POLITECNICO DI TORINO

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea

DCT-V for Video Coding: A reconfigurable implementation for length 32 and 4

Relatore:
Prof. Maurizio Martina - Politecnico Di Torino

Candidato:
Jurgen Kello

Settembre 2018

Table of contents

Summary

This work of thesis shows a new factorization and its implementation for DCT-V(Discrete Cosine Transform) of length 4 and 32, used in the most recent video compression standard.
During the last years, the Joint Video Exploration Team(JVET) of ITU-T VCEG and ISO/IEC MPEG, now the Joint Video Experts Team, are trying to develop the future video coding technology, that will improve the compression efficiency with respect to the current HEVC standard. An experimental software was introduced, JEM, based on the HEVC Model (HM) software, developed as reference for the HEVC standard. One of the fundamental data compression techniques used for this standard, as in many previous image and video coding standards, is transform coding. What is interesting to consider for the future video coding is the introduction of the new transforms like DCT-V, DCT-VIII, DST-I and DST-VII. This new transforms need low-complexity factorization to efficiently compute them.
This work exploits a factorization of the DCT using the algebraic theory, the Reznik decomposition for DCT-II and an already known fast algorithm for the DCT-V of length 4. In particular, all this known relationships are combined together to find that DCT-V of length 32 can be computed using 5 of the same type DCTs of length 4. Moreover, a Matlab model has been written for functional validation, followed by a model written in C language. This allows for the evaluation of the rate-distortion performance within the video codex for the new DCT-V of length 32 and 4. To do this, it is important to adapt the proposed algorithms for DCT-V with the existing DCT-V description in the JEM software, that is a normalized version with integer coefficients. Finally a reconfigurable unit exploiting multiplexers has been designed, which computes both the DCT-V of length 32 and 5 DCTs of the same type of length 4. For synthesis the umc65 library is used and the reports of time, area and power are obtained.

Chapter 1

introduction

1.1 introduction to video compression

[4.1]Video, from all the different sources of data, is the one that needs more memory. That is why video compression is needed. Video compression can be seen as image compression with a time component. Most of the algorithms on video compression use the temporal correlation to remove redundancy. The current frame is predicted by the previous reconstructed frame. The prediction operation in video coding has to consider the motion of the objects in the frame, known as motion compensation.

Motion compensation
In most video sequences a lot of the images parts are not changing from one frame to the next, even in sequences with a lot of activity. When predicting the frame it is important to consider the motion of the objects in the image. The frame being encoded is divided into blocks of size $M \times M$. For each block, the previous reconstructed frame is searched for the block of size MxM that matches the block being encoded. Fixing a prespecified threshold, the block being encoded is declared uncompensable and is encoded without prediction if the distance between this block and the closest one in the previous reconstructed frame is greater than this threshold. This fact will also be transmitted to the receiver. If the distance is below the threshold then a motion vector is transmitted to the receiver. This motion vector is the relative location of the block to be used for prediction. A way to reduce the number of times the motion compensation is performed, is by increasing the size of the blocks. In this way there will be less blocks per frame but also more computations per comparison. The drawback is that by increasing the block size, it also increases the probability of objects inside the block moving in different directions. The motion compensation is performed in what are called macroblocks.

The earliest video coding standard is the ITU-T H.261 standard. A block diagram of the H.261 video coder is shown in Figure 1.1. An input image is divided into blocks of 8×8 pixels. For a given block it is subtracted the prediction using the previous frame. The difference between the encoded block and the prediction passes through a DCT transform. Further on the transform coefficients are quantized and by using a variable-length code the quantization label is encoded.

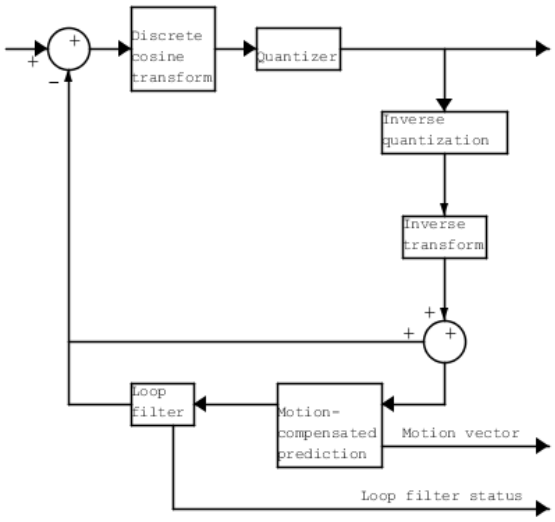

Figure 1.1. Block diagram of the ITU.T H.261 encoder

The loop filter
There is a problem with sharp variations in the prediction error. They can cause high values for the high-frequency coefficients in the transforms. That is why a two-dimensional spatial filter is used to smoothen the prediction block. The filter can be implemented as a one-dimensional filter that first operates on the rows then on the columns.

The transform
The key idea in using transforms is to take a sequence of inputs and transforming them into another sequence limiting the information in only a few elements. So later on it can be used this new sequence, which can be encoded and transmitted resulting in data compression. For the case of interest the transform operation is performed on an 8x8 block of pixels or pixel differences using a DCT. If the transform operation is performed on a block level, either a block or the difference between the block and its predicted value is quantized and transmitted to the receiver. The transform operation is performed on an 8x8 block of pixels if no close match is provided by the motion compensation operation.

Quantization and coding
The characteristics of the coefficients to be quantized depend on the quality of the prediction. A wide variation range is possible, that is why the H.261 code switches between 32 different quantizers, possibly from one mackroblock to the next. Each mackroblock is preceded by a header which gives information of the quantizer used. In the case when the sequence has a constant amount of motion then it can be expected to use the same quantizer for a large number of makroblocks. In this case the previous method of identifying a quantizer with each macroblock would be wasteful. That is why the macroblocks are organized into what are called group of blocks(GOBs).

Rate controller
Rate control is the name used to describe the constant back and forth comunication between the transform coder and the transmission buffer. The function of this later is to keep the output rate of the encoder fixed. If the buffer starts filling up faster than the transmission rate than it sends a message to the transform coder to reduce the output from the quantization. If the opposite happens, that is the the buffer is being emptied then a message is send to the transform coder to increase the output form the quantization.

This were some of the key aspects of the H.261 standard, which can be used as a simple introduction to video compression. It can be interesting to see first in a broader view how the standard has evolved comparing it with the latest versions, and then focus the attention on the transform coding(going further till a specific type of DCT) which will be the topic of this thesis work.

1.2 HEVC

[4.2]The increasing diversity of services, the use of HD video and beyond HD formats created the need for coding efficiencies superior to the H.264 standard. HEVC was created to increase video resolution and the use of parallel processing architectures. In the following the key elements of the design by which these goals are achieved and the typical encoder operation, are described. Each picture is been divided into block shaped regions, and its important that this partition is also transmitted to the decoder. The first picture of each video sequence is coded by using intrapicture prediction only, that is a prediction with no dependence from other pictures but only from region to region in the picture. For all the remaining pictures of the sequence interpicture prediction is typically used for most blocks. The encoding process for interpicture prediction consists of comprising a selected reference picture

by choosing the motion data, and predict the samples of each block by choosing the motion vectors(MV) to be applied. Using the MV and mode decision data and applying motion compensation(MC), the encoder and decoder will generate identical interpicture prediction signals. The difference between the original block and its prediction(inter of intra), called residual, is transformed by a linear spatial transform. The coefficients are then scaled, quantized, entropy coded and transmitted together with the prediction information. In order that both encoder and decoder generate identical predictions for subsequent data, the encoder should duplicate the decoder processing loop. Therefore, by inverse scaling are constructed the quantized transform coefficients, then to duplicate the decoded approximation of the residual signal the inverse transform has been applied. The residual is added to the prediction and the result is filtered to smooth out artifacts that come as a result of block-wise processing and quantization. The final picture representation will be stored in a buffer and will be used in the prediction of subsequent pictures. In general, the order which the pictures come from the source is not the same with the order they are processed(encoding or decoding). Among the various features involved in hybrid coding using HEVC, and later on used also for future video coding, what is more interesting for us is the transform coding.

1.3 JEM

[4.3]The JEM software is based on the HEVC Test Model 16 (HM) architecture. It introduced many new coding tools which give a significant coding gain. For the Coding Tree Unit and Transform Units, larger sizes are considered(up to 256x256 and 64x64). In regard to the prediction modes, additional intra prediction modes are introduced, motion vector storage is more accurate, and the introduction of a so called bi-directional optical flow. What else is introduced is an improved adaptive loop filter and an enhanced CABAC design. Based on the Enhanced Multiple Transform is introduced the Adaptive Multiple Transform(AMT), which includes DCT-II, DST-VII, DCT-VIII, DST-I and DCT-V core transforms. This AMT enables a transform-unit level signaling where the applied horizontal and vertical transforms are derived based on a prediction mode dependent set of transforms. On top of the AMT scheme is placed a so called Non-Separable Secondary Transform(NST). According to a set derived from the selected prediction mode, a 4x4 NST is applied on each 4x4 transform coefficient groups. Finally, a variable sized Karhunen-Loeve Transform(form 4x4 to 32x32) can be obtained by activating in software the Signal Dependent Transform.

1.4 transform coding

Since the introduction of the H.261 standard, the successive generations of standards have improved the way to de-correlate the residual signals thus improving the coding efficiency. In the context of future video coding, a new approach called Adaptive Multiple Transform(AMT) has been introduced, based on the enhanced multiple transform(EMT)[4.4]. For Intra coding a mode-dependent transform candidate selection process is used. This way of selection comes from the different residual statistics associated to each mode. The newly introduced transforms in this case are DCT-V, DCT-VIII, DST-I and DST-VII. On the other hand, when the block is inter predicted, only one set is used made of DST-VII and DCT-VIII. So, summarizing everything, there are two transform candidates for each set, which should be evaluated for both the horizontal and vertical transform. In total there are five different transform candidates(the four multiple transform candidates of the AMT and the DCT-II) which have to be computed for each block in all the different prediction modes. The overall result is a very high complexity at the encoder side. Therefore, this explains the need for low-complexity factorization of different DCT types. But, while several fast algorithms have been proposed to compute the so called even type DCTs, only a few consider the problem of odd type DCTs(types V, VI, VII and VIII). This was also the starting point of this thesis work, finding a low-complexity factorization to efficiently compute DCT-V.

