal edges exist.

This can be prevented by attempting to detect for which blocks this is likely to happen and encoding that block differently. We detect the blocks for which this could happen by the following rudimentary formula based on Figure 5.5.

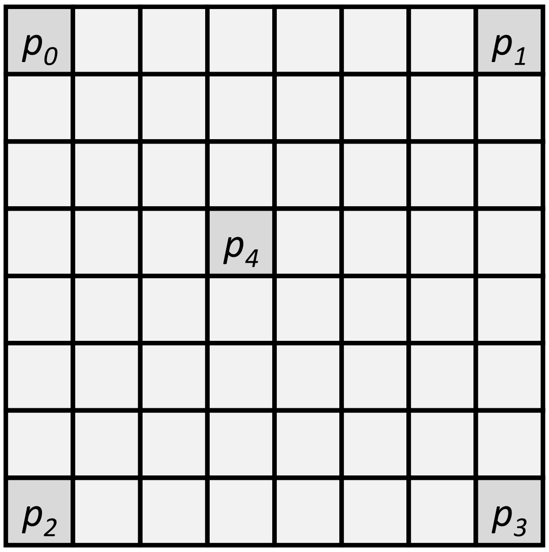

Figure 5.5 Input block to the DCT module showing pixels used for detecting the fail case of row based moving window implementation.

$$\left| p_0 - \frac{p_0 + p_1 + p_2 + p_3 + p_4}{5} \right| < 32$$

$$\left| p_1 - \frac{p_0 + p_1 + p_2 + p_3 + p_4}{5} \right| < 32$$

$$\left| p_2 - \frac{p_0 + p_1 + p_2 + p_3 + p_4}{5} \right| < 32$$

$$\left| p_3 - \frac{p_0 + p_1 + p_2 + p_3 + p_4}{5} \right| < 32$$

$$\left| p_4 - \frac{p_0 + p_1 + p_2 + p_3 + p_4}{5} \right| < 32$$

The detection algorithm needs to be kept simple in order to keep the hardware overhead cost low. Once detected, ideal implementation is performed for those blocks with three bits in the window reduced to accommodate for the window position. The overhead of this method are the additional logic for detecting the likely blocks (3 adders and four comparators. A divider isn't required for the logic since we have carefully chosen four pixels to average over so that division can be performing by simple right shift operation) and the memory array of 8×3 bits.

5.3.3 Position based on first column

Another observation for the result of 1D row transform which is stored in the transpose memory is that the DC coefficient is generally the largest of all the eight coefficients. This is due to the fact that image is generally smooth at pixel level and hence the high frequency variations will almost always be lower than the DC value. This fact can be used for not having to store the window positions repeatedly. For every row we perform the ideal window position calculation for the DC coefficient and assume the same window position for the rest of the coefficients. Thus we need to store the positions only once per row of the macro-block.

Hence the initial overhead of storing the position for each of the coefficients is vastly reduced. We can choose to use three bits of the DC coefficient memory location to store its position hence eliminating any memory overhead. But as the overhead is small and since visually we are more perceptible to DC component of the image, we add an additional array of memory (3 bits $\times$ 8) to store the window positions for each of the rows.

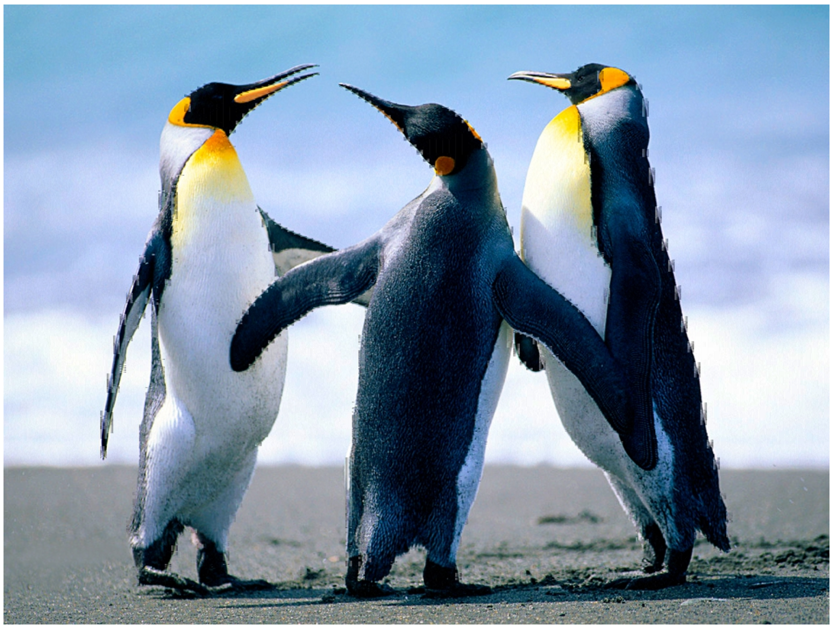

Figure 5.6 JPEG encoding result of column based moving window implementation. Artifacts can be seen for blocks near the vertical edges.

