DCT ARCHITECTURE DESIGN FOR HEVC / H.265 VIDEO CODING

Joshua Haddrill

Bachelor of Engineering
Computer Engineering

Department of Engineering
Macquarie University

November, 7 2016

Supervisor: Dr Yinan Kong

ACKNOWLEDGMENTS

I would like to acknowledge Niras Cheeckottu-Vayalil and Yinan Kong for their assistance and guidance throughout the project.

STATEMENT OF CANDIDATE

I, Joshua Haddrill, declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the Department of Engineering, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification or assessment an any academic institution.

Student's Name: Joshua Haddrill

Student's Signature: J.Haddrill (Digital)

Date:07/11/2016

ABSTRACT

H.265/High Efficiency Video Codec (HEVC) is a relatively new codec that is poised to replace H.264/AVC as the High Definition encoding standard. The Discrete Cosine Transform (DCT) is widely used for the compression of video frames and images including use in HEVC. The document proposes an architecture that completes a two dimensional DCT (2D-DCT) that uses a smaller area or a smaller gate count than existing architectures, while maintaining a similar throughput. The architecture is based on the algorithm proposed by Meher et al for a reusable architecture [4]. The architecture is written using VHDL hardware description language to construct One Dimensional DCT (1D-DCT) modules of 4, 8, 16 and 32 point lengths that are used twice in combination with a transpose unit to compute the 2D-DCT. The 1D-DCT modules, that have a length greater than 4, use a reusable architecture that incorporates the N/2 DCT module and shift-adders to compute the DCT more area efficient then the common matrix multiplication method. The architecture was synthesized with Synopsys Design Tools to produce an Application Specific Integrated Circuit (ASIC) that is able to encode 8K UHD video files at 60 FPS in a real time frame while saving more than 66% in hardware area or number of logic gates.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the demand for High Definition (HD) and Ultra High Definition (UHD) video content increases as does the need for efficient compression techniques. The H.265/High Efficiency Video Codec (HEVC) [7] is a relatively new codec that is poised to replace H.264/AVC [8] as the standard for high definition video encoding. The development of compression process for this codec is crucial to allow for HD content to be more readily available and accessible by reducing the total file size of the media, while minimizing loss of quality. The HEVC offers approximately a 50% reduced bit-rate than AVC for a near equivalent reproduction of quality [5]. The use of the Discrete Cosine Transform (DCT) is a common method in several previous codecs and could be a key factor in the development of compression techniques for the HEVC due to its near optimal efficiency for performing this task [4]. To accommodate for the varying size HEVC, the DCT needs to be computed for a matrix of varying lengths. Due to the nature of the transform being similar to that of a Fourier transform it is possible to compute this by completing the one dimensional DCT on the rows of the matrix followed by performing the DCT on the columns of the resulting matrix or vice-versa. To accommodate for the varying size of the architecture it would be ideal to develop components that would be able to be utilized by other lengths such that the architecture is more area efficient. This means that the common method of multiplying by a constant matrix would not be effective in this case, due to this it's architecture not being able to be reused by other lengths. The development of a dedicated, efficient hardware implementation that can perform a varying length DCT with a reusable architecture would allow for a more area efficient design than other compression techniques currently being applied to the HEVC.

1.1 Project Objectives

The main objective of this project is to create a well rounded two dimensional DCT architecture with input lengths of 4, 8, 16 and 32 at the hardware level, this will be completed using VHDL hardware description language. The architecture is intended to be used as part of the compression process for H.265/HEVC to reduce the total file size and computation time. The proposed architecture should provide improvements in either area efficiency, computational speed/throughput or a combination of these when compared to the existing DCT designs used for HEVC. This should be done without affecting the accuracy of the transform and without losses in other key areas of comparison.

Chapter 2

Background and Related Work

2.1 Discrete Cosine Transform

2.1.1 Discrete Cosine Transform Definition

The Discrete Cosine Transform (DCT) is a Fourier related transform that only uses real numbers to represent a set number of discrete data points within a signal, unlike the traditional Discrete Fourier Transform (DFT) the DCT only uses cosine functions to represent the data points [1]. There are multiple versions of the DCT that range from DCT-I to DCT-IV, the most common of which is the DCT-II which is referred to as 'the DCT' and defined in Equation 2.1.

$$X_k = \sum_{n=0}^{N-1} cos\left[\frac{\pi}{N}(n+\frac{1}{2})k\right] x_n for 0 \leq k < N \tag{2.1}$$

For this project however we require the use of the two-dimensional (2D) DCT, such that it will be applicable to the data type. The due to the separability of the multidimensional DCT, the 2D-DCT is a trivial expansion of the standard DCT as it can be obtained easily when viewing the data as a matrix to give Equation 2.2.

$$X_{k_1,k_2} = \sum_{n=0}^{N-1}\sum_{n=0}^{N-1} cos\left[\frac{\pi}{N_1}(n_1+\frac{1}{2})k_1\right]\left[\frac{\pi}{N_2}(n_2+\frac{1}{2})k_2\right] x_1 x_2 \tag{2.2}$$

The process involves first computing an intermediate matrix by performing the 1D-DCT of the rows of the matrix, following this the 1D-DCT is computed on the intermediate matrix to complete the 2D-DCT. This process is illustrated in Figure 2.1.

2.1.2 DCT Usage

The DCT has become a staple in image compression, specifically in the JPEG format, due to the resulting lossy compression that occurs as a result of the transform, allowing

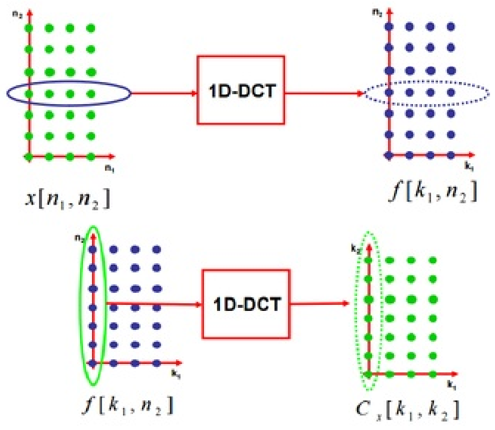


Figure 2.1: Illustrated 2D-DCT computation [10]

larger image data to be compressed. This is done by applying the DCT to a quantization of an image's pixels to obtain an approximation that requires less data to be stored.

The DCT posses a strong energy compaction property [6] as most of the signal information tends to be concentrated in the lower frequencies components that make the DCT useful for image compression.

The DCT does this particularly well when compared to that of the DFT approximation of the same image as seen in Figure 2.2. This is because human vision is much more sensitive to small variations in colour or brightness over larger areas when compared tot he strength of high frequency variations in brightness. The DCT uses the previously defined property to effectively store the magnitudes of higher frequencies with lower accuracy then that of the lower frequency components [6]. where the colour spectrum on the left represents the approximation value and the graph on the right represents the spread of results. The reduced spread is clearly seen in the DCT, resulting in less data to be stored as it can reference the other identical data in less memory.

The related Fourier properties of the DCT make it possible to use the butterfly multiplication approach described by Budagavi et al, such that the overall transformation is completed in sections that effectively 'folds' into the next section, as visualized in Figure 2.3 [2]. This method is ideal as it is more efficient then brute force matrix multiplication method that is very costly in computing time [2].

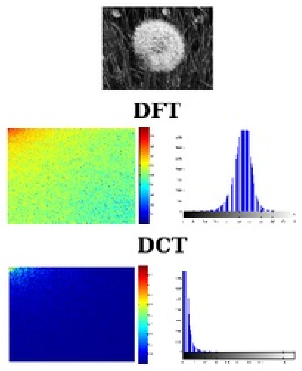


Figure 2.2: Image approximation: DCT vs DFT [10]

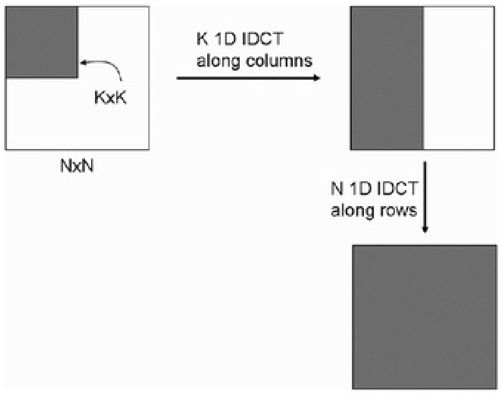


Figure 2.3: Visualization of the Butterfly technique [2]

2.2 Related Work

There are a number of proposed DCT algorithms proposed for use with the HEVC, the two that seemed most relatable to the problem were the algorithms proposed by Meher et al [4] and Masera et al.

2.2.1 Meher et al's Proposed Algorithm

Meher et al propose a reusable architecture algorithm to perform the transformation that performs the transformation in stages gradually building from the 4-point DCT to obtain the completed transforms for the 8, 16 and 32-point lengths. A generalized architecture is over viewed in Figure 2.4.

At each stage of this process the input data is first manipulated by an Input Adder

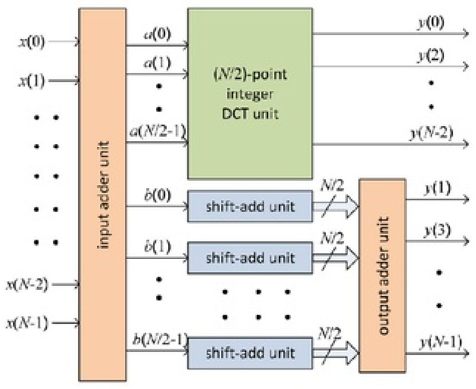


Figure 2.4: Generalized architecture for 8, 16 and 32 point DCT [4]

Unit (IAU) to create intermediate data that is then used as the input for the lower and current operations can be performed. The even numbered rows/columns, including zero, are processed as an N/2 point DCT to obtain the corresponding output values. The odd numbered rows and columns are passed through a Shift-Add Unit (SAU), which is specific to the point length of the DCT being performed, to produce the corresponding values in the output matrix.

Once the lower level transforms and operations have being completed the resulting data is then further manipulated by an Output Adder Unit (OAU) to complete the transformation. A stripped down representation of this process can be seen in Figure 2.5, where the horizontal lines represent the input and manipulated data after operations and the dots represent the addition or subtraction when a (-) is placed beneath it. this continues until the out put is reached. The stages are divided up by the red vertical lines, with the last section being similar to OAU as it only performs a simple operation.

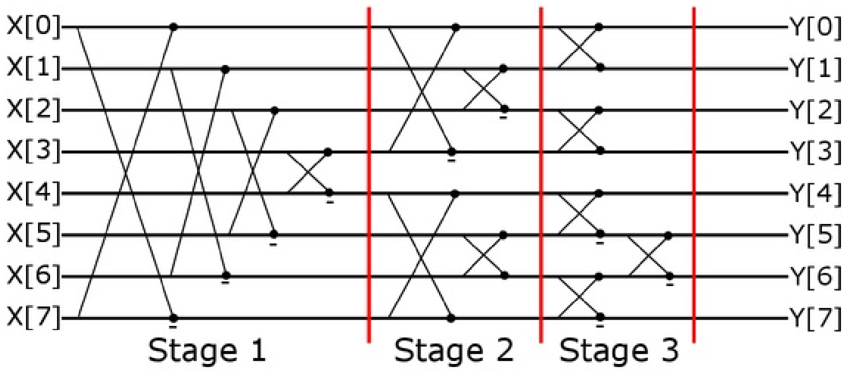


Figure 2.5: Stick diagram of the Butterfly technique applied to the DCT

The complete architecture implementation, Figure 2.6, operates recursively by gradually calling N/2-point DCTs until it reaches the four-point DCT. It uses a control unit to select a mode of operation dependent on the length of the input. It uses multiplexers on the outputs when some are not required.

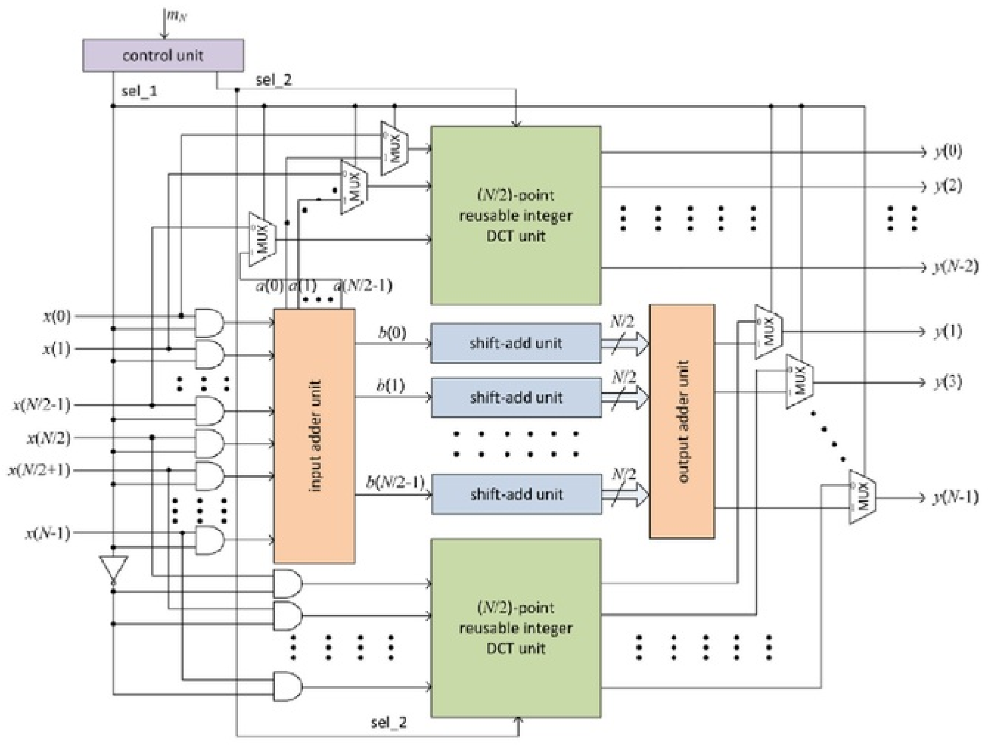


Figure 2.6: Proposed variable length DCT architecture [4]

The 2D-DCT is then calculated by using a two 1D-DCT modules with a transpose module in between, to convert the acquired row transforms into column order to perform the transforms second stage using the same unit.

2.2.2 Masera et al's Proposed Algorithm

The algorithm proposed by Masera et al, uses Walsh-Hadamard Transform (WHT) followed by Givens rotations to compute the DCT. This can be exploited to compute four different approximations by selectively skipping the appropriate Givens rotations [3]. By removing the options to skip certain rotations, this method can be converted into an algorithm that purely calculates the DCT.

The 2D-DCT implementation proposed uses a singular 1D-DCT and transposition memory as well as some other minor manipulations to prepare the data coming out of the 1D-DCT to either be stored within the memory or sent to the output signals. This process is defined in Figure 2.7. This architecture will have a smaller implemented area then that proposed by Meher et al [4] as it only uses a single 1D-DCT unit and a multiplexer (MUX) to complete the 2D-DCT instead of two complete 1D-DCT units.

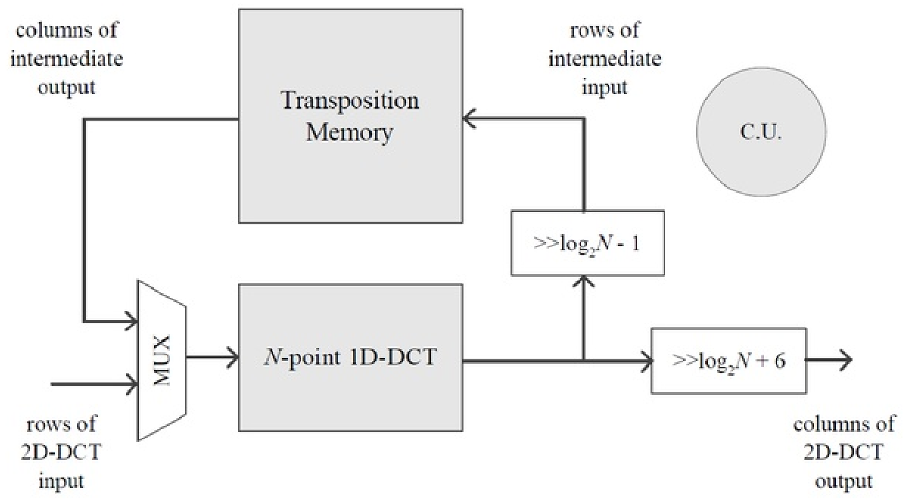


Figure 2.7: Masera et al's proposed 2D-DCT architecture [3]

Chapter 3

One Dimensional Discrete Cosine Transform Modules

3.1 Introduction

The purpose of this chapter is to give an overview of the development and resulting implementation of VHDL modules that complete the One Dimensional Discrete Cosine Transform (1D-DCT) for input lengths of 4, 8 16 and 32 points. This will include an explanation of the design method chosen, and in some cases alternate or original implementations, of each DCT length as well as reasoning behind their selection. These modules will then be used to construct the variable size one dimensional module which will be used to within the two dimensional model that is required for HEVC.