Chapter 2

The proposed algorithm

2.1 Puschel algorithm

The paper considered for this thesis uses a new approach for the derivation of the fast algorithms, the algebraic signal processing theory [4.5]. What is usually done in literature is to derive the algorithms by manipulating the transform coefficients. The derivation of the fast algorithms in [4.5] is algebraic: instead of manipulating the entries of the transform matrix, it derives the algorithms by stepwise decomposition of the associated signal models, or polynomial algebras. For the purpose of this thesis, the algorithm considered for implementation is the one for **DCT-V**, which is derived using the factorization properties of the *Chebyshev* polynomials:

$$DCT5_{3m+2} = Q_m^{3m+2}(DCT5_{m+1} \oplus DCT3_{2m+1}(\frac{2}{3}))B_{3m+2}^{(C5)} \tag{2.1}$$

From the given algorithm 2.1, different dimension **DCT5** can be computed. The **DCT5** dimensions included in the JEM software [4.6], and so the dimensions which are of interest to be considered are **N**= 4, 8, 16 and 32. Trying to substitute for the above equation 2.1, $3m + 2 = N$, an integer **N** can be found only for the case of 8 and 32 dimensions. While no integer solutions can be found for the 4 and 16 point matrices. As previously stated equation 2.1 is based on the factorization properties of the *Chebyshev* polynomials. For the **DCT5** case no other decomposition is possible, that means that this paper finds a solution only to the cases of 8 and 32 points.

2.2 $DCT5_{32}$

From the given algorithm 2.1, considering the case of **N=32**, the matrix decomposition becomes:

$$DCT5_{32} = Q_{10}^{32}(DCT5_{11} \oplus DCT3_{21}(\frac{2}{3}))B_{32}^{(C5)} \tag{2.2}$$

The same algorithm 2.1 can be used for the $DCT5_{11}$ case:

$$DCT5_{11} = Q_{3}^{11}(DCT5_{4} \oplus DCT3_{7}(\frac{2}{3}))B_{11}^{(C5)} \tag{2.3}$$

In 2.1, the pre-addition matrix **B** and permutation matrix **Q**, are defined as follows:

$$B_{3m+2}^{(C5)} = \left(\begin{array}{cc|cc} 1 & & 1 & \\ I_m & J_m & & I_m \\ \hline & & -1/2 & \\ I_{2m+1} & & & -I_m \\ & & & -J_m \end{array}\right) \tag{2.4}$$

where I_n is the n x n identity matrix and J_n, the opposite identity matrix(*i.e.* I_n with columns in reversed order).

$$Q_m^{3m+2} = i_1 + 3i_2 \mapsto \begin{cases} i_2, & \text{for } i_1 = 0; \\ 2i_2 + m + 1, & \text{for } i_1 = 1; \\ 2i_2 + m + 2, & \text{for } i_1 = 2; \end{cases} \tag{2.5}$$

This notation 2.5, is used to define a permutation matrix. Matrix Q_m^{3m+2} has in row $i_1 + 3i_2$ the only entry 1 at position:

- i_2 if i_1 is 0.
- $2i_2 + m + 1$ if i_1 is 1.
- $2i_2 + m + 2$ if i_2 is 2.

This means that i_1 will take only three possible values (0, 1, 2).
From the initial algorithm, what can be furtherly decomposed are the **DCT3** matrices. In [4.5] are included also these algorithms:

$$DCT3_{km}(r) = K_m^n(\bigoplus_{0 \le i < k} DCT3_m(r_i))(DCT3_k(r) \otimes I_m)B_{k,m}^{(C3)} \tag{2.6}$$

For the case of interest, $DCT3_7$, it can not be decomposed further since no integer values k, m could be found such that the product $k \cdot m = 7$ is a prime number.

Meanwhile for the $DCT3_{21}$ case the matrix decomposition will be:

$$DCT3_{21}(r) = K_7^{21}(\bigoplus_{0\leq i<3} DCT3_7(r_i))(DCT3_3(\frac{2}{3}) \otimes I_7)B_{3,7}^{(C3)} \tag{2.7}$$

In the equation 2.7, some important building blocks of the algorithm can be noted. They are called **skew DCTs**. The theory behind skew DCTs can be seen more in detail in another paper [4.7], but what is important for the matrix decomposition is only the fact that every skew DCT can be translated in its non-skew counter part:

$$DCT_n(r) = DCT_n \cdot X_n^{(*)}(r) \tag{2.8}$$

where $X_n^{(*)}(r)$ depends on the DCT in consideration. For the actual case of interest:

$$X_n^{(C3)}(r) = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & c_1 & & & s_{n-1} \\ \vdots & & \ddots & \iddots & \\ \vdots & & \iddots & \ddots & \\ 0 & s_1 & & & c_{n-1} \end{bmatrix} \tag{2.9}$$

In equation 2.9, $c_l = cos(1/2 - r)l\pi/n$, $s_l = sin(1/2 - r)l\pi/n$
What still remains to be explained is, in the case of $DCT3_7(r_i)$ in 2.7, what do those r_i stand for? They are actually the zeros of the *Chebyshev* polynomial $T_n - cos(r\pi)$.
It can be proved that for the case of n even, they can be found from the relation:

$$(r_l)_{0\leq l<n} = (\bigcup_{0\leq i<\frac{n-1}{2}} (\frac{r+2i}{n}, \frac{2-r+2i}{n})) \cup (\frac{r+n-1}{n}) \tag{2.10}$$

Substituting for the case of interest r=2/3, three r_i are found: 2/9, 4/9 and 8/9. Those are the values to be substituted in the final algorithm 2.7.
The $DCT3_{km}(r)$ algorithm 2.7 contains also two matrices, K_m^n and $B_{k,m}^{(C3)}$, the permutation and pre-addition matrix respectively. The definition for the pre-addition matrix is given as:

$$B_{3,m}^{(C3)} = (I_m \oplus (I_2 \otimes diag(1,2,\ldots,2))) \begin{bmatrix} I_m & -Z_m & I_m' \\ & I_m & -Z_m \\ & & I_m \end{bmatrix} \tag{2.11}$$

In 2.11 the special case for k=3 has been considered, where $I_m^{'} = diag(0,1,\ldots,1)$ and

$$Z_m = \begin{bmatrix} & & & 0 \\ & & 0 & 1 \\ & \iddots & \iddots & \\ 0 & 1 & & \end{bmatrix}$$

While the permutation matrix takes the form:

$$K_m^n = (I_k \oplus J_k \oplus I_k \oplus J_k \oplus \ldots) L_m^n \tag{2.12}$$

In 2.12, L_m^n is the stride permutation matrix which can be defined as
$L_m^n : \quad i \mapsto im \ mod \ n-1, \quad for \ 0 \leq i < n-1$
$\qquad\qquad n-1 \mapsto n-1$
It should be noted that diagonal matrices are written as $diag(x_0,\ldots,x_{n-1})$.
Further the matrix operator used, the direct sum, is defined as:

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}$$

and the Kronecker product is:
$A \otimes B = [a_{k,l}B]_{k,l}, \quad for A = [a_{k,l}]$.

These last blocks definition bring an end to the algorithm introduced by [4.5], which gives a decomposition of the $DCT5_{32}$ into simpler matrices 2.2. There are some missing pieces to this algorithm, since it does not define a way how to find the smaller blocks like $DCT5_4$, $DCT3_7$ and $DCT3_3$. To find fast algorithms for these last matrices, other papers are considered which will be explained in the following. To end this part there is a summary of the algorithm introduced till now, where for abbreviation instead of DCT is simply written C, also since it is always considered X for (C3) type then this last one is omitted so simply X will be written:

$$C5_{32} = \left[Q_{10}^{32}\right] \begin{bmatrix} \left[Q_3^{11}\right] \begin{bmatrix} \left[C5_4\right] & \\ & \left[C3_7\right]\left[X_7(\frac{2}{3})\right] \end{bmatrix} \left[B_{11}^{C5}\right] & \\ & \left[K_7^{21}\right] \begin{bmatrix} \left[C3_7\right]\left[X_7(\frac{2}{9})\right] & & \\ & \left[C3_7\right]\left[X_7(\frac{4}{9})\right] & \\ & & \left[C3_7\right]\left[X_7(\frac{8}{9})\right] \end{bmatrix} \left[C3_3X_3(\frac{2}{3}) \otimes I_7\right] \left[B_{3,7}^{(C3)}\right] \end{bmatrix} \left[B_{32}^{(C5)}\right] \quad (2.13)$$

To define the missing blocks, is done reference to some further literature. The algorithm for $DCT5_4$ has been found in [4.8], where the signal flow graph (SFG) is given as follows:

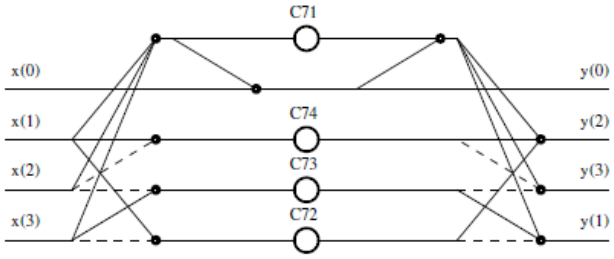

Figure 2.1. $DCT5_4$ SFG

To find the algorithm for $DCT3_7$ the same procedure followed in this paper [4.8], will be used. The starting point will be the decomposition of **DCT2** into a **DCT6** and **DST7**, [4.9]. Applying this concept to the case of interest, the result is:

$$DCT2_7 = Q_7 \begin{bmatrix} DCT6_4 & \\ & DST7_3 \end{bmatrix} \begin{bmatrix} I_3 & & J_3 \\ & 1 & \\ -J_3 & & I_3 \end{bmatrix} \tag{2.14}$$

To write the algorithm for $DCT3_7$ several relationships have to be considered. The starting point will be to use the property of **DCT3**: $DCT3 = (DCT2)^T$. From 2.14 it can be seen that $DCT2_7$ is given as a product of three matrices. Using the properties of the transpose for the product $(A \cdot B)^T = B^T \cdot A^T$, it will be applied to the product of three matrices case

$$(A \cdot B \cdot C)^T = ((A \cdot B) \cdot C)^T = C^T \cdot (A \cdot B)^T = C^T \cdot B^T \cdot A^T$$

Then for the composing matrix **DCT6**, using the relationship introduced in [4.8] $DCT6_n = D_n \cdot DCT5_n \cdot J_n$ and again doing the transpose, the final algorithm for $DCT3_7$ can be written as:

$$DCT3_7 = \begin{bmatrix} I_3 & & -J_3 \\ & 1 & \\ J_3 & & I_3 \end{bmatrix} \begin{bmatrix} J_4 \cdot DCT5_4 \cdot D_4 & \\ & DST7_3^T \end{bmatrix} Q_7^T \tag{2.15}$$

In this algorithm 2.15, $DCT5_4$ fast algorithm is known, Q is a sign alteration and reordering matrix introduced in [4.9] as in the following(for Q_{2N+1}):
$y(2n) = x(n)$ n=[0,N]
$y(2n+1) = (-1)^{n+1} \cdot x(N+1+n)$ n=[0, N-1]

D_N is a diagonal matrix implementing sign-alteration:

$$D_N = \begin{bmatrix} 1 & & & & & \\ & -1 & & & & \\ & & \ddots & & & \\ & & & 1 & & \\ & & & & -1 & \end{bmatrix}$$

For the $DST7_3$ algorithm the same procedure used in [4.8] can be followed, but this time the lower part of the signal flow graph has to be considered. The case of interest is $DST7_3^T$. To find the algorithm of the transpose, as explained in [4.10], simply the network transposition has to be done. Considering the signal flow graph used to represent the algorithm of a linear network, where the edges represent multiplication by a constant(the discontinuous edges do a sign change) and the vertices represent addition of the incoming edges. Network transposition simply consists of reversing the direction of every edge in a directed graph, which could simply be seen as starting the computation from the outputs instead of the inputs of the original graph. After using this principle and doing some final sign manipulations and the necessary permutations, the final signal flow graph for $DST7_3^T$ is derived.