5.3.3.1 Cases of failure

The prime assumption in this case was that the value of the DC component is larger than the rest of the components for every row. In case that is not true, that particular row will be recovered wrongly causing distortion in the final image. This can happen in cases where the block contains some kind of edge and is not mostly uniform. It helps to worsen the case further if the block contains a vertical edge and/or is mostly dark. This results in a reduced DC value which is easy for other coefficients to exceed.

As before, we attempt to detect the blocks which might fail this encoding and then encode those blocks ideally. The algorithm used to detect should be rudimentary to keep the overhead hardware cost low as before. The algorithm

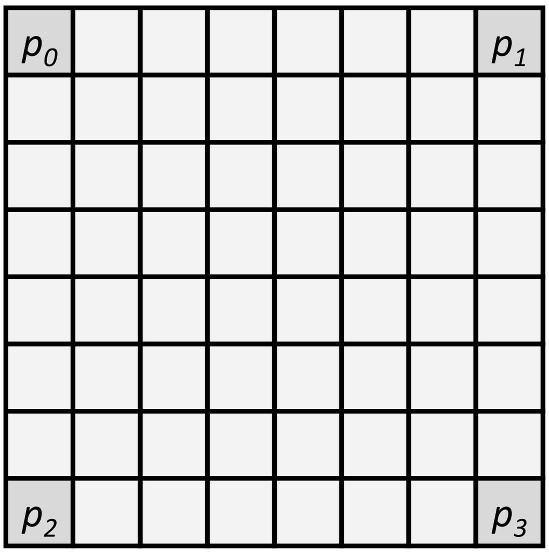

Figure 5.7 Input block to the DCT module showing pixels used for detecting the fail case of column based moving window implementation.

used is based on Figure 5.7 as follows.

$$|p_0 - p_1| < 32$$

$$|p_2 - p_3| < 32$$

The final overhead of this method is the memory array of 8×3 bits and the combinational logic required for detecting blocks likely to fail the encoding.

5.4 Hybrid implementation

In the last two cases we reduced the overhead of storing the window positions for every coefficient. Then we added logic to detect those block which are likely to fail the algorithm. The transpose memory contents of these blocks were then encoded using the ideal encoding algorithm, wherein the window position is stored for every coefficient.

One observation to be made about the above two cases is that the logic for pre-detecting the blocks likely to fail is a heuristic algorithm. This means that it is not fail proof and can fail in some cases. The resulting distortion artifacts caused in the resulting image for both the images where the detection logic fails to identify will be strongly visible.

This therefore might not be acceptable for applications where the resulting image needs to be of high quality. We therefore use another detection algorithm to identify the blocks which are going to fail which combines the ideal method with the above methods. This is not based on heuristic and hence will identify all the failing blocks.

Instead of pre-detecting the macro-blocks likely to fail, we perform the detection calculations once the 1D transform coefficients are calculated.

- *Window position based on first row*: For the first row, the encoding is performed as usual. The first row will always be decoded correctly since the positions being stored in the position array are ideal with respect to the first row. From the second row onward, for each row, the ideal positions are calculated for each of the eight coefficients as in the ideal case. These eight ideal positions are compared to the respective ideal positions of the first row. If any of the positions exceed the corresponding position value of the first row, the row is flagged. If flagged, the row (not the whole block) is encoded with ideal case encoding. We do not require the heuristic logic any more and can save the area used by the logic. The additional memory overhead for this method compared to the previous methods are:
    - Eight flag bits to indicate whether the row should be decoded using

    ideal method or base on first row

  - Eight comparators to compare the ideal position of a given row with the first row

- *Window position based on first column*: In this case, every row is independent and needs independent computation (unlike the previous case where the rows are related to the first row). The DC coefficient is assumed to be larger than all the others. Even in the cases where the DC coefficient is smaller but the ideal window position for the DC coefficient is equal to the ideal window positions of the other coefficients, we can continue with the original method. The reconstructed result is erroneous only in case where any of the ideal window positions of the non-DC coefficients is larger than that of the DC coefficient. Therefore for detecting, we calculate the ideal window positions for all the coefficients as before. Then we compare each of the non-DC coefficients' ideal window positions with the DC coefficient positions and make sure the latter is always larger or equal. In case the comparison fails, a marker flag is raised and the encoding method for transpose memory reverts to the ideal method. The heuristic logic is not required, however the two overheads exist as before:

  - Eight flag bits to indicate whether the row should be decoded using ideal method or base on first column

  - Seven comparators to compare the ideal position of a given row with the DC coefficient of the row

In both the cases when a row is flagged and is encoded using ideal method, the position bits need to be encoded for each of the eight coefficients. Since additional memory doesn't exist, these bits have to be accommodated within

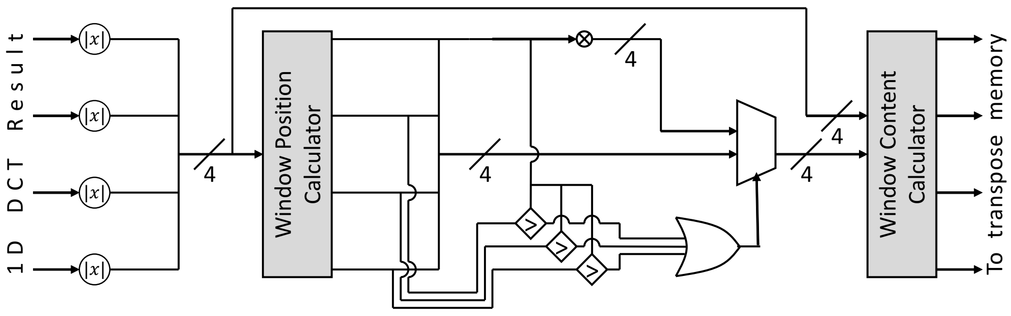

Figure 5.8 The architecture for implementing the hybrid algorithm for moving window. This architecture is the combinational logic overhead required for the implementation.

the window itself. This is performed by discarding three least significant bits of the window and using them to store the position bits. This effectively reduces the window size by 3 bits when a row is flagged.

Out of the two cases, the second case (based on first column) performs better and is hence used for the final implementation. The additional combinational logic architecture required for the implementing this is shown in Figure 5.8. The window position calculator module computes the ideal window position for each of the integer input. The positions are then compared to the DC coefficient's ideal window position. The multiplexer then chooses the appropriate set of window positions based on the flag value. The window content calculator module then takes the integers and their window positions, and gives the window content as the output.

The decoder follows the similar procedure. If the flag is raised, each coefficient is decoded separately or else decoded based on DC coefficient's position.

Chapter 6

Experimental Results

This section summarizes all the important results for both HEVC and JPEG using approximate DCTs mentioned in the respective sections. For the HEVC the approximate DCT used has modified combinational logic (multiplication module and rounding adders) as well as modified transpose memory (with the ideal moving window implementation). The approximate DCT used for JPEG uses only the modified transpose memory using hybrid moving window implementation. The idea of memory reduction can be easily generalized for applications other than HEVC or JPEG too.

The architectures for all the DCTs were implemented in Verilog and synthesized with the TSMC 65nm technology. The gate counts of the final implementation is detailed along with the degradation.

6.1 HEVC Experiments and Results

The experiments for HEVC were performed with HM 13.0 as reference. HM stands for HEVC Test Model and provides reference C++ code implementing the H.256 encoder and decoder. The modifications were performed in the HM 13.0 code in software to reflect the changes in hardware implementation.

All the results are provided for two sets of videos, one set is of B class sequences (full HD - 1920×1080pp) and the other set contains D class sequences (low resolution - 416×240pp). The first set contains sequences Basketball Drive, BQ Terrace and Park Scene. The second set consists of the sequences Blowing Bubbles, Basketball Pass and BQ Square. The results for B class is the average of the first three sequences and the results for the D class the average of the latter three sequences.

The final implementation for transpose memory was performed with 6 bit moving window. The memory overhead is 3 bits for describing the position of the window. The coefficient modified multiplier module with 4 add/subtract operations was chosen for the final implementation.

The original gate count for 32×32 DCT module was 318K, of which the transpose memory requires 192K gates. The new memory requires 115K gates with a combinational logic overhead of 16K resulting in a total saving of 61K (31% of the initial memory gate count). The multiplier module was reduced from 50K to 16K. The final gate count is 218K resulting in a total saving of 32%. The degradation caused by the final implementation is 0.42%.