The modules that will be examined in this section include the Four-Point, Eight-Point, Sixteen-Point and ThirtyTwo-Point modules as well as their respective helper functions used to perform particular calculations throughout each module.

3.2 VHDL Coding Decisions

For the data types within VHDL, the std_logic_vector type is used for the inputs and outputs of each module to represent the incoming integers in binary form. This was decided such that they could be easily manipulated through the shifting operations while still maintaining the ability to add the vectors together, this also allows for easier conversion between the models presented by Meher et al [4]. All std_logic_vectors are of size 7 downto 0 such that they could represent an 8 bit binary number which was decided to be sufficient.

For some module implementations, it was decided that it would be beneficial to use a VHDL function to replace repetitive calculations that used more then one line in obtaining the resulting value. As a result some subsidiary functions were created to calculate specific intermediate values with the aim of reducing the total implementation time and creating a more interpretable design for others to follow the implemented design.

For the 1D-DCT Modules it is possible to use combination logic to perform the calculations as they are not heavily dependent on timing and therefore will not require the presence of a CLK to trigger the next stage. The process that performs the calculations in each of the modules is sensitive to the inputs of the respective module such that it will recalculate the outputs each time an input value is altered.

3.3 Four-Point DCT and Subsidiary Functions

3.3.1 Introduction

This section of the document will cover the development and implementation of the four-point (4X4) 1D-DCT. This is the base model of the entire implementation and will be used in every calculation as the end point of the stacked process of the reusable architecture. This module is based on the proposed algorithm by Meher et al [4]. The module was to be supported by various VHDL functions that would perform repeated operations such as shifting and adding the input and processed data at various times throughout the transform.

3.3.2 Algorithm

The algorithm used to implement the 1D-DCT for a 4 x 4 matrix is outlined in stages in Table 3.1. The algorithm is broken into the Input Adder Unit (IAU), Shift Adder Unit (SAU) and the Output Adder Unit (OAU) such that each stage can be examined and implemented individually to ensure necessary values are available as each stage completes. This algorithm was extracted from the larger algorithm proposed by Meher et al for their multiple length algorithm, as such it is intended that the four-point module produced will be used in a similar manner [4].

Within the table, the X(i) values represent the inputs and the Y(i) values the outputs of the algorithm, with $i = 0to3$, such that there are four inputs and outputs.

This algorithm is used to replicate the kernel matrix represented by Equation 3.1 [9] such to perform the transform without directly performing matrix multiplication. This is done to improve the computational speed and efficiency of the architecture.

$$C_4 = \begin{bmatrix} 64 & 64 & 64 & 64 \\ 83 & 36 & -36 & -83 \\ 64 & -64 & -64 & 64 \\ 36 & -83 & 83 & -36 \end{bmatrix} \tag{3.1}$$

3.3.3 Subsidiary Functions

These functions were developed first to represent and implement the basic functions of the architecture, mainly the SAU unit and the right shift operators.

Table 3.1: Four-Point DCT Algorithm by Stage

Four Point DCT			
Stage	Computation	Binary Expression	Other Notes

Stage 1 (IAU)	a(i) = x(i) + x(3-i)
	b(i) = x(i) - x(3-i)
			For i = 0 to 3

Stage 2 (SAU)	$m_{i,9} = 9b(i)$ $m_{i,64} = 64a(i)$ $t_{i,83} = 83b(i)$ $t_{i,36} = 36b(i)$	$(b(i) << 3) + b(i)$ $a(i) << 6$ $(b(i) << 6) + (m_{i,9} << 1) + b(i)$ $m_{i,9} << 2$	For i = 0 to 3
Stage 3 (OAU)	$y(0) = t_{0,64} + t_{1,64}$ $y(1) = t_{0,83} + t_{1,36}$ $y(2) = t_{0,64} - t_{1,64}$ $y(3) = t_{0,36} - t_{1,83}$		

Input Adder Unit

This unit is split into two functions that perform the two operations of the IAU, vector addition and vector subtraction. The first function, PPartial, is used to calculate the addition of the two input vectors, for the Four-point IAU this adds X(0) & X(3) and X(1) & X(2). The second function, NPartial, is used to calculate the subtraction of the one input vector from the other, for the four-point IAU this subtracts X(3) from X(0) and X(2) from X(1).

The functions are both built using the standard operators of the numeric std logic packages. Both of these functions can be found under Appendix A.1.2.

Four-Point Shift-Add Unit

The SAU for the Four-Point DCT is implemented using the structure diagram presented in Figure 3.1.

The SAU requires the ability to shift left to complete, the first approach used towards this was to use the in built arithmetic shift left (sla) to complete the shifts, however this function was not compatible with the std_logic_vector resulting in the need for a shift function to be developed.

The unit is divided into two functions such that the middle value, up until the second split pointin Figure 3.1 only has to be calculated once when called from the Four-point DCT module. This function computes the three left shifts, addition with the original and then a single left shift, from here the i performed another shift left to obtain the the $t_{i,36}$ intermediate value.

A second function is used to call the six shift left operation and final addition of the previously created intermediate value and the original to obtain $t_{i,83}$.

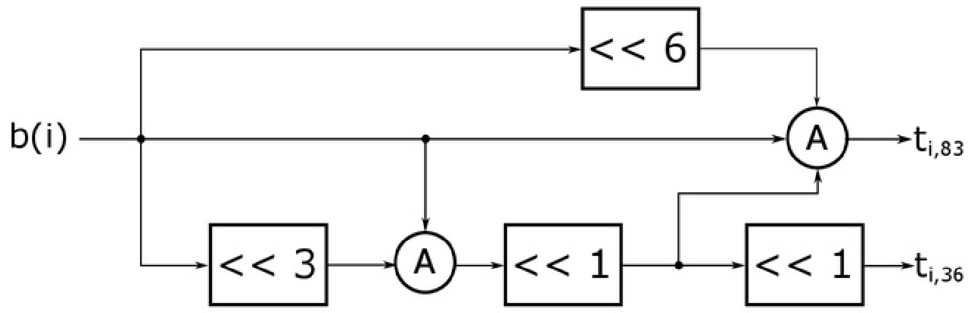


Figure 3.1: Structure of the 4-point SAU

The two intermediate values are then fed back into the output adder unit to complete the transformation.

Output Adder Unit

The purpose of this unit is to manipulate the output values to complete the DCT correctly. This unit uses the PPartial and NPartial functions created for the IAU to complete the Operation of the OAU. Another function that called both of the partial functions was considered such that only one function call was made by the module, however it was decided that it would perform the calculations easier if the partials were just called within the module itself. The subsidiary function were able to be used here due the simple addition and subtraction of only two inputs, however this will not be as easily completed in later units due to multiple inputs and varying operators.

The use of the subsidiary functions is not completely necessary due to the simplicity of the calculation, however they were implemented before this realization was made and it was decided that they would remain as it had minimal effect on the units performance and efficiency.

3.3.4 Four-Point Module

This module is the top level executor that performed the Four-point DCT by calling the subsidiary functions for the IAU, SAU and OAU while performing simple instructions that do not require a function. All code for this section is based on the set values outlined in Table 3.1.

The code is be executed within a VHDL process that is triggered by a change in the input signal, that is it will run each time there is a change in any of the input signals. The intention of this method is to complete the operations using combination logic without the use of a clock to drive the hardware at this level. A clock signal (CLK) was originally used as the driver for the process and was controlled using a series of if statements controlled by the 'currentState' signal that defined what stage of the DCT was to be completed at each clock cycle, however this was deemed to be unnecessary and was therefore altered to the combination logic technique.

This method has some drawbacks when the IO data is not managed correctly, such that if incoming data is not loaded simultaneously it will begin its calculations early which could cause an incorrect output to be recorded for use in a later section. The solution to this problem is to manage the inputs correctly when using an implemented version of the module.

The implemented module will follow the method represented by Figure 3.2.

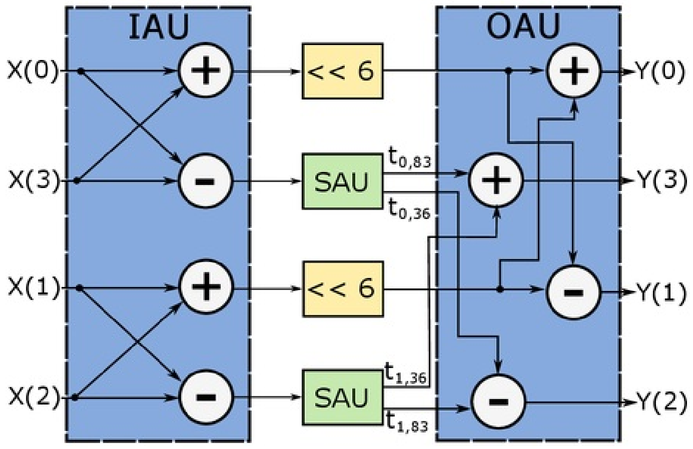


Figure 3.2: Visualization of the Four-Point DCT module

The IAU is first performed using the PPartial and NPartial creating the values in IAU section of Table 3.2. These values then feed into the SAU functions to create the inputs for the OAU. These values are as displayed in Table 3.2.These finally pass through the output adder unit to give the output values as shown in the OAU section of Table 3.2

This module is used throughout the rest of the project as the first stage of larger point DCTs as well as to be incorporated twice as part of the 2D-DCT for the 4x4 matrix. This is be outlined in the 2D-DCT chapter of this report.

Table 3.2: VHDL Code for the 4Point Module

Stage	VHDL Implementation
IAU	a(0): X(0) + X(3)
	b(0): X(0) - X(3)
	a(1): X(1) + X(2)
	b(1): X(1) - X(2)
SAU	$t_{0,64} : a(0) << 6$
	$t_{0,83}$: SAU(b(0)) Output 1
	$t_{0,36}$: SAU(b(0)) Output 2
	$t_{1,64} : a(1) << 6$
	$t_{0,83}$: SAU(b(1)) Output 1
	$t_{0,36}$: SAU(b(1)) Output 2
OAU	y(0): $t_{0,64} + t_{1,64}$
	y(1): $t_{0,83} + t_{1,36}$
	y(2): $t_{0,64} - t_{1,64}$
	y(3):$t_{0,36} - t_{1,83}$

3.4 Eight-Point DCT

3.4.1 Introduction

The purpose of this chapter is to give an overview and explanation of the development and resulting architecture of the eight-point (8x8) 1D-DCT module. This module's main purpose is to perform the DCT for an 8x8 matrix, from which the result is then used in the 2D-DCT as well as the 16 and 32-point modules. The module completes the transform using a combination of the four point module and shift adders to compute the resulting output signals.

3.4.2 Eight-Point Algorithm

Similarly to the algorithm for the Four-point DCT, the algorithm that has been used to implement part of the 1-D DCT for an 8x8 matrix is outlined by stage in Table 3.3. This algorithm was also extracted from that of Meher et al's variable length algorithm such that it can be examined and implemented as its own functional module [4].

The algorithm outlined in Table 3.3 deals with producing half of the output signals or values of the eight-point DCT. After the IAU generates the a(i) and b(i) values for $i = 0to3$, the b(i) values are then manipulated by the SAU and OAU appropriately while the a(i) values are passed to a copy of the four-point DCT to compute the remaining output values.

Table 3.3: Eight-Point DCT Algorithm by Stage

Eight Point DCT			
Stage	Computation	Binary Expression	Other Notes

Stage 1 (IAU)	a(i) = x(i) + x(7-i)
	b(i) = x(i) - x(7-i)
			For i = 0 to 3

Stage 2 (SAU)	$m_{i,9} = 9b(i)$ $m_{i,25} = 25b(i)$ $t_{i,18} = 18b(i)$ $t_{i,50} = 50b(i)$ $t_{i,75} = 75b(i)$ $t_{i,89} = 89b(i)$	$(b(i) << 3) + b(i)$ $(b(i) << 4) + m_{i,9})$ $m_{i,9} << 1$ $m_{i,25} << 1$ $t_{i,50} + m_{i,25}$ $(b(i) << 6) + m_{i,25}$	For i = 0 to 3
Stage 3 (OAU)	$y(1) = t_{0,89} + t_{1,25} + t_{2,50} + t_{3,18}$ $y(3) = t_{0,75} - t_{1,18} - t_{2,89} - t_{3,50}$ $y(5) = t_{0,50} - t_{1,89} + t_{2,18} + t_{3,75}$ $y(7) = t_{0,18} - t_{1,50} + t_{2,75} - t_{3,89}$		

3.4.3 Eight-Point VHDL Module

The Eight-Point VHDL module builds upon the existing Four-Point module to perform the DCT using eight inputs. The module was broken down into the same three stages (IAU, SAU and OAU) as the Four-Point module such that they were able to be implemented and examined more effectively. The module follows the architecture shown in Figure 3.3, the design and function of the units shown in this figure are outlined below.

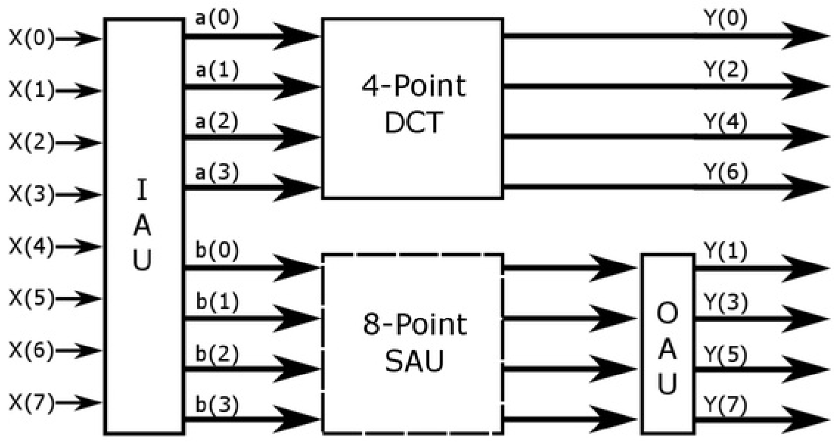


Figure 3.3: Visualization of the 8-Point DCT module's architecture

Eight-Point Input Adder Unit

The IAU functions almost the same way as the corresponding IAU in the Four-Point module as it calculates the sum and subtraction of the values x(i) and x(7-i), where 0 to 7 represent the eight inputs of the module. The addition results are stored as a(i) and the subtractions stored as b(i) for i = 0 to 3. Unlike the previous module, functions were not used to complete these calculations as they are simple operations, in a small number, that could be completed efficiently without the need for a function.

Eight-Point Shift Adder Unit

The SAU for the Eight-Point module works similarly to that of the Four-Point module, not all values generated by the IAU stage are used in this section. The b(i) values generated in the IAU will be manipulated by the SAU while the a(i) values will be processed by an implemented Four-Point module.

The SAU for this module is more complicated then that of the Four-Point as the increase in input size requires the transform to use a greater range of multiplications to complete. The transform can be expressed as a kernel matrix as shown by Equation 3.2.

$$C_8 = \begin{bmatrix} 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 \\ 89 & 75 & 50 & 18 & -18 & -50 & -75 & -89 \\ 83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 \\ 75 & -18 & -89 & -50 & 50 & 89 & 18 & -75 \\ 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 \\ 50 & -89 & 18 & 75 & -75 & -18 & 89 & -50 \\ 36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 \\ 18 & -50 & 75 & -89 & 89 & -75 & 50 & -18 \end{bmatrix} \tag{3.2}$$

This results in a larger number of partial values to calculate, to represent each multiplication, these values are then stored such that they can be reused to replace larger multiplications with addition or subtractions of the previously calculated multiplications. An example of this would be multiplying by 75 which can be calculated by adding the existing results of multiplying by 50 and 25, $t_75 = t_50 + t_25$. This is very beneficial in regards to computational efficiency due to addition operation being far more efficient than multiplication and the previous values being readily available from previous calculations.