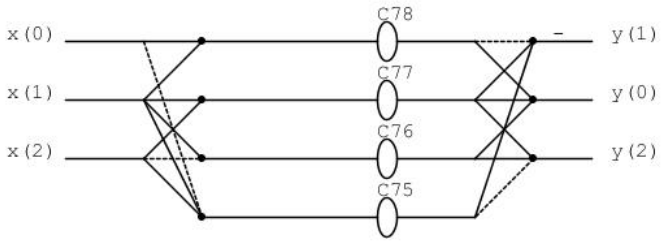

Figure 2.2. $DST7_3^T$ SFG

The final remaining block for which the fast algorithm has to be defined is $DCT3_3$. From [4.11] an odd length **DCT2** is derived from the real valued **DFT** of the same length. The algorithm for the case of interest N=3 is also given in this paper. Moving from the DCT2 to DCT3 algorithm can be easily done as explained in [4.10]. Since the DCT3 is the transpose of DCT2, a fast DCT3 algorithm is obtained by network transposition of a fast DCT2 algorithm. The final SFG will be:

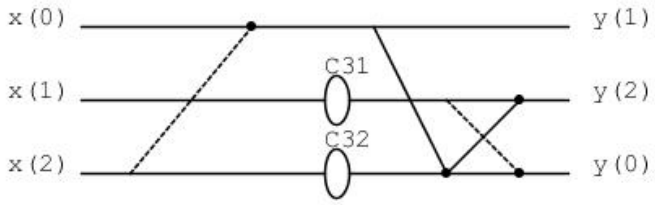

Figure 2.3. DCT3 SFG

The coefficients used for all this algorithms are introduced by [4.12] for the computation of the DFT. In table 2.1 are defined all coefficients used:
Starting from the algorithm in 2.13 and substituting the decomposition in 2.15,

Coefficient	value
C31	$-\frac{\sqrt{3}}{2}$
C32	1.5
C71	$\frac{7}{6}$
C72	$\frac{2cos(u)-cos(2u)-cos(3u)}{3}$
C73	$\frac{cos(u)-2cos(2u)+cos(3u)}{3}$
C74	$\frac{cos(u)+cos(2u)-2cos(3u)}{3}$
C75	$\frac{sin(u)+sin(2u)-sin(3u)}{3}$
C76	$\frac{2sin(u)-sin(2u)+sin(3u)}{3}$
C77	$\frac{sin(u)-2sin(2u)-sin(3u)}{3}$
C78	$\frac{sin(u)+sin(2u)+2sin(3u)}{3}$

Table 2.1. Coefficients of the algorithm

for the $DCT3_7$, blocks an interesting result can be noted. To compute the **DCT5** matrix of dimensions *32x32*, 5 smaller **DCT5** matrices of dimensions *4x4* have to be computed. Since both **DCT5** of 32 and 4 samples are included in 3.1, then it is possible to implement them by a single hardware component. That is, by applying this solution and trying to implement the $DCT5_{32}$ block, it will at the same time implement also 5 $DCT5_4$ blocks. This means that the important concept of resource sharing has been used in this case, reducing the HW cost needed to implement the different coding tools included in 3.1. This was the reason why this idea was chosen to be developed further in this thesis work.

2.3 $DCT5_8$

The same algorithm introduced in [4.5] allows the factorization of $DCT5_8$ matrix. In this case it will be:

$$DCT5_8 = Q_2^8(DCT5_3 \oplus DCT3_5(\frac{2}{3}))B_8^{(C5)} \tag{2.16}$$

Again matrices **Q** and **B** can be found as previously described, matrix **Q** being a permutation matrix gives no operation cost, while the pre-addition **B** matrix requires 10 additions and 1 right shift. Moving on to the block diagonal matrix, one of the blocks in the diagonal is the skew matrix $DCT3_5(\frac{2}{3})$ which as previously described can be written as:
$DCT3_5(\frac{2}{3}) = DCT3_5 \cdot X_5^{(C3)}(\frac{2}{3})$ The $X_5^{(C3)}(\frac{2}{3})$ will introduce 8 multiplications and 4 additions, while to compute $DCT3_5$ again [4.11] is considered and then it is done the network transposition(this last operation does not change the operation count). The signal flow graph for the $DCT3_5$ fast algorithm is given below in figure 2.4:

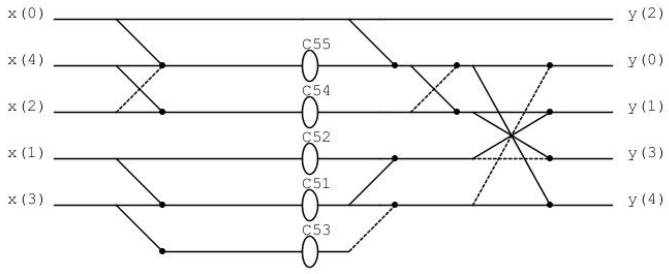

Figure 2.4. $DCT3_5$ SFG

As it can be seen there are 5 multiplications and 13 additions.
To find the fast algorithm for the remaining block $DCT5_3$, the same procedure used in [4.8] is followed, but instead of starting from $DCT2_7$, this time it is started from $DCT2_5$. The final signal flow graph is given below in figure 2.5:

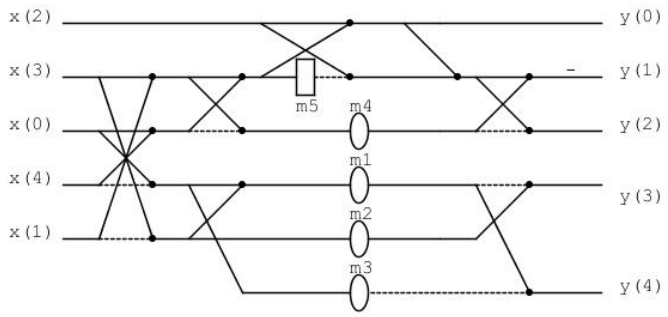

Figure 2.5. $DCT3_5$ SFG

The top part in figure 2.5, considering the upper three rows, is the signal flow graph for the $DCT6_3$. Using the relation given by [4.8], $DCT6_n = D_n \cdot DCT5_n \cdot J_n$, the SFG for $DCT5_3$ can be found just by doing some permutations and sign variations. The result is given in figure 2.6:

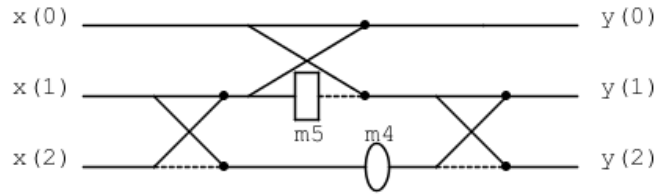

Figure 2.6. $DCT5_3$ SFG

For the different algorithms and signal flow graphs described in this section, the used coefficients are given in table 2.2:

Coefficient	value
m1	0.95106
m2	1.53884
m3	0.36327
m4	0.55902
m5	$0.25(\gg 2)$

Table 2.2. $DCT3_5$ coefficients

So the fast algorithm for $DCT5_3$ will introduce 6 additions, 1 multiplication and 1 right shift of 2. In overall the operation count for the complete $DCT5_8$ algorithm will be 33 additions, 14 multiplications and 2 shifts.

What could be noticed from the beginning was the fact that for this algorithm, differently from the $DCT5_{32}$ case, no reusable blocks could be identified. Meaning that this algorithm has to be implemented as standalone, without being able to use the principle of resource sharing. This makes the $DCT5_{32}$ block more interesting to be implemented with respect to the $DCT5_8$. So for the next stages of the work for this thesis only the $DCT5_{32}$ algorithm has been considered.

Chapter 3

MatLab model

3.1 normalized algorithm model

After developing the algorithm to be implemented, the next step will be to create a MatLab model to be able to better simulate and test the algorithm. Before starting with the model, some final clarifications have to be done. What should be noted is the fact that [4.5] defines a non normalized algorithm. This is different from the software [4.6], which is using a normalized matrix for the DCTs, as shown in the code below extracted from the software [4.6]:

```
c = 4;
for ( i=0; i<5; i++ )
{
        short *iT = NULL;
        const double s = sqrt((double)c) * (64<<2);     //
    COM16_C806_TRANS_PREC=2


        for( int k=0; k<c; k++ )
        {
                for( int n=0; n<c; n++ )
                {
                        double w0, w1, v;

                        // DCT-V
                        w0 = ( k==0 ) ? sqrt(0.5) : 1.0;
                        w1 = ( n==0 ) ? sqrt(0.5) : 1.0;
                        v = cos(PI*n*k/(c-0.5)) * w0 * w1 * sqrt
    (2.0/(c-0.5));
```

```
18                          iT[1*c*c + k*c + n] = (short) ( s * v +
    ( v > 0 ? 0.5 :-0.5));//DCT5=1
19                      }
20
21              }
22              c <<= 1;
23  }
```

Listing 3.1. JEMs DCT5 definition

All the details are not important for the scope of the thesis, what is needed is just to notice the fact that **v** represents every entry of the **DCT5** matrix. This entries are normalized, since are multiplied also with the factors **w0**, **w1** and $\sqrt{\frac{2}{c-0.5}}$ where **c** stands for the number of samples in consideration, in this case c=32. **w0** and **w1** will take the value $\sqrt{\frac{1}{2}}$ for the first row and column of the **DCT5** matrix respectively. But in comparison, the algorithms described till now are using the non normalized definition of the DCT5: $[DCT5_N]_{k,l} = cos(\frac{2\pi kl}{2N-1}) \qquad k,l = [0,N-1]$ which will have the first row and the first column simply all equal to one. Listing the two matrices helps to understand better the differences between the two. Starting from an input vector

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix}$$

the non normalized DCT5 can be computed by doing the matrix product:

$$\begin{aligned} \mathbf{Z} = \begin{bmatrix} Z_0 \\ Z_1 \\ \vdots \\ Z_{N-1} \end{bmatrix} = [DCT5_N] \cdot \mathbf{x} = \\ = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & cos(\frac{2\pi}{2N-1}) & cos(\frac{4\pi}{2N-1}) & \dots & cos(\frac{(N-1)2\pi}{2N-1}) \\ 1 & cos(\frac{4\pi}{2N-1}) & cos(\frac{8\pi}{2N-1}) & \dots & cos(\frac{2(N-1)2\pi}{2N-1}) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & cos(\frac{(N-1)2\pi}{2N-1}) & cos(\frac{2(N-1)\pi}{2N-1}) & \dots & cos(\frac{(N-1)^2 2\pi}{2N-1}) \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{bmatrix} \end{aligned} \tag{3.1}$$