6.2 JPEG Experiments and Results

All the JPEG experiments were performed with IJG JPEG software as reference. The IJG JPEG reference software provides C code implementing the

Table 6.1 BDBR degradation for HEVC for different sized bit widths.

Bits Allowed	BDBR Degradation
16	0.00%
15	0.00%
14	0.00%
13	0.01%
12	0.02%
11	0.03%
10	0.03%
9	0.09%
8	0.17%
7	0.19%
6	0.23%
5	0.44%
4	1.10%

Table 6.2 Summary of gate count for modified DCT for JPEG

Module	Gate Count
Original 8×8 DCT	19407
Original transpose memory	10392
New total memory	4557
Combinational logic overhead	3038
Modified 8×8 DCT	11727

JPEG image encoder and decoder. All the modifications emulating the transpose memory modification were performed in the C code.

Four images were used as set of reference images for all the experiments. The images used are Desert, Hydrangeas, Koala and Penguins. These images were selected for their varying features. Desert image contains a lot of smooth and bright areas, Hydrangeas contains decent amount of sharp edges alternating between bright and dark regions. The Koala image contains lots of minute details with sharp features and high frequency components and the Penguins image contains smooth and dark regions.

The final implementation of moving window uses 7 bit moving window with hybrid algorithm.

From the summary in Table 6.2 the effective saving on area is ~27%. The memory itself composes ~53% of the total DCT hardware area. The total saving for the complete DCT module translates to ~14%. The comparison of the PSNR values using the original JPEG encoding and the modified JPEG encoding is shown in Figure 6.1 for four test images. The resulting BDBR degradation doesn't exceed 0.86% for any of the tested images.

Original set of images are shown in Figures 6.2, 6.3, 6.4 and 6.5. The re-

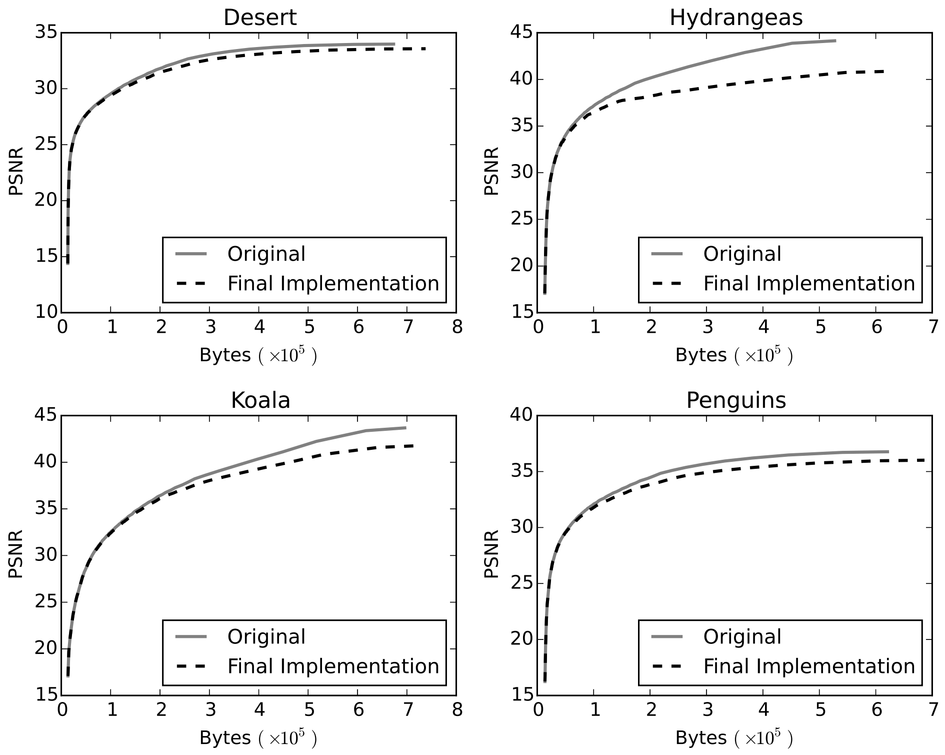

Figure 6.1 PSNR vs bytes plots for the original JPEG encoding and our final implementation. The four plots are for four different test images used.

Figure 6.2 Test sample image - Desert

sulting images obtained when encoded using modified DCT implementation are shown in Figures 6.6, 6.7, 6.8 and 6.9.