This method is more area efficient then completing the transform using dedicated architecture as it will 'reuse' existing architecture, however this creates a small delay in which the values are passed and the outputs are retrieved from the unit. This is solved by ensuring the output values of the unit are taken once the lower level unit is completed, this will be relevant for all modules from this point as they will all call lower level DCT modules to complete their operation.

The reusable architecture is evident in the Eight-Point SAU where each addition is able to build on the next with some minor manipulations to create the next intermediate value to be acted on by the OAU. This is visualized Figure 3.4, where the addition represented by The A's flow into each other from the top to bottom while being altered slightly at each

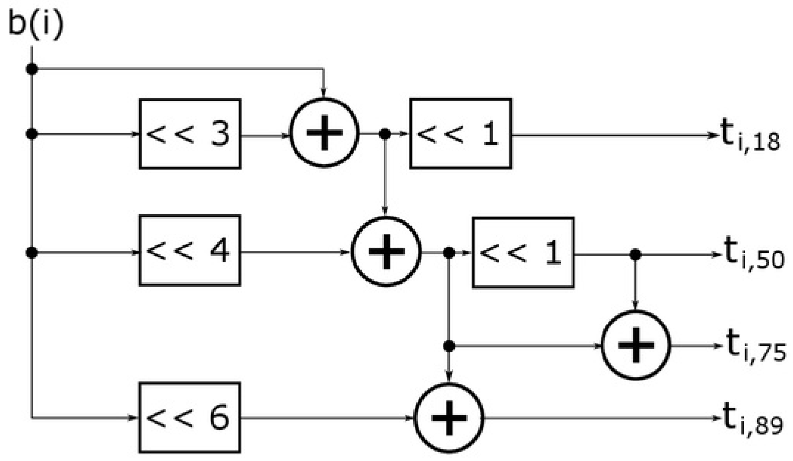


Figure 3.4: Visualization of the Eight Point SAU

stage to give a particular intermediate value. These stages were implemented as VHDL functions as they were used multiple times and required multiple lines of code for each value. These results were then used to compute the remaining values through addition and shifting to attain the remaining intermediate values.

Eight-Point Output Adder Unit

Similarly to that of the Four-Point, the Eight-Point OAU is quite simple in operation as it only performs basic addition and subtraction between four intermediate values to obtain the output values for the odd numbered index values. These are specified by the corresponding OAU equations in Table 3.3.

3.5 Sixteen-Point DCT

3.5.1 Introduction

The purpose of this chapter is to overview and explain key concepts and developments of the Sixteen Point (16x16) 1D-DCT module, Sixteen-Point. Similarly to the Eight-Point module, the Sixteen-Point will use a combination of shift adders and shorter length DCT modules to complete the transform.

3.5.2 Algorithm

The algorithm was extracted from Meher et al's variable length algorithm, such that the Sixteen Point module could be implemented as a stand alone module [4]. The extracted algorithm is represented in Table 3.4. The unit contains a functional copy of the Eight-Point module, and inherently the Four-Point module, to complete the even indexed output signal computations using the generated addition results from the IAU. The remaining IAU subtraction results will be processed by the SAU to generate the odd indexed output values.

3.5.3 Sixteen Point Module Architecture

By expanding the model of the Eight-Point Architecture in Figure 3.3, a reusable architecture is extracted for use with the remaining DCT lengths was developed, using similar methods to Meher et al [4]. This model uses an N/2 DCT module and a SAU specifically designed for each length to complete the transform. Purpose built IAUs and OAUs were also defined for each length, as this was the simpler and most efficient implementation of both units. A diagram overview of the reusable architecture is shown in Figure 3.5, while the design of supplementary units (IAU, SAU, OAU) are defined within the sections of their corresponding modules.

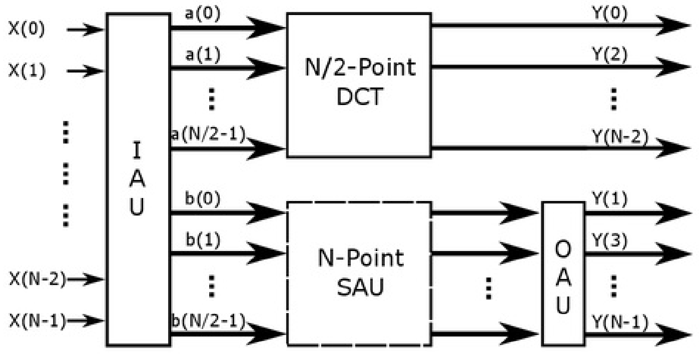


Figure 3.5: A generalized structure of larger length length DCTs, where N = 8, 16, 32 [4].

Table 3.4: Sixteen-Point DCT Algorithm by Stage

Sixteen Point DCT			
Stage	Computation	Binary Expression	Other Notes
Stage 1 (IAU)	a(i) = x(i) + x(7-i)		For i = 0 to 7
	b(i) = x(i) - x(7-i)		
Stage 2 (SAU)	$m_{i,8} = 8b(i)$	$b(i) << 3$	For i = 0 to 7
	$t_{i,9} = 9b(i)$	$m_{i,8} + b(i)$	
	$m_{i,18} = 18b(i)$	$t_{i,9} << 1$	
	$m_{i,72} = 72b(i)$	$t_{i,9} << 3$	
	$t_{i,25} = 25b(i)$	$(b(i) << 4) + t_{i,9}$	
	$t_{i,43} = 43b(i)$	$m_{i,18} + t_{i,25}$	
	$t_{i,57} = 57b(i)$	$(b(i) << 5) + t_{i,25}$	
	$t_{i,70} = 70b(i)$	$(b(i) << 6) - t_{i,70}$	
	$t_{i,80} = 80b(i)$	$m_{i,72} + m_{i,8}$	
	$t_{i,87} = 87b(i)$	$(t_{i,43} << 6) + b(i)$	
	$t_{i,90} = 90b(i)$	$m_{i,72} + m_{i,18}$	
Stage 3 (OAU)	$\mathrm{y}(1) = t_{0,90} + t_{1,87} + t_{2,80} + t_{3,70} + t_{4,57} + t_{5,43} + t_{6,25} + t_{7,9}$		
	$\mathrm{y}(3) = t_{0,87} + t_{1,57} + t_{2,9} - t_{3,43} - t_{4,80} - t_{5,90} - t_{6,70} - t_{7,25}$		
	$\mathrm{y}(5) = t_{0,80} + t_{1,9} - t_{2,70} - t_{3,87} - t_{4,25} + t_{5,57} + t_{6,90} + t_{7,43}$		
	$\mathrm{y}(7) = t_{0,70} - t_{1,43} - t_{2,87} + t_{3,9} + t_{4,90} + t_{5,25} - t_{6,80} - t_{7,57}$		
	$\mathrm{y}(9) = t_{0,57} - t_{1,80} - t_{2,25} + t_{3,90} - t_{4,9} - t_{5,87} + t_{6,43} + t_{7,70}$		
	$\mathrm{y}(11) = t_{0,43} - t_{1,90} + t_{2,57} + t_{3,25} - t_{4,87} + t_{5,70} + t_{6,9} - t_{7,80}$		
	$\mathrm{y}(13) = t_{0,25} - t_{1,70} + t_{2,90} - t_{3,80} + t_{4,43} + t_{5,9} - t_{6,57} + t_{7,87}$		
	$\mathrm{y}(15) = t_{0,9} - t_{1,25} + t_{2,43} - t_{3,57} + t_{4,70} - t_{5,80} + t_{6,87} - t_{7,90}$		

3.5.4 Sixteen Point Input Adder Unit

The IAU for the Sixteen Point module follows the same structure as the previous IAU's in the values a(i) and b(i) are generated by computing the addition and the subtraction respectively for $X(i)$ and $X((N/2) - 1 - i)$, in this case N = 16. The same method of computing the values at time of assignment is used here again for the same reasons given for the Eight-Point IAU due to the number of calculations still being relatively small when compared to the project as a whole.

3.5.5 Sixteen Point Shift Adder Unit

The Sixteen Point SAU produces more intermediate values then that of the previous modules due to the increase in inputs resulting in a greater range of multipliers. The output values are defined in Stage 2 of Table 3.4. The unit also reuses previously calculated results in later multiplications such to increase the efficiency of the architecture.

3.5.6 Sixteen Point Output Adder Unit

The Sixteen Point OAU works in a similar manner to those in the Four-Point and Eight-Point modules, in which output values are calculated using a series of addition and subtractions of specific intermediate values to obtain the result. For the Sixteen-Point module this involves the resulting calculation being determined by a set of eight values being manipulated using varying operation patterns depending on the output value being determined, these are defined by Stage 3 of Table 3.4. Due to the varying nature of operations and a undefinable pattern, a suitable VHDL function could not be designed in a way that could be used in a general case therefore the outputs were all calculated at the point of assignment as it was equally efficient and required less overall implementation time.

3.6 Thirty-Two Point DCT Module

3.6.1 Introduction

This purpose of this section of the document is to give an overview and insight into the design and implementation for the Thirty-Two Point (32x32) 1D-DCT Module. The purpose of this unit is to successfully perform the DCT for a 32x32 matrix, which will then be used to complete the two dimensional transform as well as being the base of the variable length DCT module. The module achieves this purpose using a combination of smaller length DCT modules and shift adders.

3.6.2 Algorithm

The algorithm for the Thirty-Two Point module that computes the odd indexed output values is outlined in Table 3.5 and extracted from Meher et al's variable length algorithm [4]. The even indexed outputs, including Y(0), are calculated by an implemented Sixteen Point module. The use of the Sixteen Point module inherently utilizes the Eight-Point and Four-Point to perform the lower level calculations as outlined in their respective sections.

Table 3.5: 32-Point DCT Algorithm by Stage

Thirty-Two Point DCT			
Stage	Computation	Binary Expression	Other Notes

Stage 1 (IAU)	a(i) = x(i) + x(15-i)
	b(i) = x(i) - x(15-i)
			For i = 0 to 15

Stage 2 (SAU)	$m_{i,2} = 2b(i)$	$b(i) << 1$	
	$m_{i,4} = 4b(i)$	$b(i) << 2$	
	$t_{i,9} = 9b(i)$	$(b(i) << 3) + b(i)$	
	$m_{i,18} = 18b(i)$	$t_{i,9} << 1$	
	$m_{i,72} = 72b(i)$	$t_{i,9} << 3$	
	$t_{i,13} = 13b(i)$	$t_{i,4} + t_{i,9}$	
	$m_{i,52} = 52b(i)$	$t_{i,13} << 2$	
	$t_{i,22} = 22b(i)$	$t_{i,4} + m_{i,18}$	
	$t_{i,31} = 31b(i)$	$(b(i) << 5) - b(i)$	
	$t_{i,38} = 38b(i)$	$t_{i,9} << 2 + m_{i,2}$	For i = 0 to 15
	$t_{i,46} = 46b(i)$	$t_{i,22} << 1 + m_{0,2}$	
	$t_{i,54} = 54b(i)$	$m_{i,52} + m_{i,2}$	
	$t_{i,61} = 61b(i)$	$(t_{i,31} << 1) + b(i)$	
	$t_{i,67} = 67b(i)$	$t_{i,54} + t_{i,13}$	
	$t_{i,73} = 73b(i)$	$m_{i,72} + b(i)$	
	$t_{i,78} = 78b(i)$	$t_{i,13} << 1 + m_{i,52}$	
	$t_{i,82} = 82b(i)$	$t_{i,4} + t_{i,78}$	
	$t_{i,85} = 85b(i)$	$t_{i,13} + m_{i,18}$	
	$t_{i,88} = 88b(i)$	$t_{i,22} << 2$	
	$t_{i,90} = 90b(i)$	$m_{i,72} + m_{i,18}$	
Stage 3 (OAU)	$y(1) = t_{090} + t_{190} + t_{288} + t_{385} + t_{482} + t_{578} + t_{673} + t_{767} + t_{861} + t_{954} + t_{1046} + t_{1138} + t_{1231} + t_{1322} + t_{1413} + t_{1504}$		
	$y(3) = t_{090} + t_{182} + t_{267} + t_{346} + t_{422} - t_{5,4} - t_{631} - t_{754} - t_{873} - t_{985} - t_{1090} - t_{1188} - t_{1278} - t_{1361} - t_{1438} - t_{1513}$		
	$y(5) = t_{088} + t_{167} + t_{231} - t_{313} - t_{454} - t_{582} - t_{690} - t_{778} - t_{846} - t_{94,} + t_{1038} + t_{1173} + t_{1290} + t_{1385} + t_{1461} + t_{1522}$		
	$y(7) = t_{085} + t_{146} - t_{213} - t_{367} - t_{490} - t_{573} - t_{622} + t_{738} + t_{882} + t_{988} + t_{1054} - t_{114,} - t_{1261} - t_{1390} - t_{1478} - t_{1531}$		
	$y(9) = t_{082} + t_{122} - t_{254} - t_{390} - t_{461} + t_{513} + t_{678} + t_{785} + t_{831} - t_{946} - t_{1090} - t_{1167} + t_{124,} + t_{1373} + t_{1488} + t_{1538}$		
	$y(11) = t_{078} - t_{1,4} - t_{282} - t_{373} + t_{413} + t_{585} + t_{667} - t_{722} - t_{888} - t_{961} + t_{1031} + t_{1190} + t_{1254} - t_{1338} - t_{1490} - t_{1546}$		
	$y(13) = t_{073} - t_{131} - t_{290} - t_{322} + t_{478} + t_{567} - t_{638} - t_{790} - t_{813} + t_{982} + t_{1061} - t_{1146} - t_{1288} - t_{134,} + t_{1485} + t_{1554}$		
	$y(17) = t_{061} - t_{173} - t_{246} + t_{382} + t_{431} - t_{588} - t_{613} + t_{790} - t_{84,} - t_{990} + t_{1022} + t_{1185} - t_{1238} - t_{1378} + t_{1454} + t_{1567}$		
	$y(19) = t_{054} - t_{185} - t_{24,} + t_{388} - t_{446} - t_{561} + t_{682} + t_{713} - t_{890} + t_{938} + t_{1067} - t_{1178} - t_{1222} + t_{1390} - t_{1431} - t_{1573}$		
	$y(21) = t_{046} - t_{190} + t_{238} + t_{354} - t_{490} + t_{531} + t_{661} - t_{788} + t_{822} + t_{967} - t_{1085} + t_{1113} + t_{1273} - t_{1382} + t_{144} + t_{1578}$		
	$y(23) = t_{038} - t_{188} + t_{273} - t_{34,} - t_{467} + t_{590} - t_{646} - t_{731} + t_{885} - t_{978} + t_{1013} + t_{1161} - t_{1290} + t_{1354} + t_{1422} - t_{1582}$		
	$y(25) = t_{031} - t_{178} + t_{290} - t_{361} + t_{44,} + t_{554} - t_{688} + t_{782} - t_{838} - t_{922} + t_{1073} - t_{1190} + t_{1267} - t_{1313} - t_{1446} + t_{1585}$		
	$y(27) = t_{022} - t_{161} + t_{285} - t_{390} + t_{473} - t_{538} - t_{64,} + t_{746} - t_{878} + t_{990} - t_{1082} + t_{1154} - t_{1213} - t_{1331} + t_{1467} - t_{1588}$		
	$y(29) = t_{013} - t_{138} + t_{261} - t_{378} + t_{488} - t_{590} + t_{685} - t_{773} + t_{854} - t_{931} + t_{104,} + t_{1122} - t_{1246} + t_{1367} - t_{1482} + t_{1590}$		
	$y(31) = t_{04,} - t_{113} + t_{222} - t_{331} + t_{438} - t_{546} + t_{654} - t_{761} + t_{867} - t_{973} + t_{1078} - t_{1182} + t_{1285} - t_{1388} + t_{1490} - t_{1590}$		

3.6.3 Thirty-Two Point Input Adder Unit

The IAU for the Thirty-Two Point IAU follows the same structure as the previous IAU's such that the values a(i) and b(i) are generated by computing the addition and the subtraction respectively for X(i) and $X((N/2) - 1 - i)$, in this case N = 32. The same method of computing the values at time of assignment is used here again due to the number of calculations still being relatively small when compared to the project as a whole.