Another way to write this is by using the equation 3.2:

$$Z_n = \sum_{k=0}^{N-1} x_k cos(nk\frac{2\pi}{2N-1}) \quad for\ n = 0,1,\dots,N-1 \tag{3.2}$$

Instead in the normalized case the DCT5 will be computed by the matrix product:

$$\mathbf{Y} = \begin{bmatrix} Y_0 \\ Y_1 \\ \vdots \\ Y_{N-1} \end{bmatrix} = [DCT5_N]_n \cdot \mathbf{x} =$$

$$= \frac{2}{\sqrt{2N-1}} \begin{bmatrix} \frac{1}{2} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \cdots & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & cos(\frac{2\pi}{2N-1}) & cos(\frac{4\pi}{2N-1}) & \dots & cos(\frac{(N-1)2\pi}{2N-1}) \\ \frac{1}{\sqrt{2}} & cos(\frac{4\pi}{2N-1}) & cos(\frac{8\pi}{2N-1}) & \dots & cos(\frac{2(N-1)2\pi}{2N-1}) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{1}{\sqrt{2}} & cos(\frac{(N-1)2\pi}{2N-1}) & cos(\frac{2(N-1)\pi}{2N-1}) & \dots & cos(\frac{(N-1)^2 2\pi}{2N-1}) \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{bmatrix} \quad (3.3)$$

which again can be expressed as an equation of the form:

$$Y_n = \frac{2}{\sqrt{2N-1}} T_n \sum_{k=0}^{N-1} x_k T_k cos(nk \frac{2\pi}{2N-1}) \quad for\ n = 0,1,\dots,N-1 \qquad (3.4)$$

where

$$T_n = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } n = 0; \\ 1, & \text{if } n \neq 0; \end{cases}$$

$$T_k = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } k = 0; \\ 1, & \text{if } k \neq 0; \end{cases}$$

Comparing the two equations, 3.2 and 3.4, a connection between the non normalized and normalized DCT5 can be found. Starting from the normalized equation 3.4, it can be written as:

$$Y_n = \frac{2}{\sqrt{2N-1}} T_n (\frac{1}{\sqrt{2}}) x_0 + \frac{2}{\sqrt{2N-1}} T_n \sum_{k=1}^{N-1} x_k cos(nk \frac{2\pi}{2N-1}) \quad for\ n = 0,1,\dots,N-1 \qquad (3.5)$$

Separating the cases between n=0 and $n \neq 0$ it can decompose into this two equations:

$$Y_0 = \frac{1}{\sqrt{2}} (\frac{2}{\sqrt{2N-1}} \frac{1}{\sqrt{2}} x_0 + \frac{2}{\sqrt{2N-1}} \sum_{k=1}^{N-1} x_k cos(nk \frac{2\pi}{2N-1})) \quad for\ n = 0 \qquad (3.6)$$

$$Y_n = \frac{2}{\sqrt{2N-1}} \frac{1}{\sqrt{2}} x_0 + \frac{2}{\sqrt{2N-1}} \sum_{k=1}^{N-1} x_k cos(nk \frac{2\pi}{2N-1}) \quad for\ n = 1,\dots,N-1 \qquad (3.7)$$

The summation in the two equations, 3.6 and 3.7, is similar to the non normalized algorithm 3.2(what is changing is just the first element of the sum). To have the equality, an initial vector

$$\tilde{\mathbf{x}} = \begin{bmatrix} 0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{bmatrix}$$

has to be considered. It is exactly the same starting vector used for the two algorithms 3.2 and 3.4, with exception only to the first element which is 0. Using this vector in the previous two equations 3.6 and 3.7, everything can be rewritten as:

$$Y_n = \begin{cases} \frac{1}{\sqrt{2}}(\frac{2}{\sqrt{2N-1}}(\frac{1}{\sqrt{2}})x_0 + \frac{2}{\sqrt{2N-1}}\sum_{k=0}^{N-1}\tilde{x}_k cos(nk\frac{2\pi}{2N-1})) & for\ n = 0 \\ \frac{2}{\sqrt{2N-1}}(\frac{1}{\sqrt{2}})x_0 + \frac{2}{\sqrt{2N-1}}\sum_{k=0}^{N-1}\tilde{x}_k cos(nk\frac{2\pi}{2N-1}) & for\ n = 1,\ldots,N-1 \end{cases} \tag{3.8}$$

which in a further step can be modified as:

$$Y_n = \begin{cases} \frac{1}{\sqrt{2}}(\frac{2}{\sqrt{2N-1}}(\frac{1}{\sqrt{2}})x_0 + \frac{2}{\sqrt{2N-1}}Z_0) & for\ n = 0 \\ \frac{2}{\sqrt{2N-1}}(\frac{1}{\sqrt{2}})x_0 + \frac{2}{\sqrt{2N-1}}Z_n & for\ n = 1,\ldots,N-1 \end{cases} \tag{3.9}$$

where from 3.2, Z is the non normalized DCT5, substituting the summation. So this last equation 3.9 gives the connection between the non normalized and the normalized DCT5. This equation can be broken in steps, explaining the procedure to be used in obtaining the final normalized result. This steps can be listed as:

1. Starting from the non normalized DCT5 algorithm that is introduced in 2.13, this algorithm is applied to the modified input vector, which has the first element equal to 0.
2. Multiply the output vector of the non normalized algorithm with $\frac{2}{\sqrt{2N-1}}$
3. Sum to every element of the output vector the term $\frac{2}{\sqrt{2N-1}}(\frac{1}{\sqrt{2}})x_0$
4. For the first element of the output vector an extra multiplication with $\frac{1}{\sqrt{2}}$ has to be done

In overall the computational cost seems to be 33 extra multiplications and 32 extra additions. Making reference to equation 2.13, exploiting its form and the fact that a linear algorithm is been considered, some final modifications can be done to reduce

the operation count. The idea is to try to integrate the final additions and multiplications in some earlier stages of the algorithm. To better explain the following passages a schematic representation of the algorithm is given in figure 3.1:

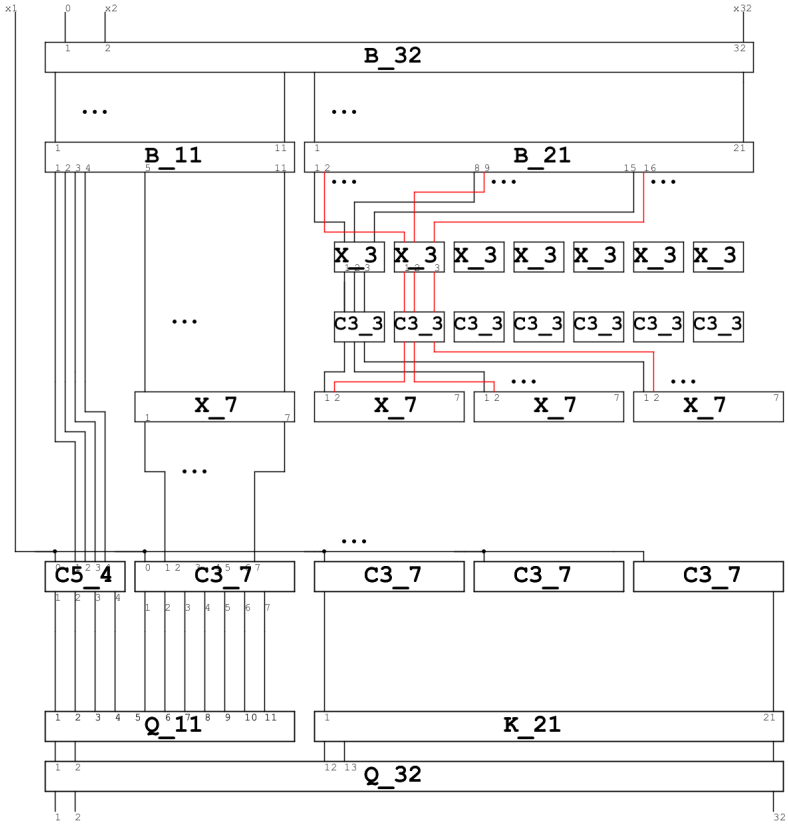

Figure 3.1. algorithm schematic

Looking at the schematic of the algorithm, one possibility for integrating the final multiplications and additions can be to integrate them in the $DCT5_4$ and $DCT3_7$ blocks, since they are the last stage that produces the output(after this stage only the permutation matrices remain, which just change the position of the output elements). In the following, a first version of the $DCT5_4$ and $DCT3_7$ non normalized blocks is written in Matlab 3.2, and then a second normalized version is introduced, modifying the first so that it does the final additions and multiplications 3.3.

```
function y=dct5_4(x)
        %constants
        u=2*pi/7;
        C(1)=-7/6;
```

```
        C(2)=-(2*cos(u)-cos(2*u)-cos(3*u))/3;
        C(3)=(cos(u)-2*cos(2*u)+cos(3*u))/3;
        C(4)=(cos(u)+cos(2*u)-2*cos(3*u))/3;
        C(5)=1;
        %preadd
        a1=x(2)+x(3);
        a2=a1+x(4);
        a3=x(2)-x(3);
        a4=x(4)-x(3);
        a5=x(2)-x(4);
        a6=a2+x(1);
        %multiplications
        m1=C(1)*a2;
        m2=C(5)*a6;
        m3=C(4)*a3;
        m4=C(3)*a4;
        m5=C(2)*a5;
        %postadd
        a7=m1+m2;
        a8=a7+m3;
        a9=a8+m5;
        a10=a7-m3;
        a11=a10-m4;
        a12=a7+m4;
        a13=a12-m5;
        %outputs
        y(1)=m2;
        y(2)=a13;
        y(3)=a9;
        y(4)=a11;
        end
```

Listing 3.2. unnormalized DCT5_4

```
function y=ndct5_4(x, x_0, N, first)
        k=2/sqrt(2*N-1);
        %constants
        u=2*pi/7;
        C(1)=-k*7/6;
        C(2)=-k*(2*cos(u)-cos(2*u)-cos(3*u))/3;
        C(3)=k*(cos(u)-2*cos(2*u)+cos(3*u))/3;
        C(4)=k*(cos(u)+cos(2*u)-2*cos(3*u))/3;
        %coeffitients for doing the normalization
        C(5)=1*k;
```

```
        C(6)=k/sqrt(2);
        C(7)=1/sqrt(2);

        %preadd
        a1=x(2)+x(3);
        a2=a1+x(4);
        a3=x(2)-x(3);
        a4=x(4)-x(3);
        a5=x(2)-x(4);
        a6=a2+x(1);
        %multiplications
        m1=C(1)*a2;
        m2=C(5)*a6;
        m3=C(4)*a3;
        m4=C(3)*a4;
        m5=C(2)*a5;
        m6=C(6)*x_0;
        %postadd
        a7_0=m2+m6;%it does the normalization
        a7=m1+a7_0;%a7 will be used in every output
        a8=a7+m3;
        a9=a8+m5;
        a10=a7-m3;
        a11=a10-m4;
        a12=a7+m4;
        a13=a12-m5;
        %partial outputs
        if first==1
                y(1)=a7_0*C(7);
        else
                y(1)=a7_0;
        end
        y(2)=a13;
        y(3)=a9;
        y(4)=a11;
end
```