Figure 6.3 Test sample image - Hydrangeas

Figure 6.4 Test sample image - Koala

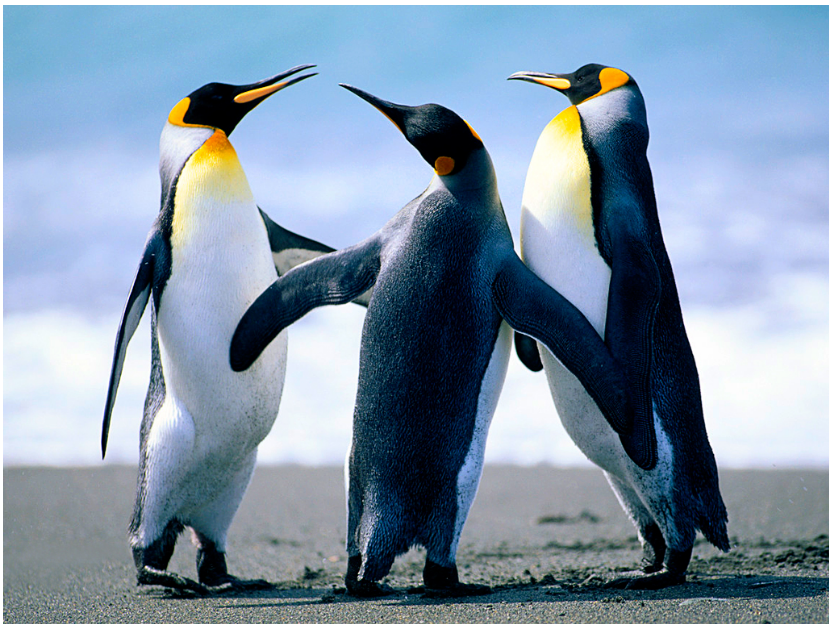

Figure 6.5 Test sample image - Penguins

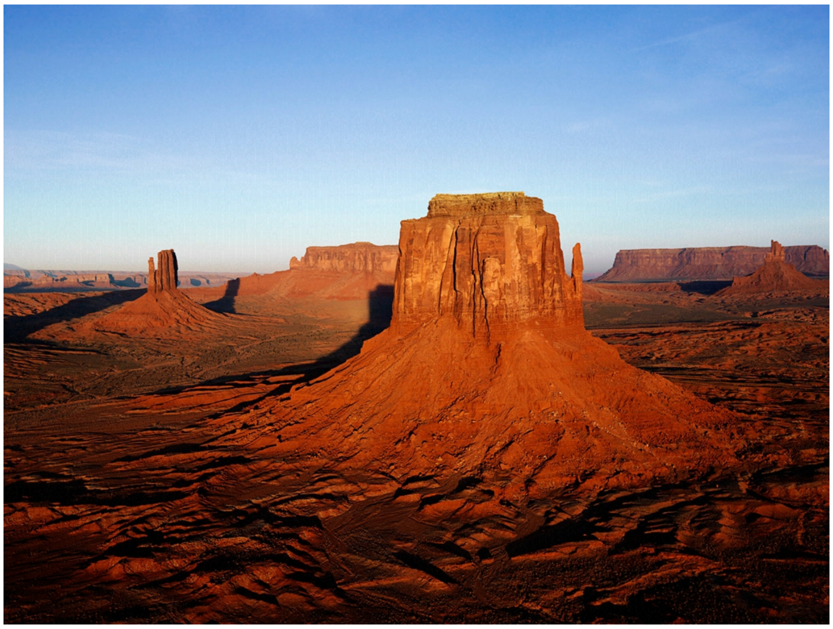

Figure 6.6 Final decoded image - Desert

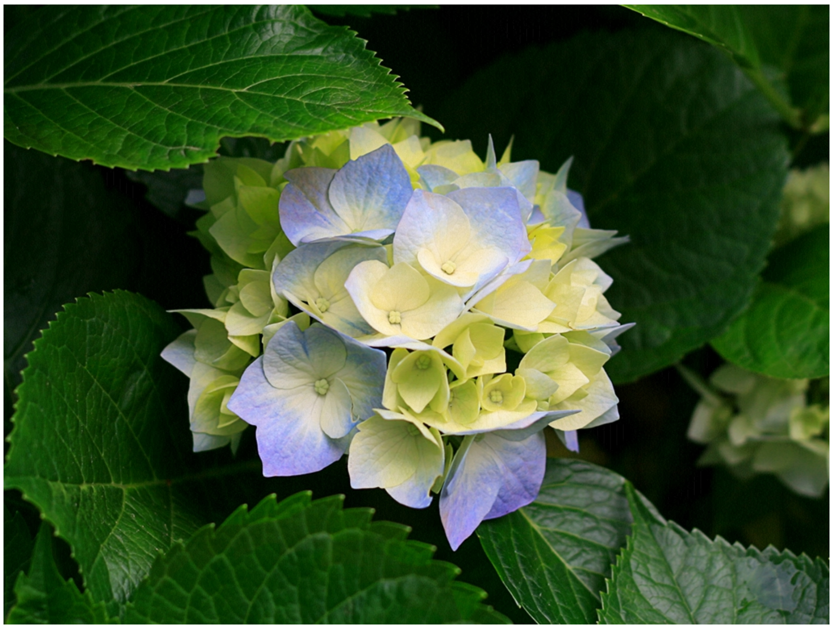