3.6.4 Thirty-Two Point Shift Adder Unit

The Thirty-Two Point SAU uses similar methods to those of the smaller length DCT by using shifts and additions to perform multiplications, while also reusing previously calculated results to perform larger multiplications. In the Thirty-Two Point case however there are many of these multiplications to be completed as there are twenty multipliers that each need to be applied to sixteen different inputs, giving a total of 320 values to be calculated. Due to the repetitive nature of these calculations, VHDL functions were created to calculate and perform basic operations require anything more than a single operator, such as $x << 1$, as well as to compute values that required multiple lines of code. Due to the larger number of calculations some variables initially had the same name following the naming pattern that was implemented throughout the project, for example t152 represented both the multiplications t15(2) and t1(52). The solution to this was to make exceptions for these cases as there were only several instances of this problem rather then altering the naming scheme for the entire project, this was usually done by adding a '0' in the appropriate case i.e t1502 to create a variation for the previously given example

3.6.5 Thirty-Two Point Output Adder Unit

This stage of the module uses the calculated results of the intermediate multiplications from the SAU to generate the output values for the odd indexed signals. The calculations are quite simple only using addition and subtractions operations, however there are fifteen operation to be performed per output value without an obvious pattern to the order of operations from one signal to the next. As a result of this a VHDL functions was not appropriate and each signal was calculated at the time of assignment according to the respective calculations defined in Stage 3 of Table 3.5. This calculation initially contained a few minor errors caused by incorrect operators being used in several of the output calculations, once the source of the errors were discovered they were quickly corrected to give the expected output value.

Chapter 4

Variable Length Discrete Cosine Transform Module

4.1 Introduction

HEVC allows for the use of a variable block length (4, 8, 16 and 32) to accommodate for this it is beneficial to create a module that is capable of performing the DCT for these common block sizes. This module will allow for the same architecture to be used to process any block size to act as a foundation for the two dimensional DCT. This section outlines the development of this module and its related concepts, such as length selection.

4.2 Length Selection

To implement the variable length module it was necessary to implement a method that allowed for the unit to determine the appropriate length DCT to use for the incoming data. The initial attempt at this was to use only the 32 input signals, X(0) to X(31) and examine specific signals to determine if they were used or not. The signals to be examined were X(4), X(8) and X(16), as by examining the input on these signals conclusions would able to be drawn whether data existed beyond the threshold of each length DCT such that only the largest size necessary would be used. However this method proved to be ineffective in cases where the data on these points happened to be 00000000 as this was the default assigned value that was being checked against to determine if it was being used or not. It was considered to check the next values in this case but a similar problem could arise and there would be more efficient methods then checking each value.

The method which was implemented was to use an extra 2-bit std_logic_vector signal named 'SEL' with allowed for four different values to be stored, each of which was assigned to one length DCT as expressed in Table 4.1. This allows for the user to predetermine which length DCT is used to suit the needs of their input data which can then be easily examined by the top level module and passed into lower level modules, if necessary, to create the appropriate result. As there still exists 32 input signals, for the modules lower

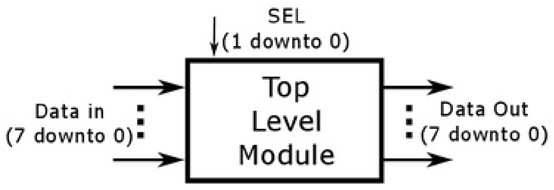


Figure 4.1: 'SEL' selection method diagram

Table 4.1: Variable Length DCT SEL Assignments

DCT Module	Length	SEL Value
Four-Point	4	00
Eight-Point	8	01
Sixteen-Point	16	10
ThirtyTwo-Point	32	11

then the ThirtyTwo-Point, the output signals will be given a default value of 00000000 for use in other modules such as the Two-dimensional DCT. This idea is briefly illustrated in Figure 4.1, in which the types are all std_logic_vector where the numbers represent the sizes of the the vectors.

4.3 VHDL Implementation

For the implementation of this model, two different implementations were considered each having its own advantages in terms of ease of implementation and efficiency. The two models are outlined briefly in this section.

4.3.1 Direct Access Implementation

This implementation uses a top-level module of 32 input and 32 output signals which instantises each of the 4 length DCT modules such that the input data can be directed to the appropriate module and performed quickly. The approach is quite simple and effective, however it is lacking efficiency due to the fact that multiple versions of the lower length DCTs will be created within each of the larger modules. This is not ideal as the architecture generated for the other modules that are not currently in use will be wasted in most cases and will not totally utilize the architecture without restructuring the existing modules. The basic layout of this method is visualized in Figure 4.2, where the numbers after the X and Y's respectively representing the output and inputs of the numbered length DCT.

It would be possible to make better use of this architecture by increasing the number of input and output signals such that they could be assigned to each individual module's IO

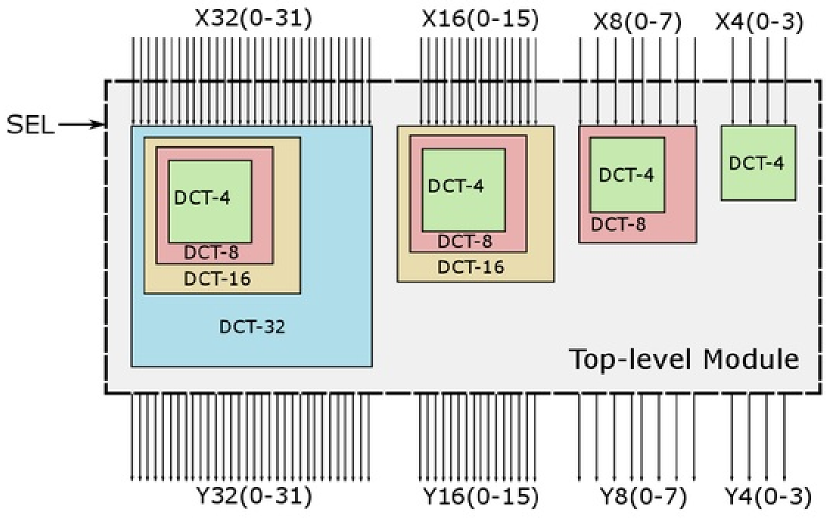


Figure 4.2: Diagram representing the Direct Access Implementation

signals, this would allow for multiple sizes to be processed simultaneously. This scenario seems unlikely as the data input would generally be the same size when performing the DCT on a single source, this means that to use the architecture in this way it would be likely processing data from multiple sources. Processing data in this way increases the chance if an error if data is mishandled when being processed outside of the module, such as when being stored for data_out or in preparation to perform the transpose before the second round of the DCT for the two dimensional case required for HEVC.

Another way that this implementation could be utilized would be to use each of the larger modules to perform lower length DCT simultaneously though the existing copies in the larger, similarly to the method described in the Layered Implementation section. For example, the Four-Point module could be running four at a time by passing the values through the larger length modules until it reaches their Four-Point module. While this method has potential, all of the modules will be running with different timings as it passes through the modules creating points of overlap. This could create similar problems of storing the output data correctly, such that it is in the correct position within the matrix, as well as timing what data is to be given to each module at each time. Additional calculations would need to be completed at each stage before implementation to determine when and which data would be passed to each module to ensure the correct output is generated.

4.3.2 Layered Implementation

This implementation uses either a top level module or alters the Thirty-Two Point Module such that data can be passed through the larger sized DCT modules to access the smaller modules within them to perform the appropriate length DCT. Due to 32-point DCT containing a version of each module, it is the only module that is required to be used within this model. This design is more area efficient than the previous model as there are only a total of 4 module architectures in use compared to the 10 used in the direct access method. A visulaization nof this model is shown in Figure 4.3.

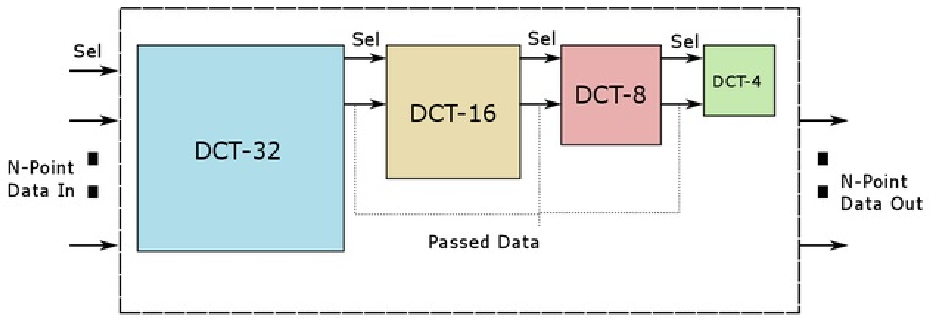


Figure 4.3: Diagram representing the Layered Variable Size Implementation

However this method would require modification of each module. This is necessary to handle data being passed to a lower length module through a higher level module, this is done to bypasses the IAU such that the data is not altered before reaching the appropriate module. Preventing the SAU and OAU from being used unnecessarily would also be beneficial, however this is not a priority as the generated output signals could be ignored when passing data out of the module.

Using multiplexers (MUX) to accomplish this goal is suitable with minimal modifications to the existing modules, allowing for control of the input into the lower level modules and the IAU. These are easily implemented as 'if' statements in VHDL, effectively gating the unnecessary architecture when there is no use for it. The signal controlling these MUXs will be the SEL signal described in Section 4.2, which the inputs of all modules will be modified to accept and mange this signal as an input. Each module will check if it is the module to be used according to the assigned values in Table 4.1, if the current module it is to be used it functions as normal if else it passes the input into the next module without modification.

This method is preferred as there is lower risk of the mishandling of data while also being more area efficient, allowing for the architecture to require less physical area when implemented while also requiring less power to perform the transform. It does lack in computational speed/efficiency when compared tot he potential of the direct access method,however the benefits of using this design outweigh the negatives of the direct access design.

Chapter 5

Two Dimensional Discrete Cosine Transform Module

5.1 Introduction

When processing two dimensional signals, such as video frames, it is necessary to use a two dimensional version of the DCT (2D-DCT) to complete the process. Due to the trivial expansion of the 1D-DCT the 2D-DCT is separable into two steps, the 1D-DCT of the columns of the signal followed by a 1D-DCT of the rows of the resulting rows. The 2D-DCT is necessary for processing the video frames of the HEVC, as such the separable properties allow for a reduction in total computations to complete the transform when compared to the direct matrix multiplication. This comes at the cost of required data storage, as it is clear that the second step of the process (row/column 1D-DCT) can only begin once the first step is fully completed. The proposed architecture attempts to take advantage of these properties while reusing the existing architecture to compute both 1D-DCTs. To do this it is necessary to transpose the results of the first step, allowing it to be run through the same architecture while computing the opposite 1D-DCT to the first step.

5.2 Transpose Module

Transposition of the output of the first step 1D-DCT is crucial to the function of 2D-DCT architecture, as this module also needs to be able to store the outgoing data from multiple iterations of 1D-DCT. For this purpose two user defined types were created to represent a single column or row as well as the entire matrix. These are defined as:

```
type slv32 is array (0 to 31) of std_logic_vector(7 downto 0)
type reg32x32_t is array (0 to 31, 0 to 31) of std_logic_vector(7 downto 0)
```

As can be seen in the given listing above, the classes are both arrays of std_logic_vectors and are implemented for a size of 32 x 32 byte as it is the largest block size that is compatible with HEVC. The slv32 type will be used to mange the input and output data of the module, while the The reg32x32_t type is used to store and manipulate the data within the module as what is effectively a set of 2D shift registers, set out in the same pattern as the 4 input and output shown in Figure 5.1.

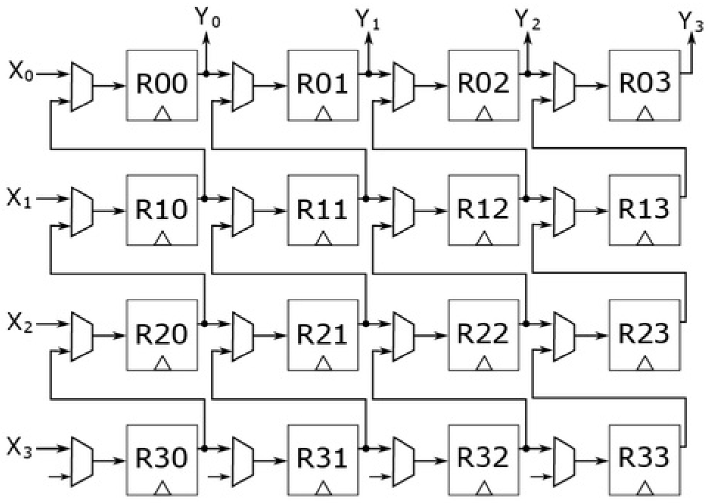


Figure 5.1: The proposed 2D shift register architecture, showing 4 inputs and 4 outputs. Data shifted in horizontal direction from left to right, and shifted out in up direction, all MUX selection changes accordingly

The module operates in two modes, 'shift in' and 'shift out'. During each clock cycle of the shift in mode, the transpose module receives data from the output of the first step 1D-DCT, the module then shifts the existing data in the registers to the right by one column to accommodate for the incoming data which is then stored within the left most column of the shift registers. This process continues until all of the first step transforms have been completed, where the module changes into 'shift out mode'. The shift out mode passes the stored values back into the input of the 1D-DCT in row order, starting from the top row of the shift registers. The values are shifted upwards through the registers each clock cycle until all rows have being passed back into the 1D-DCT. An implementation of this is included in Appendix B.1.

The size of this unit can be altered easily by altering the size of the input and output arrays and altering the for loops within the behavioral to correspond with the altered value. Using generics to create a generic model of the transposition along with a select signal would allow for general use of the module without the need for set size specification.

5.3 Two Dimensional DCT Architecture

The 2D-DCT architecture is implemented as a top-level module that encapsulates the 1D-DCT and the Transpose modules as well as a basic control module that changes the operating mode of the shift registers within the Transpose. The module uses the slv32 type for the input signal data_in and the output signal data_out, this allows for the input data to be easily assigned a row/column at a time which is then subsequently passed though a variable into the assigned signals of the 1D-DCT. The proposed architecture is shown in Figure 5.2. This design took inspiration from the 2D-DCT architecture proposed by Masera et al [3], where the transposition memory had been replaced by that described in the Transpose Module section above. This change allows for data to flow more freely between the 1D-DCT, the transposition memory and the output signals.

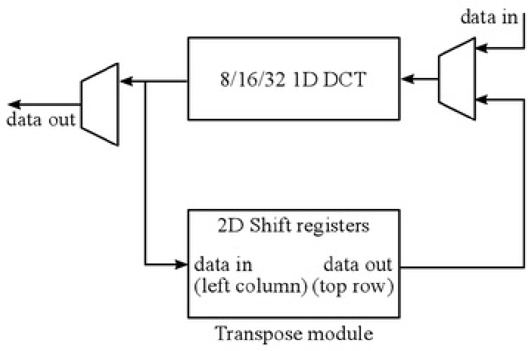


Figure 5.2: The proposed 2D-DCT architecture. Transposed module implemented using a 2-D register array, data enters in left column and shifted rightward, and the data shifted upward to get transposed data.

The proposed architecture requires 2N clock cycles to complete an N-Point 2D-DCT. For example when performing the 32-point 2D-DCT, the first 32 clock cycles are required to complete the column transforms and store them within the shift registers, it then requires another 32 clock cycles to shift the rows out of the registers to perform the row transformations for a total of 64 clock cycles.

Alternate designs were considered for this method, however this implementation was chosen due to its greater area efficiency with minute differences in computational speed when compared with other methods.

Chapter 6

Results and Comparison

6.1 Introduction

This section investigates the results obtained through simulation and testing of the overall 2D-DCT architecture design, as well as the lower level modules of the 1D-DCT architecture. These results are then compared with that of existing implementations that are currently associated with the HEVC.