Listing 3.3. normalized DCT5_4

For the normalized model 3.3, the normalization constant k has been included inside the already existing multiplications. In this way it does not increase the number of operations excessively. This can be done due to the linearity of the algorithm. By including the multiplications in this stage, the output will be just a linear

combination of the multiplications outputs, so the final result will be the same as doing the multiplication at the end. The only thing to be kept in consideration is that every element of that linear combination should be multiplied by k, otherwise the final result will be wrong. That is why compared with the unnormalized case 3.2, where only 4 multiplications were necessary, in the normalized case it is necessary to consider one of the edges in the signal flow graph 2.1 as a multiplication by 1. In this way when doing the normalization, an extra multiplication should be included also, having a coefficient $1 \cdot k$.
C(6) is the coefficient that should multiply x_0, **m6** is the result of this multiplication, and later it should be added to every output. But exploiting the fact that **a7** is used in each of the outputs, then it is necessary to do the sum only once and then it will be included in the computation of every output. Finally the first row should be multiplied also with $\frac{1}{\sqrt{2}}$. The condition in line 39 of 3.3 does that, it includes this multiplication for computing the first element of the output vector if the first $DCT5_4$ is been considered, otherwise it is not included. The algorithm has been defined as a Matlab function, which takes as input arguments **x** and **x_0**. **x** is the input vector of the $DCT5_4$ block, **x_0** is the first element of the input vector of the overall $DCT5_{32}$ algorithm, so it is the first element of a 32 element vector. N is the number of samples for the algorithm, this is used to define between 4 and 32 samples, since will be both computed with the same code. And finally the last input is **first** which takes two possible values, 1 in case the top $DCT5_4$ has been considered, the one used as a single block, not included inside a $DCT3_7$ block, and 0 when the $DCT3_7$ blocks are being considered. In this way can be separated the case when to use the normalization multiplication with C(7) and when to not use it.
After defining the code for the normalized DCT5(the n in the name of the function stands for normalized), the modification of the $DCT3_7$ can be considered:

```
function y=dct3_7(x)
        %first do the permutation
        z_1=Q_Reznik(7)'*x';
        %now multiply the block diagonal matrix
        z_2(5:7)=inv_dst7_3(z_1(5:7));
        %for the dct5_4 part it should be multiplied also with D
    and J matrixes
        z(1:4)=dct5_4([z_1(1) -z_1(2) z_1(3) -z_1(4)]);
        z_2(1:4)=fliplr(eye(4))*z(1:4)';
        %finally do the postadd
        a1=z_2(1) - z_2(7);
        a2=z_2(2) - z_2(6);
        a3=z_2(3) - z_2(5);
```

```
        a4=z_2(4);
        a5=z_2(3) + z_2(5);
        a6=z_2(2) + z_2(6);
        a7=z_2(1) + z_2(7);
        %output assignment
        y(1)=a1;
        y(2)=a2;
        y(3)=a3;
        y(4)=a4;
        y(5)=a5;
        y(6)=a6;
        y(7)=a7;
end
```

Listing 3.4. unnormalized DCT3_7

where the Q_Reznik function, corresponds to the **Q** matrix introduced in [4.9], and its code is as follows:

```
function x=Q_Reznik(M)
        N=(M-1)/2;
        x=zeros(2*N+1, 2*N+1);
        for j=0:N
                x(2*j+1, j+1)=1;
        end
        for j=1:N
                x(2*j, N+j+1)=(-1)^(j);
        end
end
```

Listing 3.5. Q_Reznik

While the code for the normalized algorithm for the $DCT3_7$ block will be:

```
function y=dct3_7(x, x_0, N, first)
        %first do the permutation
        z_1=Q_Reznik(7)'*x';
        %now multiply the block diagonal matrix
        z_2(5:7)=inv_dst7_3(z_1(5:7), N);
        %for the dct5_4 part it should be multiplied also with D
    and J matrixes
        z(1:4)=ndct5_4([z_1(1) -z_1(2) z_1(3) -z_1(4)], x_0, N,
   first);
```

```
        z_2(1:4)=fliplr(eye(4))*z(1:4)';
        %finally do the postadd
        a1=z_2(1) - z_2(7);
        a2=z_2(2) - z_2(6);
        a3=z_2(3) - z_2(5);
        a4=z_2(4);
        a5=z_2(3) + z_2(5);
        a6=z_2(2) + z_2(6);
        a7=z_2(1) + z_2(7);
        %output assignment
        y(1)=a1;
        y(2)=a2;
        y(3)=a3;
        y(4)=a4;
        y(5)=a5;
        y(6)=a6;
        y(7)=a7;
end
```

Listing 3.6. normalized DCT3_7

The code in 3.6 is the same with the unnormalized case **??**, it just has some extra inputs, the one passed to DCT5_4. Since in the algorithm for $DCT3\_7$ the final step is to do the addition between the output elements of $DCT5\_4$ and the output elements of $DST7_3^T$, means that it is included in automatic the sum with x_0 for the normalization. This sum was already included in the $DCT5_4$ algorithm, so the outputs of this matrix, i.e. z_2(1:4) in the code are already summed with that factor. It means that also the final output of $DCT3\_7$ will correctly include that normalization factor. So the x_0 term should not be included in the $DST7_3^T$ algorithm, which in the code corresponds to inv_dst7_3. Also the multiplication with $\frac{1}{\sqrt{2}}$ should not be included in the inv_dst7_3 code. The reason is that after doing some considerations for the permutation matrices which will change the order of the elements of the output vector, none of the $DCT3\_7$ outputs correspond to the first element of the output vector, which is the one multiplied by $\frac{1}{\sqrt{2}}$. So to have correct results, the only modification to be introduced to $DST7_3^T$ is to multiply the existing coefficients with the $\frac{2}{\sqrt{2N-1}}$ constant. A part of the code is shown below in listing 3.7:

```
c=2/sqrt(2*N-1);
u=2*pi/7;
C(5)=c*(sin(u)+sin(2*u)-sin(3*u))/3;
C(6)=-c*(2*sin(u)-sin(2*u)+sin(3*u))/3;
C(7)=-c*(sin(u)-2*sin(2*u)-sin(3*u))/3;
```

```
C(8)=c*(sin(u)+sin(2*u)+2*sin(3*u))/3;
```

Listing 3.7. inv_dst_7 coefficients

As previously stated, the only difference is in the coefficients, which in the normalized case will be multiplied with the constant c.
These were all the blocks which were affected by the normalization. The other matrices in the algorithm should not differ between the normalized and the unnormalized case. Finally the top level algorithm can be written, to every matrix in the original algorithm now corresponds a function call, so the writing of the code becomes very straightforward.

```
function y=puschel_32(x)
        N=32;
        x_norm=[0; x(2:N)];
        %initial preadditions from the B_32 matrix
        y1=B_N_a(x_norm, N);
        %second preadditons for DCT5_11
        %multiplying B11
        y2(1:11)=B_N_a(y1(1:11), 11);
        %third preadditons for DCT3_21
        %multiplying B_C3_3m
        y2(12:32)=B_C3_3m_a(y1(12:32), 7);
        %the block diagonal matrixes made of DCT5_4
        %use the algorithm given for DCT5_4 as a function
        first=1;
        y3(1:4)=ndct5_4(y2(1:4), x(1), N, first);
        first=0;
        y2_1(5:11)=X_C3_a(y2(5:11), 7, 2/3);
        %the result of this will multiply DCT3_7
        y3(5:11)=dct3_7(y2_1(5:11), x(1), N, first);
        %do the diagonal matrix for dct3_21
        %start with the kroneker product
        for i=0:6
                z=X_C3_a([y2(12+i) y2(19+i) y2(26+i)], 3, 2/3);
                t=dct3_3(z);
                y2_1(12+i)=t(1);
                y2_1(19+i)=t(2);
                y2_1(26+i)=t(3);
        end
        %the dct3_7 block diagonal matrix
        y2_2(12:18)=X_C3_a(y2_1(12:18), 7, 2/9);
        y3(12:18)=dct3_7(y2_2(12:18), x(1), N, first);
        y2_2(19:25)=X_C3_a(y2_1(19:25), 7, 4/9);
```

```
        y3(19:25)=dct3_7(y2_2(19:25), x(1), N, first);
        y2_2(26:32)=X_C3_a(y2_1(26:32), 7, 8/9);
        y3(26:32)=dct3_7(y2_2(26:32), x(1), N, first);
        %doing the permutations
        %K_21_7
        y4(12:32)=K_n_m_a(y3(12:32), 21, 7);
        %Q_11_3
        y4(1:11)=Q_a(y3(1:11), 11, 3);
        %Q_32_10
        y=(Q_a(y4, 32, 10))';
end
```

Listing 3.8. Puschel algorithm

Some of the steps used for this algorithm given in listing 3.8 should be commented. The input vector is **x**, passed as argument of the function, since it will be defined externally to the algorithm. In the function body, the vector that is been used is **x_norm** which has the first element equal to 0, as explained before for the normalization of the algorithm. The first element of the input vector, x(1)(since the indexes in Matlab start from 1), will be used only inside the DCT5_4 blocks to do the extra sum. So every time the Matlab function ndct5_4 or dct3_7 is called, the value of x(1) should be passed as argument.
When doing the algorithm for $DCT3_21$ as given by the equation 2.7, there is a Kronecker product to be computed between $DCT3_3(\frac{2}{3})$ and I_7. If $DCT3_3$ is defined in a general way as:

$$DCT3_3 = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

given an input vector:

$$X = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_21 \end{bmatrix}$$

Then the product between the input vector and the Kronecker product, will provide a result of the type:

$$Y = \begin{bmatrix} a_{1,1} \cdot X_1 + a_{1,2} \cdot X_8 + a_{1,3} \cdot X_{15} \\ a_{1,1} \cdot X_2 + a_{1,2} \cdot X_9 + a_{1,3} \cdot X_{16} \\ \vdots \\ a_{1,1} \cdot X_7 + a_{1,2} \cdot X_{14} + a_{1,3} \cdot X_{21} \\ a_{2,1} \cdot X_1 + a_{2,2} \cdot X_8 + a_{2,3} \cdot X_{15} \\ \vdots \\ a_{3,1} \cdot X_1 + a_{3,2} \cdot X_8 + a_{3,3} \cdot X_{15} \end{bmatrix}$$

So the result can be found equivalently by forming from the input vector of 21 points, 7 smaller vectors of 3 points each, and apply to this input vectors the $DCT3_3(\frac{2}{3})$. In the Matlab function puschel_32, listing 3.8, the loop in row 22 is used to do exactly this.
The other parts of the algorithm are straightforward. At the end it will return the value of the **y** output vector of 32 elements.