Figure 6.7 Final decoded image - Hydrangeas

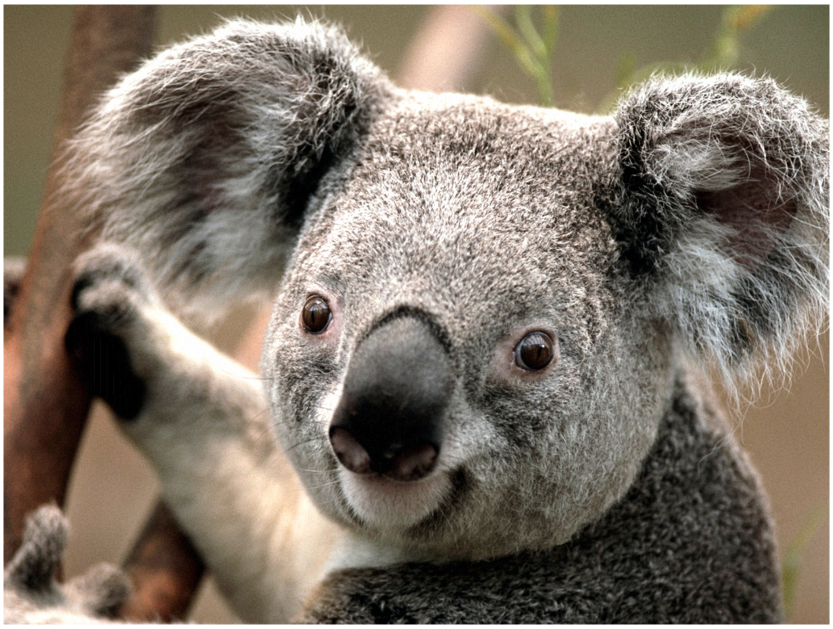

Figure 6.8 Final decoded image - Koala

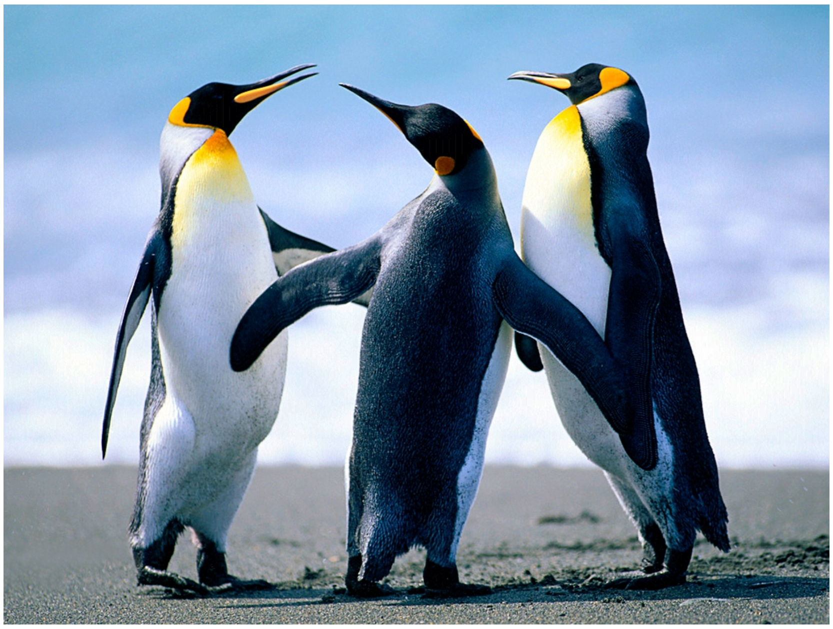

Figure 6.9 Final decoded image - Penguins

Chapter 7

Conclusion

Two key ideas are presented to reduce the area of the DCT hardware module.