6.2 Verification of DCT Designs

The proposed architecture, written in VHDL hardware description language, is verified through simulations in ModelSim that produce waveforms that are then compared to mathematical results produced by MATLAB to ensure that output given by the architecture correctly matches that expected of the DCT. The mathematical results produced by MATLAB were acquired by performing the matrix multiplication of the incoming data matrix (X) with the kernel matrices (T) to achieve the processed matrix (Y) using Equation 6.1. An expanded matrix form of this equation is shown displayed in Equation 6.2

$$X \bullet T = Y \tag{6.1}$$

$$Y = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \dots & x_{NN} \end{bmatrix} \bullet \begin{bmatrix} t_{11} & t_{12} & \dots & t_{1N} \\ t_{21} & t_{22} & \dots & t_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ t_{N1} & t_{N2} & \dots & t_{NN} \end{bmatrix} = \begin{bmatrix} y_{11} & y_{12} & \dots & y_{1N} \\ y_{21} & y_{22} & \dots & y_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ y_{N1} & y_{N2} & \dots & y_{NN} \end{bmatrix} \tag{6.2}$$

A simple example of this process, using the 4-Point DCT is outlined in Equation 6.3. This gives the expected result of the 2D-DCT and can be compared to the waveform produced by ModelSim to confirm the transform has completed successfully.

$$Y = \begin{bmatrix} 64 & 64 & 64 & 64 \\ 83 & 36 & -36 & -83 \\ 64 & -64 & -64 & 64 \\ 36 & -83 & 83 & -36 \end{bmatrix} \bullet \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} = \begin{bmatrix} 566 & -388 & 132 & -54 \\ 1554 & -576 & 320 & -18 \\ 2542 & -764 & 508 & 18 \\ 3530 & -952 & 696 & 54 \end{bmatrix} \tag{6.3}$$

These results were then able to be compared to the output data of the appropriate length DCT simulation to verify the transform. Additionally the 1D-DCT was also tested by inputting the equations defined in the respected algorithms of each length DCT into MATLAB and checking the results against them. This was used as an initial verification before the 2D-DCT architecture was completed. Each 1D-DCT module outlined Chapter 3.3 successfully completed the DCT for their respective lengths. The 2D-DCT was tested for the 32-point length only, however this ensures that all lower modules would also complete successfully as they are implemented within this module and are needed for the correct output value to occur.

6.3 Synthesis of 1D-DCT Architectures

The 1D-DCT modules were synthesized using Xilinx Design Suite to give basic information about produced architecture. These results are displayed in Table 6.1. The synthesis was preformed with the target device belonging to the Kintex7 family of FPGA devices which allowed for 410000 Slice LUTs and 300 IOBs to be assigned to the device.

Table 6.1: Synthesis Results of the 1D-DCT Modules, performed by Xilinx ISE

1D-DCT Synthesis Results			
Module	Slice LUTs	IOBs	Est. Delay
4-Point	57	64	3.282ns
8-Point	336	128	4.691ns
16-Point	1289	256	6.878ns
32-Point	4860	512	7.645ns

The synthesis results are all quite positive producing no errors and only one warning in the 16 and 32-point modules as it exceeded the number of IOBs available on the target device. This means that the modules would not be able to be implemented onto the device in question, however it would still be possible when using an FPGA device with an appropriate number of IOBs. It is also possible to make an Application Specific Integrated Circuit (ASIC) which can be purposefully made to suit the design, as this was not necessary to verify that the designs of the 16 and 32 point 1D-DCTs this was not done at this stage.

The estimated delay of each module is satisfactory with the most important module to examine being the 32 Point 1D-DCT. The delay in this device is crucial as it will affect the maximum clock speed of 2D-DCT, such that there is enough time for the transform to be completed before being passed to the shift registers for storage. Using this result it is estimated that the delay of completing the column-wise 32 point 1D-DCT will take approximately $32 \times 7.645ns = 244.64ns$ and will take approximately $32 \times 244.64ns = 7,828.48ns = 7.828\mu s$ to complete the 2D-DCT transformation of a single 32 x 32 video frame.

6.4 Synthesis of the 2D-DCT Architecture

6.4.1 Xilinx FPGA Synthesis

The 2D-DCT architecture was synthesized using the Xilinx Design Suite to obtain an overview of the designs parameters, these are displayed in Table 6.2. The synthesis was performed using the same target device of the Kintex7 family of FPGAs as used by the 1D-DCT.

Table 6.2: Synthesis Results of the 2D-DCT Architecture, using Xilinx ISE

	# Used	# Available
Slice Registers	8201	82000
Slice LUTs	12957	41000
IOBs	514	300

The synthesis results shows, as expected, an increase in slice LUTs when compared to the results of the 1D-DCT while also introducing the registers when compared to the 1D-DCT, this is due to storage in the 2D-DCT that is required to be able to complete the transpose and the second step transform. There was a small increase in the IOBs to accommodate for the extra logic within the controller and input data, this retains the same problems as the 1D-DCT architecture.

The timing of the 2D-DCT is more important due to the requirement of a clock to manage the shift registers correctly. The clock in this case needs to cover a minimum period that allows for the transform to complete and for the result to be stored in the

Table 6.3: Clock Timing Breakdown for the 2D-DCT, from the Xilinx Synthesis Report

Property	Value
Min. Period	7.199 ns
Max. Clock Freq	138.91 MHz
Min. input arrival time before clock	7.075 ns
Max. Output required time after clock	7.146 ns
Max. Combinational Path Delay	7.022 ns

shift registers. Using the information provided by the Synthesis Report, the minimum period is approximately 7.199ns which allows for a maximum clock frequency of 138.915 MHz, a more detailed breakdown of the timing of the 2D-DCT architecture is shown in Table 6.3. The maximum frequency is relatively high for an FGPA implementation and is satisfactory for the purpose of the project, however this could be improved by using an ASIC design in its place.

6.4.2 ASIC Synthesis

An ASIC design was synthesized using the VHDL designed modules using Synopsys Design Compiler version K-2015.06 with Synopsys Armenia Educational Department (SAED) design kit standard 28nm logic cell libraries, for operating conditions of 1.16V and a worst case temperature of $125°C$. The hardware requires clock cycles to complete a 32 x 32 block, i.e 16 pixels per clock cycle when processing a 32 point block. This can vary depending on the block size used when processing the video frames, the worst case of 2 pixels per cycle occurs when processing using a 4x4 block size, however this is unlikely due as the average block size for UHD content is 16 x 16. To be comparable to other existing models, the proposed architecture needs to be able to process 8k UHD @ 60 Hz in 4:2:0 YUV format which requires a minimum clock speed of 374 MHz. The synthesis results of the ASIC, shown in Table 6.4, estimates the design can operate with a maximum clock speed of 450 MHz, which is much higher than the required clock speed. This is also much higher then the Xilinx synthesis' maximum clock speed, making the ASIC the better option in terms of performance. The synthesized design has an area of $0.0985mm^2$ or a 68k standard 2-input NAND equivalent gate count.

Table 6.4: Comparison of 2D-DCT Architectures

Design	Technology	# of Gates	Max. Freq.	Throughput	Supporting Video Format
TCSVT'14 Arch.-1 [4]	90 nm	347 k	187 MHz	5.984 G	8K UHD @ 60 FPS
TCSVT'14 Arch.-2 [4]	90 nm	208 k	187 MHz	2.992 G	8K UHD @ 60 FPS
TCSVT'16 Arch.-1 [3]	90 nm	243 k	250 MHz	3.212 G	8K UHD @ 64 FPS
TCSVT'16 Arch.-2 [3]	90 nm	157 k	250 MHz	1.302 G	8K UHD @ 26 FPS
Proposed	32 nm	65 k	450 MHz	3.600 G	8K UHD @ 60 FPS

The two proposals discussed in [4] and [3] for 2D-DCT architectures based on the unfolded and folded 1D-DCT referred to as Arch. 1 and Arch. 2 respectively in Table 6.4. The comparison of the key properties of the existing and proposed architectures is displayed in Table 6.4, from this it is clear that the proposed method uses less than half the logic gates of other designs while also having a greater maximum clock speed. The throughput is higher or approximately equivalent to most other designs with the exception of the full parallel method (Arch. 1) described by Meher et al [4]. In general, the proposed architecture saves more than 66% of the gate counts with approximately equivalent throughput when compared to the other designs in the table.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The research and project discussed in this document focuses around the development of a variable length Two-Dimensional Discrete Cosine Transform (2D-DCT) that was to be used for compressing the video frames of the HEVC/H.265 codec. The outcome of this produced an ASIC design, using 32nm technology, that is capable of processing 8K UHD video content at 60 FPS at a similar throughput to that of existing designs with a reduced total gate number and therefore reduced area. The design is able to run at a maximum clock frequency of 450 MHz which allows for a throughput of 3.600 G, with a total area of $0.0985mm^2$ or 68K standard NAND equivalent gate count. The improvement in transposition memory, the use of a reusable architecture and the assumption of the use of larger block sizes allows for the removal of unnecessary complexities that are present in other designs accommodates for the lower total gate count.

7.2 Future Work

Future developments of this project would be to further improve the area reduction though optimization of the larger length modules. This optimization could allow for further reduction of area or an increase in throughput/computational speed.

Chapter 8

Abbreviations

1D-DCT	One Dimensional Discrete Cosine Transform
2D-DCT	Two Dimensional Discrete Cosine Transform
ASIC	Application Specific Integrated Circuit
CLK	Clock
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
FPS	Frames Per Second
HD	High Definition
IAU	Input Adder Unit
MUX	Multiplexer
OAU	Output Adder Unit
SAED	Synopsys Armenia Educational Department
SAU	Shift-Add Unit
VR	Virtual Reality

Appendix A

1D-DCT VHDL Modules and Related Packages

A.1 Four-Point DCT

A.1.1 FourPoint.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.FourPointFunctions.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity FourPoint is
  port ( X40: in std_logic_vector(7 downto 0);
         X41: in std_logic_vector(7 downto 0);
         X42: in std_logic_vector(7 downto 0);
         X43: in std_logic_vector(7 downto 0);
       Y40: out std_logic_vector(7 downto 0);
         Y41: out std_logic_vector(7 downto 0);
         Y42: out std_logic_vector(7 downto 0);
         Y43: out std_logic_vector(7 downto 0));



end FourPoint;

architecture Behavioral of FourPoint is
```

```vhdl
begin

  process(X40, X41, X42, X43)

     variable A0  : std_logic_vector (7 downto 0);
     variable   A1  : std_logic_vector (7 downto 0);
     variable   B0  : std_logic_vector (7 downto 0);
     variable   B1   : std_logic_vector (7 downto 0);
     variable   T083 : std_logic_vector (7 downto 0);
     variable   T036 : std_logic_vector (7 downto 0);
     variable   T183 : std_logic_vector (7 downto 0);
     variable   T136 : std_logic_vector (7 downto 0);
     variable   SAUP  : std_logic_vector (7 downto 0);

  begin

  -- IAU Phase

      A0 := PPartial(X40, X43);
      B0 := NPartial(X40, X43);
      A1 := PPartial(X41, X42);
      B1 := NPartial(X41, X42);

  -- SAU Phase

      A0 := A0 (1 downto 0) & "000000"; -- t0,64
      A1 := A1 (1 downto 0) & "000000"; -- t1,64

      SAUP := SAU_Partial(B0);
      T083 := SAU_83(B0, SAUP);
      T036 := SAU_36(B0);

      SAUP := SAU_Partial(B1);
      T183 := SAU_83(B1, SAUP);
      T136 := SAU_36(B1);



  -- OAU Phase/Output

      Y40 <= PPartial(A0,A1);
      Y41 <= PPartial(T083,T136);
      Y42 <= NPartial(A0,A1);
      Y43 <= NPartial(T036,T183);


  end process;
end Behavioral;
```

Code/FourPoint.vhd

A.1.2 Four Point Functions

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

package FourPointFunctions is

function SL6 (input : in std_logic_vector(7 downto 0)) return
   std_logic_vector;
function SAU_Partial (input : in std_logic_vector(7 downto 0)) return
   std_logic_vector;
function SAU_36 (input : in std_logic_vector(7 downto 0)) return
   std_logic_vector;
function SAU_83 (input, SAUP : in std_logic_vector(7 downto 0)) return
   std_logic_vector;
function PPartial (A,B : in std_logic_vector(7 downto 0)) return
   std_logic_vector;
function NPartial (A,B : in std_logic_vector(7 downto 0)) return
   std_logic_vector;


end FourPointFunctions;

package body FourPointFunctions is
----- SL6 (arithmatic)
  function SL6 (input : in std_logic_vector(7 downto 0)) return
   std_logic_vector is
    begin
    return input(1 downto 0) & "000000";
    --return temp;
  end SL6;

---- SAU SHIFT 3 add Shift 1
  function SAU_Partial (input : in std_logic_vector(7 downto 0)) return
   std_logic_vector is
    variable temp : std_logic_vector (7 downto 0);
    variable x : std_logic_vector (7 downto 0);
  begin
    temp := input;
    x    := temp(4 downto 0) & "000";
    x    := x + temp;
    x    := x(6 downto 0) & "0";
    return x;
  end SAU_Partial;

---- t36 output
    function SAU_36 (input : in std_logic_vector(7 downto 0)) return
   std_logic_vector is
    variable temp : std_logic_vector (7 downto 0);
  begin
```

```
    temp := input;
    temp := temp(6 downto 0) & "0";
    return temp;
  end SAU_36;

---- t83 output
  function SAU_83  (input, SAUP: in std_logic_vector(7 downto 0)) return
   std_logic_vector is
    variable temp : std_logic_vector(7 downto 0);
   variable x : std_logic_vector(7 downto 0);
   variable y : std_logic_vector(7 downto 0);

  begin
   temp := input;
   x   := SAUP(6 downto 0) & "0";
   y    := temp(1 downto 0) & "000000";
   x   := x + y + temp;
    return x;
  end SAU_83;

---- IAU/OAU Positive Partial
  function PPartial ( A, B : in std_logic_vector(7 downto 0)) return
   std_logic_vector is
    variable temp : std_logic_vector (7 downto 0);
  begin
    temp := A + B;
    return temp;
  end PPartial;

---- IAU/OAU Negative Partial
  function NPartial ( A, B : in std_logic_vector(7 downto 0)) return
   std_logic_vector is
    variable temp : std_logic_vector (7 downto 0);
  begin
    temp := A - B;
    return temp;
  end NPartial;


end FourPointFunctions;
```

Code/FourPointFunctions.vhd

A.2 Eight Point Module

A.2.1 EightPoint.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.EightPointFunctions.ALL;

entity EightPoint is
    Port ( X80, X81, X82, X83, X84, X85, X86, X87 : in  STD_LOGIC_vector(7
   downto 0);
           Y80, Y81, Y82, Y83, Y84, Y85, Y86, Y87 : out  STD_LOGIC_vector(7
    downto 0));

end EightPoint;


architecture Behavioral of EightPoint is
  signal A0, A1, A2, A3   : std_logic_vector (7 downto 0);


component FourPoint
  Port(
  X40, X41, X42, X43 : in std_logic_vector(7 downto 0);
  Y40, Y41, Y42, Y43 : out std_logic_vector(7 downto 0));
end component;

begin

  uut: FourPoint port map (
  X40 => A0,
  X41 => A1,
  X42 => A2,
  X43 => A3,
  Y40 => Y80,
  Y41 => Y82,
  Y42 => Y84,
  Y43 => Y86
  );

  process (X80, X81, X82, X83, X84, X85, X86, X87)
  variable B0, B1, B2, B3   : std_logic_vector (7 downto 0);
  variable t018, t118, t218, t318 : std_logic_vector(7 downto 0);
   variable t089, t189, t289, t389 : std_logic_vector(7 downto 0);
   variable t075, t175, t275, t375 : std_logic_vector(7 downto 0);
   variable t050, t150, t250, t350 : std_logic_vector(7 downto 0);

   begin
---IAU (Stage 1)
```

```
  A0 <= X80 + X87;
  B0 := X80 - X87;
  A1 <= X81 + X86;
  B1 := X81 - X86;
  A2 <= X82 + X85;
  B2 := X82 - X85;
  A3 <= X83 + X84;
  B3 := X83 - X84;

-- Stage 2 (SAU and FourPoint)
  --Itermdeites
  t018 := SAU8_A1(B0);
  t118 := SAU8_A1(B1);
  t218 := SAU8_A1(B2);
  t318 := SAU8_A1(B3);