3.2 Verification

After writing the algorithm in Matlab, the next step will be to verify its correctness. For this purpose the comparison between the algorithm and the $DCT5_32$ matrix definition has to be done. **ntrue_dct5** is the Matlab function for the DCT5, following exactly the definition of DCT5 given by the equation 3.4. The code for this function is as follows:

```
function C_real=ntrue_dct5(N_matrix)
        C_real=zeros(N_matrix, N_matrix);
        for m = 1: N_matrix
                for n = 1: N_matrix
                        if m == 1
                                km = 1/sqrt(2);
                        else
                                km = 1;
                        end
                        if n == 1
                                kn = 1/sqrt(2);
                        else
                                kn = 1;
                        end
                        C_real(m,n) = 2/sqrt(2*N_matrix-1)*km*kn
   *cos((m-1)*(n-1)*2*pi/(2*N_matrix-1));
                end
```

```
17          end
18  end
```

Listing 3.9. DCT5 definition

What is done in the verification of the normalized code 3.8, the two matrices to be compared are multiplied with an input vector of 32 points, containing only the i-th element as 1 and all the other elements are zero. By multiplying this vector with the matrix, the result will be the i-th column of the matrix. What the code does is to compute the difference between the two obtained column vectors. After executing the code, all the column vectors were equal, with difference approximately 0. The actual results are shown in figure:

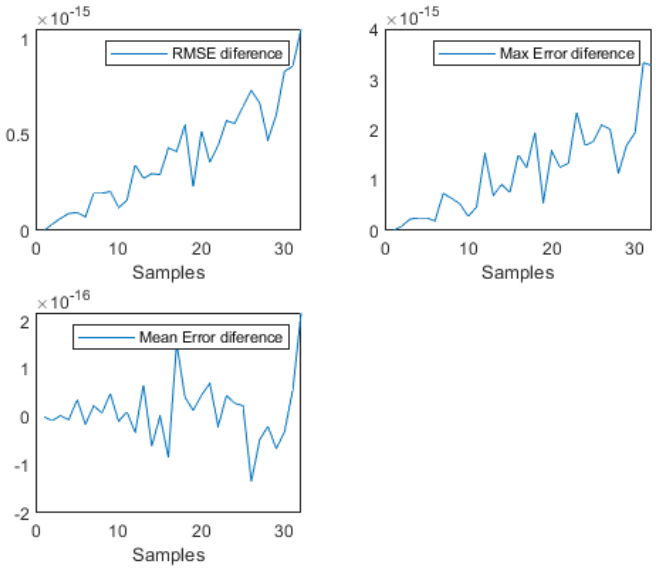

Figure 3.2. normalized algorithm error

As it can be seen from the graphs of root mean square, max error and mean error, they are of the order of 10^{-15}, so it is negligible. It can be said that the normalized algorithm is correct.

3.3 Integer coefficients

Till now an exact algorithm was considered using Matlab. Meaning that the coefficients used for the fast algorithm models were all in floating point. Making reference again to the JEM software and its definition of DCT5 matrix 3.1, after defining a

double **v** as the generic element of the matrix, what is finally stored is actually a *short*, which is **v** represented as integer. To do this, in 3.1 it multiplies by $s = \sqrt{c} \cdot 2^8$ and then does the rounding up, by truncating the fractional part of the number. For the two cases of interest $DCT5_4$ and $DCT5_{32}$, **c** will take respectively the values 4 and 32, meaning that **s** will be 2^9 for the first case and $2^{10.5}$ for the second. By doing this shift, allows for the normalization of the coefficients into integers, but also changes the value of the final results. So for having the same results expected by the software 3.1, it is important to normalize also the coefficients of the algorithm found, 2.13, trying to have as a final shift respectively 2^9 and $2^{10.5}$ for the $DCT5_4$ and $DCT5_{32}$.

The concept applied for the normalization of the coefficients is similar to what was used for the normalization of the DCT5. That is by multiplying the coefficients by a constant, it changes their value, and this will propagate into changing the value of the results. This means that again by doing a shift, the same as done in 3.1, for the coefficients of $DCT5_4$ and $DCT3_7$, will give the correct value of the final results. But some further considerations have to be done about for the introduced algorithm 3.1, since there are several multiplication stages preceding $DCT5_4$ and $DCT3_7$(in the schematic they are nominated as C5_4 and C3_7 respectively). Stages like $X_n^{(C3)}$, $DCT3_3$ also perform multiplications inside with floating point coefficients. Meaning that also this coefficients need to be normalized. Since every coefficient normalization is done by multiplying by a power of two(also can be seen as shift in the later stages of the implementation), each of these stages increase even further the order of magnitude of the results since what is done is none other than cascaded multiplications. To have the correct order of magnitude of the results, it is important to shift back, or truncate, the final computation.
During the work for this thesis, several versions of Matlab code were written, each of them did the truncation in different positions of the algorithm. In the first version, all the stages preceding $DCT5_4$ and $DCT3_7$ were truncated immediately after they did the multiplication. Since later on, when doing the implementation, the order of magnitude of the results will also define the number of bits used, then this solution will save more from the HW point of view, since it gives a minimum parallelism. The code listed below in listing 3.10, tries to explain the immediate truncation:

```
function y=X_C3_a(x, N, r, nb)
y(1)=x(1);
for i=2:N
        c=cos((0.5-r)*(i-1)*pi/N);
        sign_c=sign(c);
        c_mod=abs(c);
        c=sign_c*floor(c_mod*power(2,nb-1)+0.5);
```

```
8
9           s=sin((0.5-r)*(N-(i-1))*pi/N);
10          sign_s=sign(s);
11          s_mod=abs(s);
12          s=sign_s*floor(s_mod*power(2,nb-1)+0.5);
13
14          y_partial=c*x(i)+s*x(N-i+2);
15          y(i)=floor(y_partial/power(2,nb-1));
16          %the result has been shifted back immediately
17  end
18  end
```

Listing 3.10. min parallelism X

Starting from the code for $X_n^{(C3)}$, now a new input argument is added to the function, the number of bits nb. This nr is related with the normalization factor used for the coefficients. As shown, the normalization factor will be 2^{nb-1}. In the last line of code, after doing the computation of the output, it is immediately truncated. So the $X_n^{(C3)}$ block does not introduce any additional factor that multiplies the final result. The same concept is applied for every block which has floating point multiplications.
In the case of $B_{3m+2}^{(C5)}$ there is a minimal variation with respect to the original code, since to use only integer numbers the floor operator has to be applied after doing the division by 2(the floor operator truncates the fractional part of the number).
Finally the change happening for $DCT5_4$ is shown below:

```
1
2   for i=1:7
3           sign_c=sign(C(i));
4           c_mod=abs(C(i));
5           C(i)=sign_c*floor(c_mod*power(2,nb1-1)+0.5);
6   end
7
8   ...
9
10  %partial outputs
11  if first==1
12          y_0=a7_0*C(7);
13          y(1)=floor(y_0/power(2,nb-1));
14  else
15          y(1)=a7_0;
16  end
```

Listing 3.11. min parallelism $DCT5_4$

The normalization of the coefficients is done in the same way. Another thing to be noted is the fact that the extra multiplication done only for the first output element has to be truncated back. This is because this multiplication is in cascade with the other multiplications, meaning that if not truncated back, then the result will be wrong(not the same with the software 3.1 result). Also for $DST7_3^T$ is done in a similar way for the normalization of the coefficients. It has to be noted that doing the normalization in this way no new multiplications have to be introduced, simply the value of the coefficients is changing. The other Matlab functions which do not introduce multiplications with floating point numbers remain the same.
Now to verify this new version of Matlab code again it is used the same method as before of comparing the columns of the matrices. The **true_dct5** matrix used as comparison must have integer coefficients, meaning that the previous code 3.9 will change as follows:

```
function C_real=ntrue_dct5(N_matrix, nb1)
C_real=zeros(N_matrix, N_matrix);
for m = 1: N_matrix
        for n = 1: N_matrix
                if m == 1
                        km = 1/sqrt(2);
                else
                        km = 1;
                end
                if n == 1
                        kn = 1/sqrt(2);
                else
                        kn = 1;
                end

                K_real = 2/sqrt(2*N_matrix-1)*km*kn*cos((m-1)*(n
    -1)*2*pi/(2*N_matrix-1));
                sign_cf= sign(K_real);
                X_mod= abs(K_real);
                C_real(m,n) = sign_cf*floor(X_mod*power(2, nb1
    -1)+0.5);
        end
end
end
```

Listing 3.12. true DCT5 normalized

The result of the comparison is given in figure 3.3, which clearly shows that the two are different.

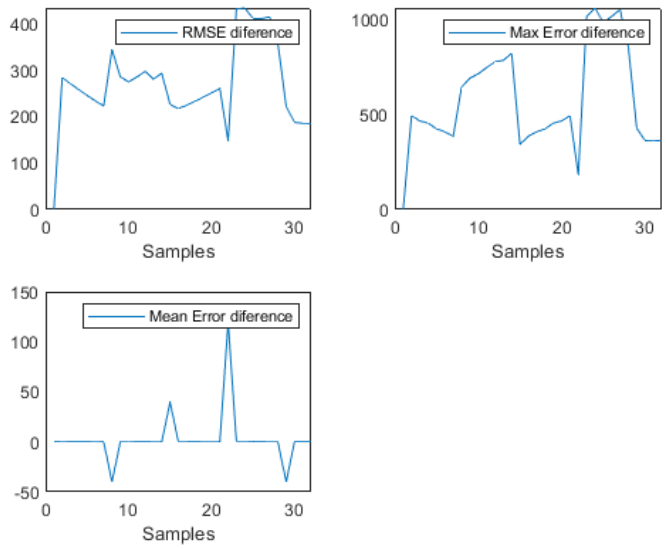

Figure 3.3. minimum parallelism error

Doing the truncation immediately after the operation gives an error, this due to the fact that the algorithm works with small values, meaning that when truncated back, those small values are approximated with one or zero, that is the reason of the big values for the error. What reinforces this fact even more is that, looking at the difference between the columns there are 4 elements which are equal. Tracing back the position of those elements, they actually correspond to the first $DCT5_4$ which is not preceded by any of the blocks which introduce multiplications. This is why no error is seen for these cases.
To move on, another version of the algorithm is tried, this time doing the truncation at the end. This means that it is worked with the full parallelism and only at the end it is rescaled back to the correct result. The following code should be added to the $DCT5_4$ function:

```
if (first==1)&&(first_dct3==1)
        m6=C(6)*4*x_0;
```

```
end

if (first==0)&&(first_dct3==1)
        m6=C(6)*power(2, (nb2+1))*x_0;
end
if (first==0)&&(first_dct3==0)
        m6=C(6)*power(2, (2*(nb2-1)+nb3))*x_0;
end
```