- The first idea is for the non-combinational part (register memory) of the complete module.
- The second idea is for reducing the multiplication block which composes a part of the combinational logic of the module.

The idea used for reducing the register memory is called moving window implementation. It reduces the bits required to represent a single integer. This is further extended to all the integers stored in the transpose memory. The overhead cost of the implementation is the additional combinational logic required to calculate the contents of the window. However, the gates required for the overhead logic doesn't exceed the number of gates saved by reducing the memory. Therefore the idea is viable. The combinational logic overhead required for implementing this idea for the DCT module scales linearly with the 2D DCT size, i.e. the combinational area overhead for a 32×32 DCT will roughly be four times the size of the overhead for an 8×8 DCT. The memory savings on the other hand increase as a square of the 2D DCT size, i.e. the memory area saved for 32×32 DCT will be sixteen times that for an 8×8 DCT. Therefore the percentage savings improve with size of the DCT. It should be noted that the idea of moving window is generic. Hence it can easily be extended to other memory applications too where approximate values can be tolerated.

The idea for reducing the multiplier block aims to do so by changing the multiplication coefficients of the block. The MCM output which is generated by an existing algorithm for the original multiplier block is simplified, thus reducing the adders and shifters required. The process of changing the coefficients is performed manually with multiple iterations. The benefit of this idea too improves with the size of the DCT module. Larger the DCT module, more the original number of adders and more coefficients are available for modification. Therefore there is more scope for larger reduction.

These ideas were applied to the DCT modules of JPEG and HEVC encoders. JPEG encoder utilizes 8×8 DCT module. The moving window implementation with 7 bit window was used and no combinational logic reduction was implemented (benefit of multiplication module reduction was not much since the coefficients available are few), resulting in ~14% area reduction. HEVC encoder utilizes DCT of maximum size of 32×32. Both the reductions, register memory as well as multiplication module were implemented. The total area reduction 32% is achieved. The quality loss in both the cases is non-significant.

Bibliography

[1] Y. Voronenko, “Spiral multiplier block generator. spiral project,” 2010.

[2] C.-T. Hsu and J.-L. Wu, “Energy compaction capability of dct and dht with ct image constraints,” in *Digital Signal Processing Proceedings, 1997. DSP 97., 1997 13th International Conference on*, vol. 1, Jul 1997, pp. 345–348 vol.1.

[3] N. Ahmed, T. Natarajan, and K. Rao, “Discrete cosine transform,” *Computers, IEEE Transactions on*, vol. C-23, no. 1, pp. 90–93, Jan 1974.

[4] W.-H. Chen, C. Smith, and S. Fralick, “A fast computational algorithm for the discrete cosine transform,” *Communications, IEEE Transactions on*, vol. 25, no. 9, pp. 1004–1009, Sep 1977.

[5] B. Lee, “A new algorithm to compute the discrete cosine transform,” *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 32, no. 6, pp. 1243–1245, Dec 1984.

[6] M. Wagh and H. Ganesh, “A new algorithm for discrete cosine transform of arbitrary number of points,” *Computers, IEEE Transactions on*, vol. C-29, no. 4, pp. 269–277, April 1980.

[7] B. Tseng and W. Miller, "On computing the discrete cosine transform," *Computers, IEEE Transactions on*, vol. C-27, no. 10, pp. 966–968, Oct 1978.

[8] H. Malvar, "Fast computation of the discrete cosine transform and the discrete hartley transform," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 35, no. 10, pp. 1484–1485, Oct 1987.

[9] A. Edirisuriya, A. Madanayake, R. Cintra, V. Dimitrov, and N. Rajapaksha, "A single-channel architecture for algebraic integer-based 8×8 2-d dct computation," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 23, no. 12, pp. 2083–2089, Dec 2013.

[10] Y. Chen, X. Cao, Q. Xie, and C. Peng, "An area efficient high performance dct distributed architecture for video compression," in *Advanced Communication Technology, The 9th International Conference on*, vol. 1, Feb 2007, pp. 238–241.

[11] Y. Shain, A. Akerib, and R. Adar, "Associative architecture for fast dct," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 5, May 1998, pp. 3109–3112 vol.5.

[12] R. Kordasiewicz and S. Shirani, "Asic and fpga implementations of h.264 dct and quantization blocks," in *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, vol. 3, Sept 2005, pp. III–1020–3.

[13] J. Bruguera and R. Osorio, "A unified architecture for h.264 multiple block-size dct with fast and low cost quantization," in *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, 2006, pp. 407–414.