-- Create 2nd intermedietes
  t050 := SAU8_A2(B0, t018);
  t150 := SAU8_A2(B1, t118);
  t250 := SAU8_A2(B2, t218);
  t350 := SAU8_A2(B3, t318);

-- Set t18s
  t018 := t018(6 downto 0) & '0';
  t118 := t118(6 downto 0) & '0';
  t218 := t218(6 downto 0) & '0';
  t318 := t318(6 downto 0) & '0';

-- Set t89s
  t089 := SAU8_t89(B0, t050);
  t189 := SAU8_t89(B1, t150);
  t289 := SAU8_t89(B2, t250);
  t389 := SAU8_t89(B3, t350);

-- Set t75s
  t075 := t050(6 downto 0) & '0';
  t075 := t075 + t050;
  t175 := t150(6 downto 0) & '0';
  t175 := t175 + t150;
  t275 := t250(6 downto 0) & '0';
  t275 := t275 + t250;
  t375 := t350(6 downto 0) & '0';
  t375 := t375 + t350;

-- Set t50s
  t050 := t050(6 downto 0) & '0';
  t150 := t150(6 downto 0) & '0';
  t250 := t250(6 downto 0) & '0';
  t350 := t350(6 downto 0) & '0';

--OAU
  Y87 <= (t018 - t389) + (t275 - t150);
  Y85 <= (t218 - t189) + (t375 - t050);
  Y83 <= (t118 - t289) + (t075 - t350);
  Y81 <= (t318 - t089) + (t175 - t250);



end process;
end Behavioral;
```

Code/EightPoint.vhd

A.2.2 EightPointFunctions.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

package EightPointFunctions is
function SAU8_A1 (input : in std_logic_vector(7 downto 0)) return
    std_logic_vector;
function SAU8_A2 (input, A1 : in std_logic_vector(7 downto 0)) return
    std_logic_vector;
function SAU8_t89 (input, A2 : in std_logic_vector(7 downto 0)) return
    std_logic_vector;

end EightPointFunctions;

package body EightPointFunctions is

-- First Stage Addition
  function SAU8_A1 ( input : in std_logic_vector(7 downto 0)) return
    std_logic_vector is
     variable temp : std_logic_vector (7 downto 0);
  begin
     temp := input(4 downto 0) & "000";
     temp := temp + input;
     return temp;
  end SAU8_A1;

-- Second Stage Addition
  function SAU8_A2 (input, A1 : in std_logic_vector(7 downto 0)) return
    std_logic_vector is
     variable temp : std_logic_vector (7 downto 0);
  begin
     temp := input(3 downto 0) & "0000";
     temp := temp + input + A1;
     return temp;
  end SAU8_A2;

-- t89 generator
  function SAU8_t89 (input, A2 : in std_logic_vector(7 downto 0)) return
    std_logic_vector is
     variable temp : std_logic_vector (7 downto 0);
  begin
     temp := input(1 downto 0) & "000000";
     temp := temp + input + A2;
     return temp;
  end SAU8_t89;
end EightPointFunctions;
```

Code/EightPointFunctions.vhd

A.3 SixteenPoint Module

A.3.1 SixteenPoint.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity SixteenPoint is

Port ( X160, X161, X162, X163, X164, X165, X166, X167, X168, X169, X1610,
    X1611, X1612, X1613, X1614, X1615 : in  std_logic_vector(7 downto 0);
        Y160, Y161, Y162, Y163, Y164, Y165, Y166, Y167, Y168, Y169, Y1610,
    Y1611, Y1612, Y1613, Y1614, Y1615 : out  std_logic_vector(7 downto 0));

end SixteenPoint;

architecture Behavioral of SixteenPoint is
  signal A0, A1, A2, A3, A4, A5, A6, A7 : std_logic_vector(7 downto 0);

  Component EightPoint
    port (X80, X81, X82, X83, X84, X85, X86, X87 : in  STD_LOGIC_vector(7
    downto 0);
         Y80, Y81, Y82, Y83, Y84, Y85, Y86, Y87 : out  STD_LOGIC_vector(7
    downto 0));
  end component;


begin
  uut: EightPoint port map (
  -- inputs
  X80 => A0,
  X81 => A1,
  X82 => A2,
  X83 => A3,
  X84 => A4,
  X85 => A5,
  X86 => A6,
  X87 => A7,
  --outputs
  Y80 => Y160,
  Y81 => Y162,
  Y82 => Y164,
  Y83 => Y166,
  Y84 => Y168,
  Y85 => Y1610,
  Y86 => Y1612,
  Y87 => Y1614
  );

```

```

process (X160, X161, X162, X163, X164, X165, X166, X167, X168, X169,
 X1610, X1611, X1612, X1613, X1614, X1615)
variable B0, B1, B2, B3, B4, B5, B6, B7   : std_logic_vector (7 downto 0)
 ;
variable m08, m18, m28, m38, m48, m58, m68, m78 : std_logic_vector (7
 downto 0);
variable t09, t19, t29, t39, t49, t59, t69, t79 : std_logic_vector (7
 downto 0);
variable t025, t125, t225, t325, t425, t525, t625, t725 :
 std_logic_vector (7 downto 0);
variable t043, t143, t243, t343, t443, t543, t643, t743 :
 std_logic_vector (7 downto 0);
variable t057, t157, t257, t357, t457, t557, t657, t757 :
 std_logic_vector (7 downto 0);
variable t070, t170, t270, t370, t470, t570, t670, t770 :
 std_logic_vector (7 downto 0);
variable t080, t180, t280, t380, t480, t580, t680, t780 :
 std_logic_vector (7 downto 0);
variable t087, t187, t287, t387, t487, t587, t687, t787 :
 std_logic_vector (7 downto 0);
variable t090, t190, t290, t390, t490, t590, t690, t790 :
 std_logic_vector (7 downto 0);
variable m018, m118, m218, m318, m418, m518, m618, m718 :
 std_logic_vector (7 downto 0);
variable m072, m172, m272, m372, m472, m572, m672, m772 :
 std_logic_vector (7 downto 0);

begin
--IAU
A0 <= X160 + X1615;
A1 <= X161 + X1614;
A2 <= X162 + X1613;
A3 <= X163 + X1612;
A4 <= X164 + X1611;
A5 <= X165 + X1610;
A6 <= X166 + X169;
A7 <= X167 + X168;

B0 := X160 - X1615;
B1 := X161 - X1614;
B2 := X162 - X1613;
B3 := X163 - X1612;
B4 := X164 - X1611;
B5 := X165 - X1610;
B6 := X166 - X169;
B7 := X167 - X168;


-- SAU (Stage 2)
-- m8's
m08 := B0(4 downto 0) & "000";
m18 := B0(4 downto 0) & "000";
m28 := B0(4 downto 0) & "000";
m38 := B0(4 downto 0) & "000";
m48 := B0(4 downto 0) & "000";
m58 := B0(4 downto 0) & "000";
m68 := B0(4 downto 0) & "000";
m78 := B0(4 downto 0) & "000";

-- t9's
t09 := m08 + B0;
t19 := m18 + B1;
t29 := m28 + B2;
t39 := m38 + B3;
t49 := m48 + B4;
t59 := m58 + B5;
t69 := m68 + B6;
t79 := m78 + B7;

-- m18's
m018 := t09(6 downto 0) & '0';
m118 := t19(6 downto 0) & '0';
m218 := t29(6 downto 0) & '0';
m318 := t39(6 downto 0) & '0';
m418 := t49(6 downto 0) & '0';
m518 := t59(6 downto 0) & '0';
m618 := t69(6 downto 0) & '0';
m718 := t79(6 downto 0) & '0';

-- m72's
m072 := t09(4 downto 0) & "000";
m172 := t19(4 downto 0) & "000";
m272 := t29(4 downto 0) & "000";
m372 := t39(4 downto 0) & "000";
m472 := t49(4 downto 0) & "000";
m572 := t59(4 downto 0) & "000";
m672 := t69(4 downto 0) & "000";
m772 := t79(4 downto 0) & "000";

-- t25's
t025 := B0(3 downto 0) & "0000";
t025 := t025 + t09;
t125 := B1(3 downto 0) & "0000";
t025 := t125 + t19;
t225 := B2(3 downto 0) & "0000";
t225 := t225 + t29;
t325 := B3(3 downto 0) & "0000";
t325 := t325 + t39;
t425 := B4(3 downto 0) & "0000";
t425 := t425 + t49;
t525 := B5(3 downto 0) & "0000";
```

```
    t525 := t525 + t59;
    t625 := B6(3 downto 0) & "0000";
    t625 := t625 + t69;
    t725 := B7(3 downto 0) & "0000";
    t725 := t725 + t79;

    -- t43's
    t043 := m018 + t025;
    t143 := m118 + t125;
    t243 := m218 + t225;
    t343 := m318 + t325;
    t443 := m418 + t425;
    t543 := m518 + t525;
    t643 := m618 + t625;
    t743 := m718 + t725;

    -- t57's
    t057 := B0(2 downto 0) & "00000";
    t057 := t057 + t025;
    t157 := B1(2 downto 0) & "00000";
    t157 := t157 + t125;
    t257 := B2(2 downto 0) & "00000";
    t257 := t257 + t225;
    t357 := B3(2 downto 0) & "00000";
    t357 := t357 + t325;
    t457 := B4(2 downto 0) & "00000";
    t457 := t457 + t425;
    t557 := B5(2 downto 0) & "00000";
    t557 := t557 + t525;
    t657 := B6(2 downto 0) & "00000";
    t657 := t657 + t625;
    t757 := B7(2 downto 0) & "00000";
    t757 := t757 + t725;

    -- t70s
    t070 := B0(6 downto 0) & '0';
    t170 := B1(6 downto 0) & '0';
    t270 := B2(6 downto 0) & '0';
    t370 := B3(6 downto 0) & '0';
    t470 := B4(6 downto 0) & '0';
    t570 := B5(6 downto 0) & '0';
    t670 := B6(6 downto 0) & '0';
    t770 := B7(6 downto 0) & '0';

    t070 := m072 - t070;
    t170 := m172 - t170;
    t270 := m272 - t270;
    t370 := m372 - t370;
    t470 := m472 - t470;
    t570 := m572 - t570;
    t670 := m672 - t670;
    t770 := m772 - t770;
```

```

-- t80s
t080 := m072 + m08;
t180 := m172 + m18;
t280 := m272 + m28;
t380 := m372 + m38;
t480 := m472 + m48;
t580 := m572 + m58;
t680 := m672 + m68;
t780 := m772 + m78;

-- t87s
t087 := t043(6 downto 0) & '0';
t187 := t143(6 downto 0) & '0';
t287 := t243(6 downto 0) & '0';
t387 := t343(6 downto 0) & '0';
t487 := t443(6 downto 0) & '0';
t587 := t543(6 downto 0) & '0';
t687 := t643(6 downto 0) & '0';
t787 := t743(6 downto 0) & '0';

t087 := t087 + B0;
t187 := t187 + B1;
t287 := t287 + B2;
t387 := t387 + B3;
t487 := t487 + B4;
t587 := t587 + B5;
t687 := t687 + B6;
t787 := t787 + B7;

-- t90s
t090 := m072 + m018;
t190 := m172 + m118;
t290 := m272 + m218;
t390 := m372 + m318;
t490 := m472 + m418;
t590 := m572 + m518;
t690 := m672 + m618;
t790 := m772 + m718;

--OAU
Y161 <= t090 + t187 + t280 + t370 + t457 + t543 + t625 + t79;
Y163 <= t087 + t157 + t29 - t343 - t480 - t590 - t670 - t725;
Y165 <= t080 + t19 - t270 - t387 - t425 + t557 + t690 + t743;
Y167 <= t070 - t143 - t287 + t39 + t490 + t525 - t680 - t757;
Y169 <= t057 - t180 - t225 + t390 - t49 - t587 + t643 + t770;
Y1611 <= t043 - t190 + t257 + t325 - t487 + t570 + t69 - t780;
Y1613 <= t025 - t170 + t290 - t380 + t443 + t59 - t657 + t787;
Y1615 <= t09 - t125 + t243 - t357 + t470 - t580 + t687 - t790;


end process;
end Behavioral;
```

Code/SixteenPoint.vhd

A.4 32-Point Module

A.4.1 ThirtyTwoPoint.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

use work.ThirtyTwoFunctions.ALL;

entity ThirtyTwoPoint is
  port ( X320, X321, X322, X323, X324, X325, X326, X327,
     X328, X329, X3210, X3211, X3212, X3213, X3214, X3215,
     X3216, X3217, X3218, X3219, X3220, X3221, X3222, X3223,
     X3224, X3225, X3226, X3227, X3228, X3229, X3230, X3231 : in
    std_logic_vector(7 downto 0);

     Y320,  Y321,  Y322,  Y323,  Y324,  Y325,  Y326,  Y327,
     Y328,  Y329,  Y3210, Y3211, Y3212, Y3213, Y3214, Y3215,
     Y3216, Y3217, Y3218, Y3219, Y3220, Y3221, Y3222, Y3223,
     Y3224, Y3225, Y3226, Y3227, Y3228, Y3229, Y3230, Y3231 : out
    std_logic_vector(7 downto 0));
end ThirtyTwoPoint;

architecture Behavioral of ThirtyTwoPoint is



    signal A0, A1, A2, A3, A4, A5, A6, A7,
         A8, A9, A10, A11, A12, A13, A14, A15  : std_logic_vector(7 downto
    0);

    component SixteenPoint
      port (X160, X161, X162, X163, X164, X165, X166, X167,
          X168, X169, X1610, X1611, X1612, X1613, X1614, X1615 : in
    std_logic_vector(7 downto 0);

          Y160, Y161, Y162, Y163, Y164, Y165, Y166, Y167,
          Y168, Y169, Y1610, Y1611, Y1612, Y1613, Y1614, Y1615 : out
    std_logic_vector(7 downto 0));
    end component;


begin
  uut: SixteenPoint port map (
    X160 => A0,
    X161 => A1,
    X162 => A2,
    X163 => A3,
    X164 => A4,
    X165 => A5,
    X166 => A6,
    X167 => A7,
    X168 => A8,
    X169 => A9,
    X1610 => A10,
    X1611 => A11,
    X1612 => A12,
    X1613 => A13,
    X1614 => A14,
    X1615 => A15,

    Y160 => Y320,
    Y161 => Y322,
    Y162 => Y324,
    Y163 => Y326,
    Y164 => Y328,
    Y165 => Y3210,
    Y166 => Y3212,
    Y167 => Y3214,
    Y168 => Y3216,
    Y169 => Y3218,
    Y1610 => Y3220,
    Y1611 => Y3222,
    Y1612 => Y3224,
    Y1613 => Y3226,
    Y1614 => Y3228,
    Y1615 => Y3230
    );

    PROCESS (X320, X321, X322, X323, X324, X325, X326, X327,
     X328, X329, X3210, X3211, X3212, X3213, X3214, X3215,
     X3216, X3217, X3218, X3219, X3220, X3221, X3222, X3223,
     X3224, X3225, X3226, X3227, X3228, X3229, X3230, X3231)

     variable B0, B1, B2, B3, B4, B5, B6, B7,
           B8, B9, B10, B11, B12, B13, B14, B15 : std_logic_vector(7 downto
     0);

     variable m02, m12, m22, m32, m42, m52, m62, m72,
           m82, m92, m102, m112, m122, m132, m142, m1502 : std_logic_vector
    (7 downto 0);

     variable t04, t14, t24, t34, t44, t54, t64, t74,
           t84, t94, t104, t114, t124, t134, t144, t1504 : std_logic_vector
    (7 downto 0);

     variable t09, t19, t29, t39, t49, t59, t69, t79,
           t89, t99, t109, t119, t129, t139, t149, t159 : std_logic_vector
    (7 downto 0);

     variable m018, m118, m218, m318, m418, m518, m618, m718,
        m818, m918, m1018, m1118, m1218, m1318, m1418, m1518 :
std_logic_vector(7 downto 0);

  variable m072, m172, m272, m372, m472, m572, m672, m772,
        m872, m972, m1072, m1172, m1272, m1372, m1472, m1572 :
std_logic_vector(7 downto 0);

 variable m052, m152, m252, m352, m452, m552, m652, m752,
        m852, m952, m1052, m1152, m1252, m1352, m1452, m1552 :
std_logic_vector(7 downto 0);

  variable t013, t113, t213, t313, t413, t513, t613, t713,
        t813, t913, t1013, t1113, t1213, t1313, t1413, t1513 :
std_logic_vector(7 downto 0);

 variable t022, t122, t222, t322, t422, t522, t622, t722,
         t822, t922, t1022, t1122, t1222, t1322, t1422, t1522 :
std_logic_vector(7 downto 0);

 variable t031, t131, t231, t331, t431, t531, t631, t731,
         t831, t931, t1031, t1131, t1231, t1331, t1431, t1531 :
std_logic_vector(7 downto 0);


 variable t038, t138, t238, t338, t438, t538, t638, t738,
         t838, t938, t1038, t1138, t1238, t1338, t1438, t1538 :
std_logic_vector(7 downto 0);


 variable  t046, t146, t246, t346, t446, t546, t646, t746,
         t846, t946, t1046, t1146, t1246, t1346, t1446, t1546 :
std_logic_vector(7 downto 0);

 variable  t054, t154, t254, t354, t454, t554, t654, t754,
         t854, t954, t1054, t1154, t1254, t1354, t1454, t1554 :
std_logic_vector(7 downto 0);

 variable  t061, t161, t261, t361, t461, t561, t661, t761,
         t861, t961, t1061, t1161, t1261, t1361, t1461, t1561 :
std_logic_vector(7 downto 0);

 variable  t067, t167, t267, t367, t467, t567, t667, t767,
         t867, t967, t1067, t1167, t1267, t1367, t1467, t1567 :
std_logic_vector(7 downto 0);

 variable  t073, t173, t273, t373, t473, t573, t673, t773,
         t873, t973, t1073, t1173, t1273, t1373, t1473, t1573 :
std_logic_vector(7 downto 0);

 variable  t078, t178, t278, t378, t478, t578, t678, t778,
         t878, t978, t1078, t1178, t1278, t1378, t1478, t1578 :
std_logic_vector(7 downto 0);
```

```

    variable  t082, t182, t282, t382, t482, t582, t682, t782,
            t882, t982, t1082, t1182, t1282, t1382, t1482, t1582 :
   std_logic_vector(7 downto 0);

    variable  t085, t185, t285, t385, t485, t585, t685, t785,
            t885, t985, t1085, t1185, t1285, t1385, t1485, t1585 :
   std_logic_vector(7 downto 0);

    variable  t088, t188, t288, t388, t488, t588, t688, t788,
            t888, t988, t1088, t1188, t1288, t1388, t1488, t1588 :
   std_logic_vector(7 downto 0);

    variable  t090, t190, t290, t390, t490, t590, t690, t790,
            t890, t990, t1090, t1190, t1290, t1390, t1490, t1590 :
   std_logic_vector(7 downto 0);



    begin

     --IAU
    A0 <= X320 + X3231;
    A1 <= X321 + X3230;
    A2 <= X322 + X3229;
    A3 <= X323 + X3228;
    A4 <= X324 + X3227;
    A5 <= X325 + X3226;
    A6 <= X326 + X3225;
    A7 <= X327 + X3224;
    A8 <= X328 + X3223;
    A9 <= X329 + X3222;
    A10 <= X3210 + X3221;
    A11 <= X3211 + X3220;
    A12 <= X3212 + X3219;
    A13 <= X3213 + X3218;
    A14 <= X3214 + X3217;
    A15 <= X3215 + X3216;

    B0 := X320 - X3231;
    B1 := X321 - X3230;
    B2 := X322 - X3229;
    B3 := X323 - X3228;
    B4 := X324 - X3227;
    B5 := X325 - X3226;
    B6 := X326 - X3225;
    B7 := X327 - X3224;
    B8 := X328 - X3223;
    B9 := X329 - X3222;
    B10 := X3210 - X3221;
    B11 := X3211 - X3220;
    B12 := X3212 - X3219;
    B13 := X3213 - X3218;
    B14 := X3214 - X3217;
    B15 := X3215 - X3216;

    -- SAU

    m02 := shift1(B0);
    m12 := shift1(B1);
    m22 := shift1(B2);
    m32 := shift1(B3);
    m42 := shift1(B4);
    m52 := shift1(B5);
    m62 := shift1(B6);
    m72 := shift1(B7);
    m82 := shift1(B8);
    m92 := shift1(B9);
    m102 := shift1(B10);
    m112 := shift1(B11);
    m122 := shift1(B12);
    m132 := shift1(B13);
    m142 := shift1(B14);
    m1502 := shift1(B15);

    t04 := shift2(B0);
    t14 := shift2(B1);
    t24 := shift2(B2);
    t34 := shift2(B3);
    t44 := shift2(B4);
    t54 := shift2(B5);
    t64 := shift2(B6);
    t74 := shift2(B7);
    t84 := shift2(B8);
    t94 := shift2(B9);
    t104 := shift2(B10);
    t114 := shift2(B11);
    t124 := shift2(B12);
    t134 := shift2(B13);
    t144 := shift2(B14);
    t1504 := shift2(B15);

    t09 := ti9(B0);
    t19 := ti9(B1);
    t29 := ti9(B2);
    t39 := ti9(B3);
    t49 := ti9(B4);
    t59 := ti9(B5);
    t69 := ti9(B6);
    t79 := ti9(B7);
    t89 := ti9(B8);
    t99 := ti9(B9);
    t109 := ti9(B10);
    t119 := ti9(B11);
    t129 := ti9(B12);
    t139 := ti9(B13);
    t149 := ti9(B14);
    t159 := ti9(B15);

    m018 := shift1(t09);
    m118 := shift1(t19);
    m218 := shift1(t29);
    m318 := shift1(t39);
    m418 := shift1(t49);
    m518 := shift1(t59);
    m618 := shift1(t69);
    m718 := shift1(t79);
    m818 := shift1(t89);
    m918 := shift1(t99);
    m1018 := shift1(t109);
    m1118 := shift1(t119);
    m1218 := shift1(t129);
    m1318 := shift1(t139);
    m1418 := shift1(t149);
    m1518 := shift1(t159);

    m072 := shift3(t09);
    m172 := shift3(t19);
    m272 := shift3(t29);
    m372 := shift3(t39);
    m472 := shift3(t49);
    m572 := shift3(t59);
    m672 := shift3(t69);
    m772 := shift3(t79);
    m872 := shift3(t89);
    m972 := shift3(t99);
    m1072 := shift3(t109);
    m1172 := shift3(t119);
    m1272 := shift3(t129);
    m1372 := shift3(t139);
    m1472 := shift3(t149);
    m1572 := shift3(t159);

    t013 := t04 + t09;
    t113 := t14 + t19;
    t213 := t24 + t29;
    t313 := t34 + t39;
    t413 := t44 + t49;
    t513 := t54 + t59;
    t613 := t64 + t69;
    t713 := t74 + t79;
    t813 := t84 + t89;
    t913 := t94 + t99;
    t1013 := t104 + t109;
    t1113 := t114 + t119;
    t1213 := t124 + t129;
    t1313 := t134 + t139;
    t1413 := t144 + t149;
    t1513 := t1504 + t159;

    m052 := shift2(t013);
    m152 := shift2(t113);
    m252 := shift2(t213);
    m352 := shift2(t313);
    m452 := shift2(t413);
    m552 := shift2(t513);
    m652 := shift2(t613);
    m752 := shift2(t713);
    m852 := shift2(t813);
    m952 := shift2(t913);
    m1052 := shift2(t1013);
    m1152 := shift2(t1113);
    m1252 := shift2(t1213);
    m1352 := shift2(t1313);
    m1452 := shift2(t1413);
    m1552 := shift2(t1513);

    t022 := t04 + m018;
    t122 := t14 + m118;
    t222 := t24 + m218;
    t322 := t34 + m318;
    t422 := t44 + m418;
    t522 := t54 + m518;
    t622 := t64 + m618;
    t722 := t74 + m718;
    t822 := t84 + m818;
    t922 := t94 + m918;
    t1022 := t104 + m1018;
    t1122 := t114 + m1118;
    t1222 := t124 + m1218;
    t1322 := t134 + m1318;
    t1422 := t144 + m1418;
    t1522 := t1504 + m1518;

    t031 := ti31(B0);
    t131 := ti31(B1);
    t231 := ti31(B2);
    t331 := ti31(B3);
    t431 := ti31(B4);
    t531 := ti31(B5);
    t631 := ti31(B6);
    t731 := ti31(B7);
    t831 := ti31(B8);
    t931 := ti31(B9);
    t1031 := ti31(B10);
    t1131 := ti31(B11);
    t1231 := ti31(B12);
    t1331 := ti31(B13);
    t1431 := ti31(B14);
    t1531 := ti31(B15);

    t038 := shift2(t09) + m02;
    t138 := shift2(t19) + m12;
    t238 := shift2(t29) + m22;
    t338 := shift2(t39) + m32;
    t438 := shift2(t49) + m42;
    t538 := shift2(t59) + m52;
    t638 := shift2(t69) + m62;
    t738 := shift2(t79) + m72;
    t838 := shift2(t89) + m82;
    t938 := shift2(t99) + m92;
    t1038 := shift2(t109) + m102;
    t1138 := shift2(t119) + m112;
    t1238 := shift2(t129) + m122;
    t1338 := shift2(t139) + m132;
    t1438 := shift2(t149) + m142;
    t1538 := shift2(t159) + m1502;

    t046 := shift1(t022) + m02;
    t146 := shift1(t122) + m12;
    t246 := shift1(t222) + m22;
    t346 := shift1(t322) + m32;
    t446 := shift1(t422) + m42;
    t546 := shift1(t522) + m52;
    t646 := shift1(t622) + m62;
    t746 := shift1(t722) + m72;
    t846 := shift1(t822) + m82;
    t946 := shift1(t922) + m92;
    t1046 := shift1(t1022) + m102;
    t1146 := shift1(t1122) + m112;
    t1246 := shift1(t1222) + m122;
    t1346 := shift1(t1322) + m132;
    t1446 := shift1(t1422) + m142;
    t1546 := shift1(t1522) + m1502;


    t054 := m052 + m02;
    t154 := m152 + m12;
    t254 := m252 + m22;
    t354 := m352 + m32;
    t454 := m452 + m42;
    t554 := m552 + m52;
    t654 := m652 + m62;
    t754 := m752 + m72;
    t854 := m852 + m82;
    t954 := m952 + m92;
    t1054 := m1052 + m102;
    t1154 := m1152 + m112;
    t1254 := m1252 + m122;
    t1354 := m1352 + m132;
    t1454 := m1452 + m142;
    t1554 := m1552 + m1502;


    t061 := shift1(t031) + B0;
    t161 := shift1(t131) + B1;
    t261 := shift1(t231) + B2;
    t361 := shift1(t331) + B3;
    t461 := shift1(t431) + B4;
    t561 := shift1(t531) + B5;
    t661 := shift1(t631) + B6;
    t761 := shift1(t731) + B7;
    t861 := shift1(t831) + B8;
    t961 := shift1(t931) + B9;
    t1061 := shift1(t1031) + B10;
    t1161 := shift1(t1131) + B11;
    t1261 := shift1(t1231) + B12;
    t1361 := shift1(t1331) + B13;
    t1461 := shift1(t1431) + B14;
    t1561 := shift1(t1531) + B15;

    t067 := t054 + t013;
    t167 := t154 + t113;
    t267 := t254 + t213;
    t367 := t354 + t313;
    t467 := t454 + t413;
    t567 := t554 + t513;
    t667 := t654 + t613;
    t767 := t754 + t713;
    t867 := t854 + t813;
    t967 := t954 + t913;
    t1067 := t1054 + t1013;
    t1167 := t1154 + t1113;
    t1267 := t1254 + t1213;
    t1367 := t1354 + t1313;
    t1467 := t1454 + t1413;
    t1567 := t1554 + t1513;

    t073 := m072 + B0;
    t173 := m172 + B1;
    t273 := m272 + B2;
    t373 := m372 + B3;
    t473 := m472 + B4;
    t573 := m572 + B5;
    t673 := m672 + B6;
    t773 := m772 + B7;
    t873 := m872 + B8;
    t973 := m972 + B9;
    t1073 := m1072 + B10;
    t1173 := m1172 + B11;
    t1273 := m1272 + B12;
    t1373 := m1372 + B13;
    t1473 := m1472 + B14;
    t1573 := m1572 + B15;

    t078 := shift1(t013) + m052;
    t178 := shift1(t113) + m152;
    t278 := shift1(t213) + m252;
    t378 := shift1(t313) + m352;
    t478 := shift1(t413) + m452;
    t578 := shift1(t513) + m552;
    t678 := shift1(t613) + m652;
    t778 := shift1(t713) + m752;
    t878 := shift1(t813) + m852;
    t978 := shift1(t913) + m952;
    t1078 := shift1(t1013) + m1052;
    t1178 := shift1(t1113) + m1152;
    t1278 := shift1(t1213) + m1252;
    t1378 := shift1(t1313) + m1352;
    t1478 := shift1(t1413) + m1452;
    t1578 := shift1(t1513) + m1552;

    t082 := t04 + t078;
    t182 := t14 + t178;
    t282 := t24 + t278;
    t382 := t34 + t378;
    t482 := t44 + t478;
    t582 := t54 + t578;
    t682 := t64 + t678;
    t782 := t74 + t778;
    t882 := t84 + t878;
    t982 := t94 + t978;
    t1082 := t104 + t1078;
    t1182 := t114 + t1178;
    t1282 := t124 + t1278;
    t1382 := t134 + t1378;
    t1482 := t144 + t1478;
    t1582 := t1504 + t1578;

    t085 := t013 + m018;
    t185 := t113 + m118;
    t285 := t213 + m218;
    t385 := t313 + m318;
    t485 := t413 + m418;
    t585 := t513 + m518;
    t685 := t613 + m618;
    t785 := t713 + m718;
    t885 := t813 + m818;
    t985 := t913 + m918;
    t1085 := t1013 + m1018;
    t1185 := t1113 + m1118;
    t1285 := t1213 + m1218;
    t1385 := t1313 + m1318;
    t1485 := t1413 + m1418;
    t1585 := t1513 + m1518;


    t088 := shift2(t022);
    t188 := shift2(t122);
    t288 := shift2(t222);
    t388 := shift2(t322);
    t488 := shift2(t422);
    t588 := shift2(t522);
    t688 := shift2(t622);
    t788 := shift2(t722);
    t888 := shift2(t822);
    t988 := shift2(t922);
    t1088 := shift2(t1022);
    t1188 := shift2(t1122);
    t1288 := shift2(t1222);
    t1388 := shift2(t1322);
    t1488 := shift2(t1422);
    t1588 := shift2(t1522);

    t090 := m018 + m072;
    t190 := m118 + m172;
    t290 := m218 + m272;
    t390 := m318 + m372;
    t490 := m418 + m472;
    t590 := m518 + m572;
    t690 := m618 + m672;
    t790 := m718 + m772;
    t890 := m818 + m872;
    t990 := m918 + m972;
    t1090 := m1018 + m1072;
    t1190 := m1118 + m1172;
    t1290 := m1218 + m1272;
    t1390 := m1318 + m1372;
    t1490 := m1418 + m1472;
    t1590 := m1518 + m1572;

    -- OAU
    Y321 <= t090  + t190 + t288 + t385 + t482 + t578 + t673 + t767 + t861 +
     t954 + t1046 + t1138 + t1231 + t1322 + t1413 + t1504;
    Y323 <= t090  + t182 + t267 + t346 + t422 - t54  - t631 - t754 - t873 -
     t985 - t1090 - t1188 - t1278 - t1361 - t1438 - t1513;
    Y325 <= t088  + t167 + t231 - t313 - t454 - t582 - t690 - t778 - t846 -
     t94  + t1038 + t1173 + t1290 + t1385 + t1461 + t1522;
    Y327 <= t085  + t146 - t213 - t367 - t490 - t573 - t622 + t738 + t882 +
     t988 + t1054 - t114  - t1261 - t1390 - t1478 - t1531;
    Y329 <= t082  + t122 - t254 - t390 - t461 + t513 + t678 + t785 + t831 -
     t946 - t1090 - t1167 + t124  + t1373 + t1488 + t1538;
    Y3211 <= t078 - t14  - t282 - t373 + t413 + t585 + t667 - t722 - t888 -
     t961 + t1031 + t1190 + t1254 - t1338 - t1490 - t1546;
    Y3213 <= t073 - t131 - t290 - t322 + t478 + t567 - t638 - t790 - t813 +
     t982 + t1061 - t1146 - t1288 - t134  + t1485 + t1554;
    Y3215 <= t067 - t154 - t278 + t338 + t485 - t522 - t690 + t74  + t890 +
     t913 - t1088 - t1131 + t1282 + t1346 - t1473 - t1561;
    Y3217 <= t061 - t173 - t246 + t382 + t431 - t588 - t613 + t790 - t84  -
     t990 + t1022 + t1185 - t1238 - t1378 + t1454 + t1567;
    Y3219 <= t054 - t185 - t24  + t388 - t446 - t561 + t682 + t713 - t890 +
     t938 + t1067 - t1178 - t1222 + t1390 - t1431 - t1573;
    Y3221 <= t046 - t190 + t238 + t354 - t490 + t531 + t661 - t788 + t822 +
     t967 - t1085 + t1113 + t1273 - t1382 + t144  + t1578;
    Y3223 <= t038 - t188 + t273 - t34  - t467 + t590 - t646 - t731 + t885 -
     t978 + t1013 + t1161 - t1290 + t1354 + t1422 - t1582;
    Y3225 <= t031 - t178 + t290 - t361 + t44  + t554 - t688 + t782 - t838 -
     t922 + t1073 - t1190 + t1267 - t1313 - t1446 + t1585;
    Y3227 <= t022 - t161 + t285 - t390 + t473 - t538 - t64  + t746 - t878 +
     t990 - t1082 + t1154 - t1213 - t1331 + t1467 - t1588;
    Y3229 <= t013 - t138 + t261 - t378 + t488 - t590 + t685 - t773 + t854 -
     t931 + t104  + t1122 - t1246 + t1367 - t1482 + t1590;
    Y3231 <= t04  - t113 + t222 - t331 + t438 - t546 + t654 - t761 + t867 -
     t973 + t1078 - t1182 + t1285 - t1388 + t1490 - t1590;

    END PROCESS;


end Behavioral;
```