[14] R. Jeske, J. de Souza, G. Wrege, R. Conceicao, M. Grellert, J. Mattos, and L. Agostini, “Low cost and high throughput multiplierless design of a 16 point 1-d dct of the new hevc video coding standard,” in *Programmable Logic (SPL), 2012 VIII Southern Conference on*, March 2012, pp. 1–6.

[15] P. Meher, S. Y. Park, B. Mohanty, K. S. Lim, and C. Yeo, “Efficient integer dct architectures for hevc,” *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 24, no. 1, pp. 168–178, Jan 2014.

[16] S. Y. Park and P. Meher, “Flexible integer dct architectures for hevc,” in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, May 2013, pp. 1376–1379.

[17] A. Hossen and U. Heute, “Fast approximate dct: basic-idea, error analysis, applications,” in *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*, vol. 3, Apr 1997, pp. 2005–2008 vol.3.

[18] R. Haweel, W. El-Kilani, and H. Ramadan, “A fast modified signed discrete cosine transform for image compression,” in *Computer Engineering Systems (ICCES), 2014 9th International Conference on*, Dec 2014, pp. 56–61.

[19] K. Lengwehasatit and A. Ortega, “Scalable variable complexity approximate forward dct,” *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 14, no. 11, pp. 1236–1248, Nov 2004.

[20] A. Madanayake, R. Cintra, V. Dimitrov, F. Bayer, K. Wahid, S. Kulasekera, A. Edirisuriya, U. Potluri, S. Madishetty, and N. Rajapaksha, “Low-power vlsi architectures for dct/dwt: Precision vs approximation for hd

video, biomedical, and smart antenna applications," *Circuits and Systems Magazine, IEEE*, vol. 15, no. 1, pp. 25–47, Firstquarter 2015.

[21] U. Sadhvi Potluri, A. Madanayake, R. Cintra, F. Bayer, S. Kulasekera, and A. Edirisuriya, "Improved 8-point approximate dct for image and video compression requiring only 14 additions," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 61, no. 6, pp. 1727–1740, June 2014.

[22] B. Shen, "From 8-tap dct to 4-tap integer-transform for mpeg to h.264/avc transcoding," in *Image Processing, 2004. ICIP '04. 2004 International Conference on*, vol. 1, Oct 2004, pp. 115–118 Vol. 1.

[23] X. Zhao, J. Sun, S. Ma, and W. Gao, "Novel statistical modeling, analysis and implementation of rate-distortion estimation for h.264/avc coders," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 20, no. 5, pp. 647–660, May 2010.

[24] H. Shen, X. Sun, and F. Wu, "Fast h.264/mpeg-4 avc transcoding using power-spectrum based rate-distortion optimization," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 18, no. 6, pp. 746–755, June 2008.

[25] Z. Chen, Q. Chen, and C. Wang, "Content-dependent frequency domain based rdo mode decision," in *Information, Communications and Signal Processing, 2005 Fifth International Conference on*, 2005, pp. 1140–1144.

초록

Discrete Cosine Transform(DCT)는 뛰어난 에너지 압축(energy compaction) 특성 덕분에 다양한 이미지, 비디오 압축 분야에서 널리 사용된다. DCT는 연산의 복잡도가 높고, 병렬화가 가능한 특성이 있다. 따라서 연산의 속도를 높이기 위해 주로 하드웨어로 구현된다. 그러나 일부 application에서는 크기나 큰 DCT나 다수의 DCT 모듈이 필요하기 때문에 이미지, 비디오 부호화기(encoder) 하드웨어의 면적에서 DCT가 차지하는 비중이 상당히 커지고 있다. 대부분의 application에서는 DCT가 정확(exact)할 필요는 없다. 이러한 점을 이용하여 본 논문에서는 DCT 모듈의 하드웨어 면적을 줄이기 위한 새로운 방법을 제안한다. DCT 하드웨어 모듈은 조합 논리회로(combinational logic)와 메모리로 구성되어 있다. 두 가지 구성요소 모두의 크기를 감소시켰으며, 전체 구현을 설명한다. 목표로 하는 application은 HEVC와 JPEG이지만, 제안하는 방법은 어느 DCT 하드웨어 구현에나 적용 가능하다. 마지막으로 이미지, 비디오 압축 성능의 저하를 BDBR을 이용해 논의하며 합성을 통해 얻은 gate count도 제시된다.

주요어: Discrete Cosine Transform, HEVC, JPEG, Approximate DCT

학번: 2013-23847