Code/ThirtyTwoPoint.vhd

A.4.2 ThirtyTwoFunctions.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

package ThirtyTwoFunctions is
  function shift1 ( input : std_logic_vector (7 downto 0)) return
   std_logic_vector;
  function shift2 ( input : std_logic_vector (7 downto 0)) return
   std_logic_vector;
  function shift3 ( input : std_logic_vector (7 downto 0)) return
   std_logic_vector;
  function ti9    ( input : std_logic_vector (7 downto 0)) return
   std_logic_vector;
  function ti31   ( input : std_logic_vector (7 downto 0)) return
   std_logic_vector;

end ThirtyTwoFunctions;

package body ThirtyTwoFunctions is

---- mi2
 function shift1  ( input : std_logic_vector(7 downto 0)) return
   std_logic_vector is
    variable var   : std_logic_vector(7 downto 0);
  begin
    var := input(6 downto 0) & '0';
    return var;
  end shift1;


  function shift2  ( input : std_logic_vector(7 downto 0)) return
   std_logic_vector is
    variable  var   : std_logic_vector(7 downto 0);
  begin
    var := input(5 downto 0) & "00";
    return var;
  end shift2;

  function shift3  ( input : std_logic_vector(7 downto 0)) return
   std_logic_vector is
    variable var   : std_logic_vector(7 downto 0);
  begin
    var := input(4 downto 0) & "000";
    return var;
  end shift3;

   function ti9  ( input : std_logic_vector(7 downto 0)) return
   std_logic_vector is
    variable var   : std_logic_vector(7 downto 0);
  begin
    var := input(4 downto 0) & "000";
   var := var + input;
    return var;
  end ti9;


  function ti31  ( input : std_logic_vector(7 downto 0)) return
   std_logic_vector is
    variable var   : std_logic_vector(7 downto 0);
  begin
    var := input(2 downto 0) & "00000";
   var := var - input;
    return var;
  end ti31;

end ThirtyTwoFunctions;
```

Code/ThirtyTwoFunctions.vhd

Appendix B

2D-DCT VHDL Modules and Related Packages

B.1 Shift Register/ Transpose Module

```
library ieee;
use ieee.std_logic_1164.all;
use work.data_type.all;

entity shift_reg is

  port (
    data_in  : in  slv32;
    data_out : out slv32;
    shift_in : in std_logic;
    clk      : in  std_logic);

end entity shift_reg;

architecture behavioural of shift_reg is
  type reg32x32_t is array (0 to 31, 0 to 31) of std_logic_vector(7 downto
    0);
  signal reg32x32 : reg32x32_t;
begin  -- architecture behavioural

  process (clk) is
  begin  -- process
    if (clk'event and clk = '1') then  -- rising clock edge
      if (shift_in = '1') then
        for i in 0 to 30 loop
          for j in 0 to 31 loop
            reg32x32(i+1, j) <= reg32x32(i, j); -- shift horizontal (right)
     dir
          end loop;  -- j
        end loop;  -- i
        for i in 0 to 31 loop
```

```
          reg32x32(0, i) <= data_in(i); -- data in to the left
        end loop;  -- i
      else
        for j in 0 to 30 loop
          for i in 0 to 31 loop
            reg32x32(i, j) <= reg32x32(i, j+1); -- shift out in up
   direction
          end loop;  -- i
        end loop;  -- j
      end if;
    end if;
    for i in 0 to 31 loop
      data_out(i) <= reg32x32(i, 0); -- data out from the top
    end loop;  -- i
  end process;

end architecture behavioural;
```

Code/shift_register.vhd

B.2 SR Control Module

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity ctrl is

  port (
    shift_ctrl : out std_logic;
    clk        : in  std_logic;
    reset      : in  std_logic);

end entity ctrl;

architecture behavioural of ctrl is
  signal count : integer range 0 to 63;

begin  -- architecture behavioural

  process (clk, reset) is
  begin  -- process
    if (reset = '1') then -- asynchronous reset (active high)
      count <= 0;
      shift_ctrl <= '1';
    elsif (clk'event and clk = '1') then  -- rising clock edge
      if (count = 63) then
        count <= 0;
        shift_ctrl <= '1'; -- Back to shift in mode, start again
      else
        count <= count + 1;
      end if;

      if (count = 31) then
        shift_ctrl <= '0'; -- Change shift reg to shift out mode
      end if;
    end if;
  end process;

end architecture behavioural;
```

Code/ctrl.vhd

B.3 Top Level Module

```
library ieee;
use ieee.std_logic_1164.all;
use work.data_type.all;

entity top is

  port (
    data_in  : in  slv32;
    data_out : out slv32;
    clk      : in  std_logic;
    reset    : in  std_logic);

end entity top;

architecture behavioural of top is

  component ThirtyTwoPoint
    port (X320, X321, X322, X323, X324, X325, X326, X327,
          X328, X329, X3210, X3211, X3212, X3213, X3214, X3215,
          X3216, X3217, X3218, X3219, X3220, X3221, X3222, X3223,
          X3224, X3225, X3226, X3227, X3228, X3229, X3230, X3231 : in
   std_logic_vector(7 downto 0);

          Y320, Y321, Y322, Y323, Y324, Y325, Y326, Y327,
          Y328, Y329, Y3210, Y3211, Y3212, Y3213, Y3214, Y3215,
          Y3216, Y3217, Y3218, Y3219, Y3220, Y3221, Y3222, Y3223,
          Y3224, Y3225, Y3226, Y3227, Y3228, Y3229, Y3230, Y3231 : out
   std_logic_vector(7 downto 0));
  end component ThirtyTwoPoint;


  component shift_reg
    port (
      data_in  : in  slv32;
      data_out : out slv32;
      shift_in : in  std_logic;
      clk      : in  std_logic);
  end component shift_reg;

  component ctrl
    port (
      shift_ctrl : out std_logic;
      clk        : in  std_logic;
      reset      : in  std_logic);
  end component ctrl;

  signal di, do, shift_reg_di, shift_reg_do : slv32;
  signal shift_in                           : std_logic;

```

```
begin  -- architecture behavioural

  i_dct_32pt : ThirtyTwoPoint port map (
    X320  => di(0),
    X321  => di(1),
    X322  => di(2),
    X323  => di(3),
    X324  => di(4),
    X325  => di(5),
    X326  => di(6),
    X327  => di(7),
    X328  => di(8),
    X329  => di(9),
    X3210 => di(10),
    X3211 => di(11),
    X3212 => di(12),
    X3213 => di(13),
    X3214 => di(14),
    X3215 => di(15),
    X3216 => di(16),
    X3217 => di(17),
    X3218 => di(18),
    X3219 => di(19),
    X3220 => di(20),
    X3221 => di(21),
    X3222 => di(22),
    X3223 => di(23),
    X3224 => di(24),
    X3225 => di(25),
    X3226 => di(26),
    X3227 => di(27),
    X3228 => di(28),
    X3229 => di(29),
    X3230 => di(30),
    X3231 => di(31),
    Y320  => do(0),
    Y321  => do(1),
    Y322  => do(2),
    Y323  => do(3),
    Y324  => do(4),
    Y325  => do(5),
    Y326  => do(6),
    Y327  => do(7),
    Y328  => do(8),
    Y329  => do(9),
    Y3210 => do(10),
    Y3211 => do(11),
    Y3212 => do(12),
    Y3213 => do(13),
    Y3214 => do(14),
    Y3215 => do(15),
    Y3216 => do(16),
    Y3217 => do(17),
    Y3218 => do(18),
    Y3219 => do(19),
    Y3220 => do(20),
    Y3221 => do(21),
    Y3222 => do(22),
    Y3223 => do(23),
    Y3224 => do(24),
    Y3225 => do(25),
    Y3226 => do(26),
    Y3227 => do(27),
    Y3228 => do(28),
    Y3229 => do(29),
    Y3230 => do(30),
    Y3231 => do(31));

  i_ctrl : ctrl port map (
    shift_ctrl => shift_in,
    clk        => clk,
    reset      => reset);

  i_shift_reg : shift_reg port map (
    data_in  => shift_reg_di,
    data_out => shift_reg_do,
    shift_in => shift_in,
    clk      => clk);

  --with shift_in select
  --  do <=
  --  data_out     when '0',
  --  shift_reg_di when others;
  data_out <= do;
  shift_reg_di <= do;

  with shift_in select
    di <=
    shift_reg_do when '0',
    data_in      when others;

end architecture behavioural;
```

Code/top.vhd

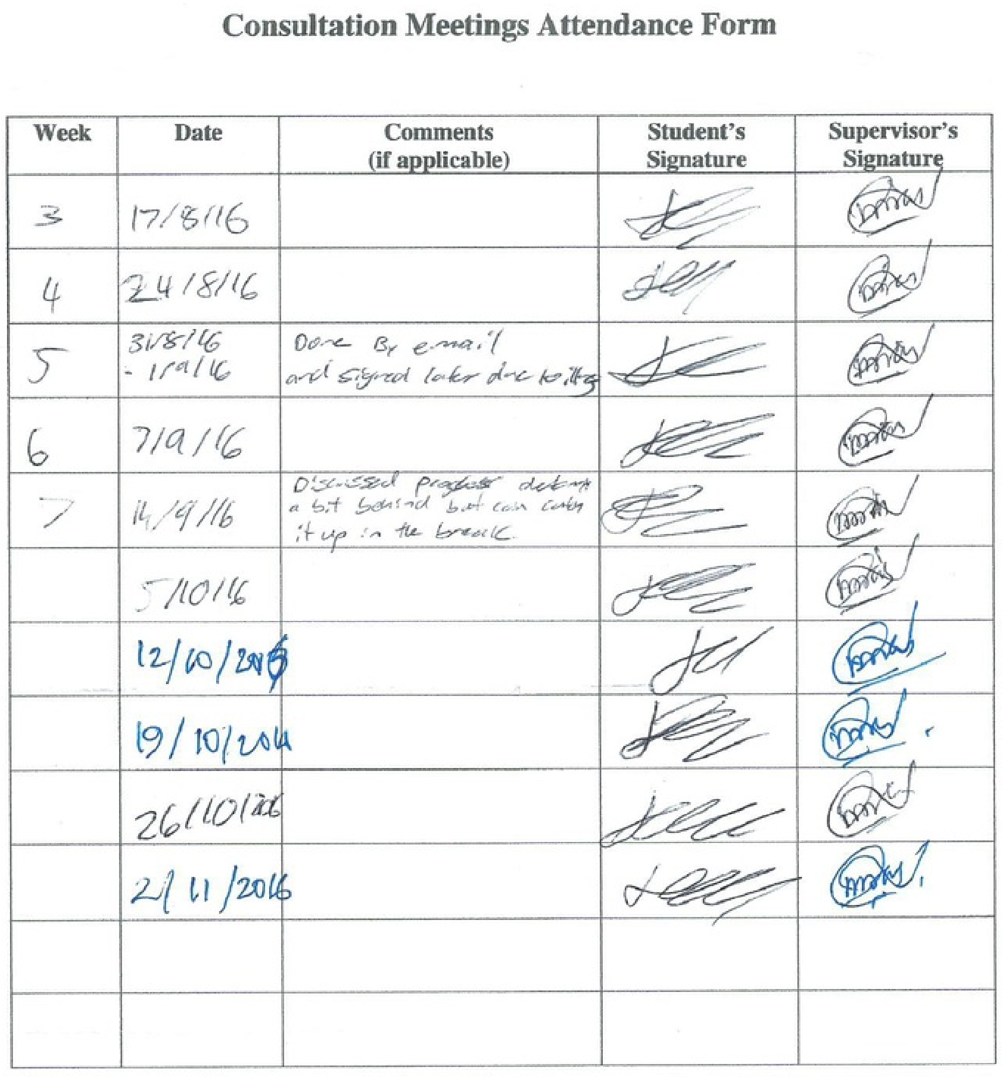
Appendix C

Consultation Meeting Attendance Form

C.1 Overview

This is a scanned version of my Consultation Attendance form,

C.2 Scanned Form



Consultation Meetings Attendance Form

Week	Date	Comments (if applicable)	Student's Signature	Supervisor's Signature
3	17/8/16		[illegible]	[illegible]
4	24/8/16		[illegible]	[illegible]
5	31/8/16 - 1/9/16	Done By e-mail and signed later due to illness	[illegible]	[illegible]
6	7/9/16		[illegible]	[illegible]
7	14/9/16	Discussed product details a bit rushed but can catch it up in the break.	[illegible]	[illegible]
	5/10/16		[illegible]	[illegible]
	12/10/2016		[illegible]	[illegible]
	19/10/2016		[illegible]	[illegible]
	26/10/2016		[illegible]	[illegible]
	2/11/2016		[illegible]	[illegible]

Bibliography

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

[2] M. Budagavi, A. Fuldseth, G. Bjntegaard, V. Sze, and M. Sadafale, "Core transform design in the high efficiency video coding (hevc) standard," *IEEE Journal of Selected Topics in Signal Processing*, vol. 7, no. 6, pp. 1029–1041, Dec 2013.

[3] M. Masera, M. Martina, and G. Masera, "Adaptive approximated dct architectures for hevc," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. PP, no. 99, pp. 1–1, 2016.

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

[5] J.-R. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan, and T. Wiegand, "Comparison of the coding efficiency of video coding standardsincluding high efficiency video coding (hevc)," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1669–1684, 2012.

[6] K. R. Rao and P. Yip, *Discrete cosine transform: algorithms, advantages, applications.* Academic press, 2014.

[7] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the high efficiency video coding (hevc) standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1649–1668, Dec 2012.

[8] J. V. Team, "Advanced video coding for generic audiovisual services," *ITU-T Rec. H*, vol. 264, pp. 14 496–10, 2003.

[9] P. Topiwala, M. Budagavi, A. Fuldseth, R. Joshi, and E. Alshina, "Ce10: Summary report on core transform design," 2011.

[10] N. Vasconcelos. Discrete cosine transform. [Online]. Available: http://www.svcl.ucsd.edu/courses/ece161c/handouts/DCT.pdf