Listing 3.13. full parallelism DCT5_4 code

The fact is that x_0 is the only input of $DCT5_4$ which is introduced separately, without passing from the cascaded matrices introducing the normalization multiplications. Meaning that while the other inputs have already had an increase in order of magnitude, to x_0 this should be done explicitly as shown above in listing 3.13. The single $DCT5_4$ is being preceded by two **B** matrices introducing a multiplication by 2 each, that is why also a factor 4 is included in the multiplication. In the first $DCT3_7$ case a power of 2 with nb2+1 is included, since this block is preceded by the same two **B** matrices of before and also an **X** block which introduces a multiplication by 2^{nb2-1}. The last case corresponds to the three remaining $DCT3_7$ blocks which are preceded by one **B** matrix and also by X_3, $DCT3_3$ and X_7. This last 3 blocks will introduce respectively multiplications of 2^{nb2-1}, 2^{nb3-1} and 2^{nb2-1}. This explains the additional coefficient used.
It should be noticed, that in this case of full parallelism a new modification is done to the code. To generalize even more, three different normalization constants have been considered in the three different positions where the multiplications are being normalized. **nb1** is used for the normalization of $DCT5_4$ and $DCT3_7$, **nb2** for $X_n^{(C3)}$ and **nb3** for $DCT3_3$. Finally, since in software 3.1 the final result will have only the multiplication with 2^{nb1-1}, it means that a renormalization should be done to remove the other multiplicative constants. To do this, at the top level function, $DCT3_7$ block, the renormalization code should be added. The rows of code doing this are given in the following:

```
...
if first_dct3==0
y(1)=floor(a1/power(2, 2*(nb2-1)+nb3));
y(2)=floor(a2/power(2, 2*(nb2-1)+nb3));
y(3)=floor(a3/power(2, 2*(nb2-1)+nb3));
y(4)=floor(a4/power(2, 2*(nb2-1)+nb3));
y(5)=floor(a5/power(2, 2*(nb2-1)+nb3));
y(6)=floor(a6/power(2, 2*(nb2-1)+nb3));
y(7)=floor(a7/power(2, 2*(nb2-1)+nb3));
```

```
else
y(1)=floor(a1/power(2, (nb2+1)));
y(2)=floor(a2/power(2, (nb2+1)));
y(3)=floor(a3/power(2, (nb2+1)));
y(4)=floor(a4/power(2, (nb2+1)));
y(5)=floor(a5/power(2, (nb2+1)));
y(6)=floor(a6/power(2, (nb2+1)));
y(7)=floor(a7/power(2, (nb2+1)));
end
```

Listing 3.14. renormalization code in DCT3_7

The last renormalization to be considered is the multiplication by 4 for the first $DCT5_4$, which is done as follows:

```
...
y3(1:4)=full_ndct5_4(y2(1:4), x(1), N, nb1, nb2, nb3, first,
    first_dct3);
y3(1:4)=floor(y3(1:4)./4);
```

Listing 3.15. renormalization code in the top level block

Now considering the correctness of this code, the same comparison between the **true_dct5** and the full parallelism algorithm has been considered. The error between the columns of the matrices is small, as shown in the figure below:

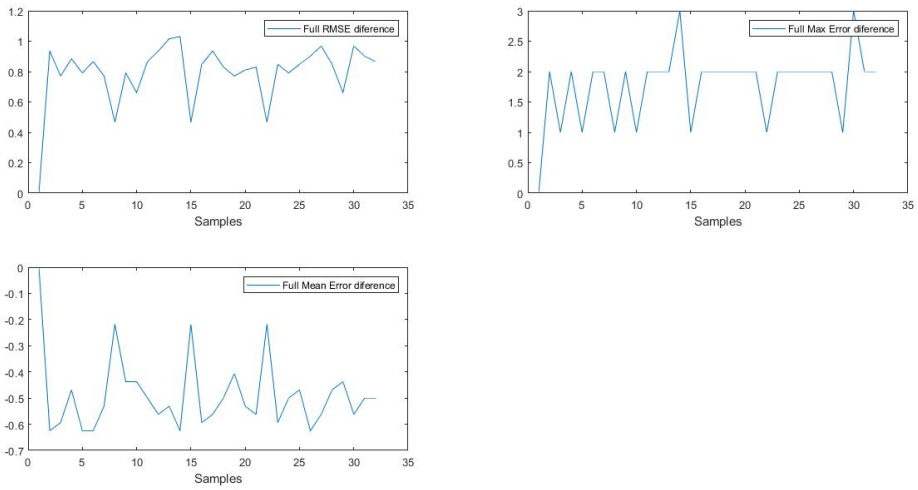

Figure 3.4. error of full parallelism algorithm

At max there are two or three samples which differ by 3 units, otherwise the difference is 1 or 2 units. This shows that the full parallelism algorithm is the best to approximate the $DCT5_{32}$ used in JEM software 3.1. The problem would be to implement this algorithm due to the very high parallelism. To compute the parallelism a Matlab script is written, which while doing the algorithm will find the maximum partial result for every addition or multiplication. In this way knowing the maximum result from each operation, the maximum number of bits representing that result can be found. To do this study of the parallelism, as a first step a variation of the initial Matlab algorithm should be done. For every created Matlab function, instead of having only the output vector as the return variable, an addition and multiplication vector should be returned also. This corresponds to the array of results of the additions and multiplications inside that block. To better understand this last part, a piece of the code is given below:

```
function [y, a, m]=para_dct3_3(x, nb3)

a(1)=x(1)-x(3);

c(1)=-sqrt(3)/2;
c(2)=1.5;

for i=1:2
        sign_c=sign(c(i));
        c_mod=abs(c(i));
        c(i)=sign_c*floor(c_mod*power(2,nb3-1)+0.5);
end
m(1)=c(1)*x(2);
m1=floor(m(1)/power(2, nb3-1));

m(2)=c(2)*x(3);
m2=floor(m(2)/power(2, nb3-1));

a(2)=a(1)+m2;
a(3)=a(2)-m1;
a(4)=a(2)+m1;

y(1)=a(3);
y(2)=a(1);
y(3)=a(4);
end
```

Listing 3.16. parallelism computation DCT3_3

While the code that does the computation of the parallelism for the adders and multipliers is given below:

```
nb1=12;
nb2=11;
nb3=11;
N=32;

for i=1:500
        x(:, i) = randi ([-2^8,2^8-1], N,1);
end
for i=1:length(x(1, :))
        [y(:, i), a(i, :), m(i, :)]=para_puschel_32(x(:, i), nb1
    , nb2, nb3);
end
for i=1:length(a(1, :))
        max_add(i)=ceil(log2(max(abs(a(:, i)))));
end
for j=1:length(m(1, :))
        max_mul(j)=ceil(log2(max(abs(m(:, j)))));
end
```

Listing 3.17. parallelism computation code

The output of this code will be the two vectors, **max_add** and **max_mul** containing the parallelism for all the additions and multiplications in the algorithm. Using the algorithm described above which does the truncation of the results only at the end, many of the operations would have parallelism even higher than 50, which makes it very costly from the architecture point of view.
The last solution tried, which makes also the final code for the algorithm, will be midway between the first and second solutions proposed above. Which is to not do the truncation immediately after the block, but to let it propagate to the next block, do the multiplication and then do the truncation immediately after. It is a trade off of not having the max parallelism with still an acceptable error. In this way the first blocks which have multiplications to normalize, X_7 preceding the first $DCT3_7$ and all the X_3, will not do the division by the normalization coefficients, meaning that the result will not be truncated. The result will be truncated in the following blocks. The code listed below tries to explain this concept:

```
function y=work_X_C3_a(x, N, r, nb2, first_dct3)

        if (first_dct3==0)&&(N==7)
```

```
                y(1)=x(1);
        else
                y(1)=(power(2, nb2-1)-1)*x(1);
        end

...

        if (first_dct3==0)&&(N==7)
                m1=floor(m1/power(2, nb2-1));
                m2=floor(m2/power(2, nb2-1));
        end
```

Listing 3.18. final X_n modification

This Matlab function 3.18, corresponds both to X_3 and X_7 blocks. The truncation has to be done only for the X_7 blocks, except the first one. In the algorithm, the X_7 blocks are preceded by the X_3 blocks, so the truncation done, that division with 2^{nb2-1} is due to the multiplication with 2^{nb2-1} done in the X_3 block. Another block to be considered is $DCT3_3$. The added code is shown below:

```
m1=c(1)*x(2);
m1=floor(m1/power(2, nb3-1));

m2=c(2)*x(3);
m2=floor(m2/power(2, nb3-1));
```

Listing 3.19. DCT3_3 final modification

In this case the truncation is done immediately with the same amount 2^{nb3-1}, since the inputs are coming from X_3 block so they are already multiplied with 2^{nb2-1}. This means that the output of $DCT3_3$ will not be totally truncated so it will not introduce a big error. Finally the last truncation should be done in the $DCT5_4$ and $DCT3_7$ blocks. The part of the code doing that is:

```
if N==32
        if first==0
                if first_dct3==1
                        m1=floor(m1/power(2, nb2+1));
                        m2=floor(m2/power(2, nb2+1));
                        m3=floor(m3/power(2, nb2+1));
                        m4=floor(m4/power(2, nb2+1));
                        m5=floor(m5/power(2, nb2+1));
                else
```

```
                m1=floor(m1/power(2, nb2));
                m2=floor(m2/power(2, nb2));
                m3=floor(m3/power(2, nb2));
                m4=floor(m4/power(2, nb2));
                m5=floor(m5/power(2, nb2));
            end
        else
            m1=floor(m1/4);
            m2=floor(m2/4);
            m3=floor(m3/4);
            m4=floor(m4/4);
            m5=floor(m5/4);
        end
end
```

Listing 3.20. truncation inside DCT5_4

Again what is left to be truncated is the multiplication introduced by the previous stage only. In the single $DCT5_4$ case, what is preceding it are the two B matrices, so a division by 4 is necessary. The $DCT3_7$ are all preceded by an X_7 block, what is changing is that for the first $DCT3_7$ there are the B_32 and B_11 matrices introducing a multiplication by 2, while for the remaining $DCT3_7$ there is only B_32. The multiplication done is respectively with 2^{nb2+1} and 2^{nb2}.
The final step will be to verify this code comparing as always with the columns of the DCT matrix used in the JEM software. In this case what is obtained is:

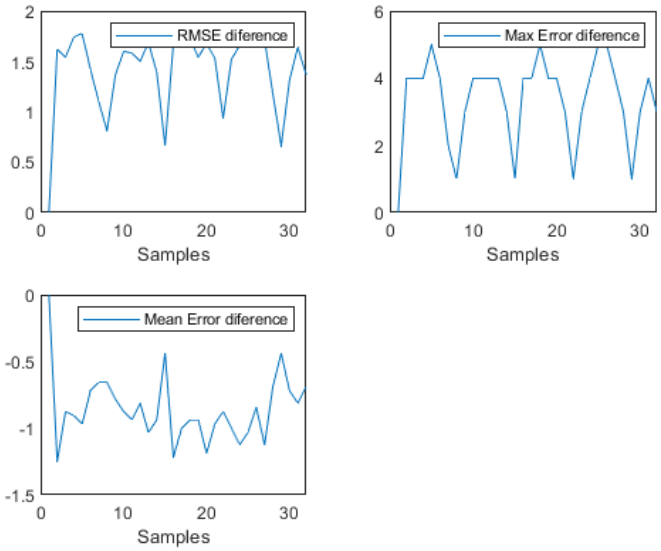

Figure 3.5. final algorithm error

The difference is larger than the one for the full parallelism, but still it remains acceptable, having a max of 5 units. And obviously having to truncate and not leaving the full parallelism, reduces the architecture cost. This is why the last solution is the one chosen for implementation.
Finally a random test is done. Instead of multiplying with unit vectors, random input vectors with values in the range (-256, 255) are used. The resulting statistics in this case are shown below for both $DCT5_32$ and $DCT5_4$:

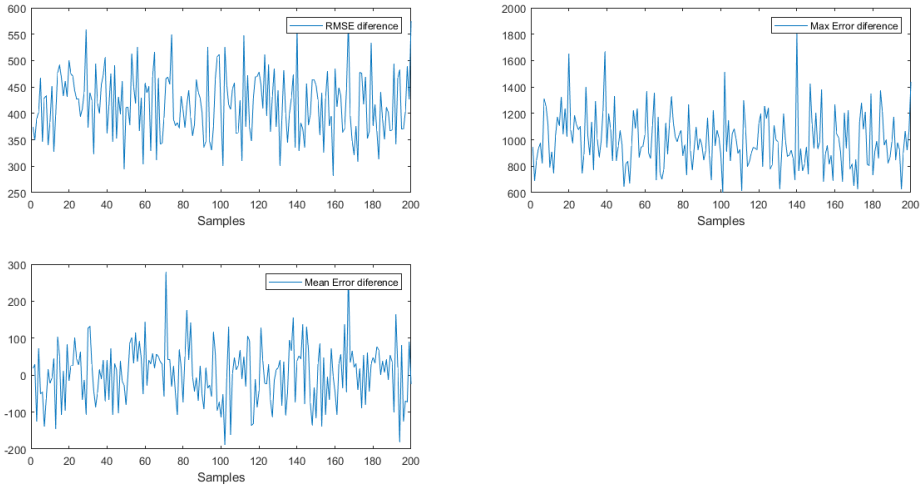

Figure 3.6. $DCT5_32$ tested with random vectors

The same Matlab script which is doing the statistics in figure 3.6, will also create two text files. One will contain the input samples which are randomly generated and in the other file will be stored the results of the algorithm. The same thing will be done for the $DCT5_4$ case and the results are shown in figure 3.7:

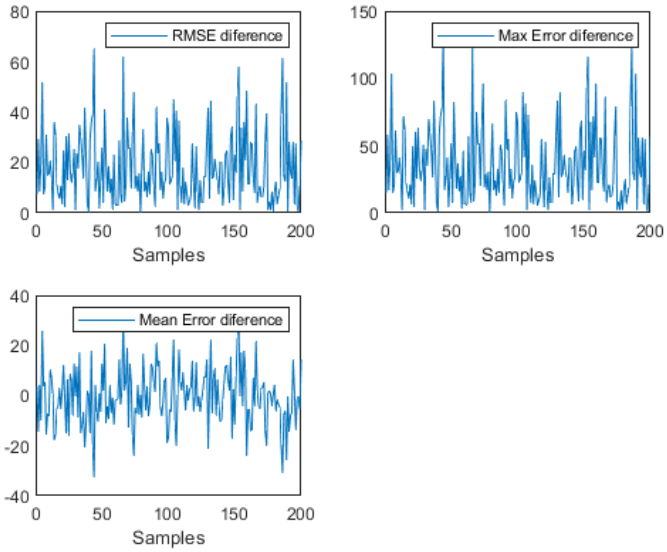

Figure 3.7. $DCT5_4$ tested with random vectors

3.4 C code

In the previous section a Matlab model was implemented. To have a reference model used to develop, debug and test the algorithm as an hardware architecture, it is necessary to develop a C model. This will be a fixed point implementation. In developing this model, the same structure is maintained. Each of the blocks are written as C functions, then the final algorithm will be a series of function calls. The passage from Matlab to C is pretty much straight forward, what changes is just the variable declaration. In C the variables used should be declared first, meaning that a preliminary study of the parallelism should be done to know if certain intermediate results should be declared as int type or long long int. From this study there are several multiplications which have results with more than 32 bits. This multiplications should be kept in consideration when writing the C code, since it can not be stored into int variables(or long int which is stored also in 32 bits), but they should be stored in long long int variables. The writing of the code later on is straightforward.
The **main** in the C code will open two files, one in read mode and one in write mode. The input file is the samples file from Matlab. Every row of the file is been read and for the vector samples the reference algorithm 2.13 has been computed. The output is written in the output file, which is later compared with the output file of the Matlab algorithm. In the $DCT5_{32}$ mode input and output file will have rows of 32 integers each, while in the other mode of operation the input and output files will have rows of 4 integers each. The format of this files has been made similar to the format of the Matlab files created by the Matlab code discussed in the previous section. Actually the input file is the same samples file generated by Matlab. While for the output text file it will be compared with the Matlab output text file by a Matlab scrip. This Matlab script simply does the difference element by element of the two files to see if they are equal. After running the simulation and comparing the two text files together, the results are as shown by the figure 3.8. It clearly shows that the two files are equal, meaning the C code is a correct representation of the Matlab algorithm.

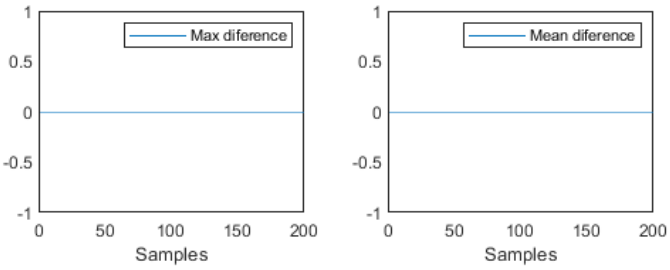

Figure 3.8. difference between C and Matlab

In a similar way it is done the simulation for $DCT5_4$. Also in this case the C code was correct.
After verifying the correctness of the C code, the next step is to include it in the source code of the JEM software and do the simulation. The JEM software is based on the HEVC Model(HM) software. It is a C++ implementation of both the encoder and the decoder. It allows to evaluate the weight of every single coding tool on the overall rate-distortion performance given by the standard. The drawback is the high computational effort. To encode a video sequence, two configuration files and some additional parameters should be specified. The first configuration file gives information on the coding tools to be used, while the second gives the information about the input video sequence file. A document is defined by the Joint Video team, named Common Test Conditions (CTC), which defines the common environment where the experiments must be conducted. It specifies to use four QP values, namely 22, 27, 32 and 37. The test sequences used cover a large range of resolutions from $480 \times 240up$ to 4096×2160.
The quality and the bit-rate are the metrics characterizing the coding efficiency of a video encoder. Peak Signal to Noise Ratio(PSNR) is an index from the video samples used to approximate the quality. The quality and the bit-rate data are used to plot the rate-distortion curve of an encoder. Therefore, by plotting both rate-distortion curves of the reference and the test encoders, it can be easily observed which of the two performs better. The rate-distortion curves are plotted by interpolating the four points resulting from the encoder simulation using the four QP values. After each

simulation the resulting PSNR value and the corresponding bit rate are obtained. As mentioned previously two configuration files are necessary for the encoding, more specifically the first configuration file used is *encoder_intra_jvet10.cfg* while the second one depends on the input video sequence file. The choice for the input video sequence is of type C, that is a resolution of 832x480. The first choice is a sequence with a frame rate of 30Hz named *Race Horses*, while the second one is a sequence with a frame rate of 50Hz named *Basketball Drill*. The corresponding configuration files used in the two cases are respectively *RaceHorsesC.cfg* and *BasketballDrill.cfg*. Finally the results for the reference and test encoder are shown. For the test encoder, the part of the source code corresponding to the DCT5 will be substituted with the developed C model. This C model introduces a new algorithm for the $DCT5_4$ and $DCT5_{32}$ matrices. Different simulations are done for different versions of the source code for the encoder. One version includes only the new algorithm for the $DCT5_{32}$, one version only the algorithm for the $DCT5_4$ and the last one is the final modification which will include both $DCT5_4$ and $DCT5_{32}$. Below in figure 3.9 is shown the rate distortion curve comparing the reference and test encoder including only the new algorithm for the $DCT5_{32}$. A more explicit definition of the four points, corresponding to the four different QP values, is given in table 3.2. It has to be noted that for these simulations the input video sequence is *Basketball Drill*:

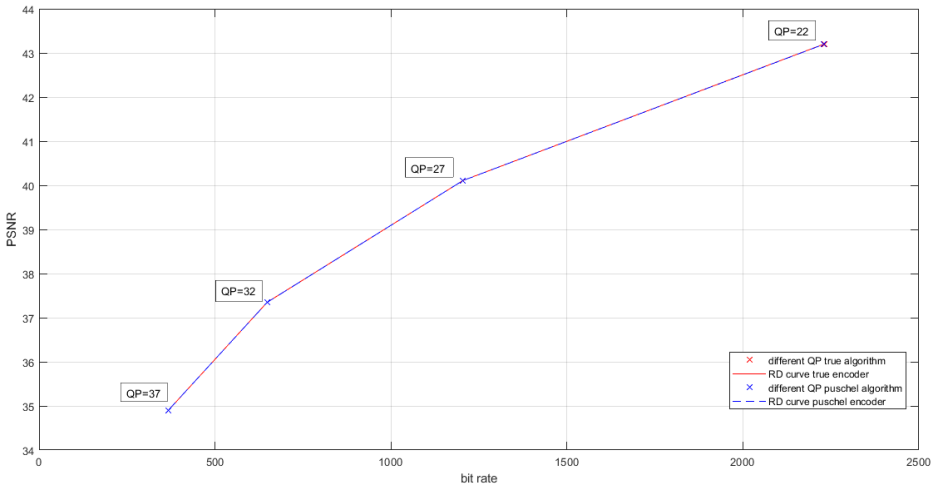

Figure 3.9. Rate distortion curve for reference and test encoder

Similar results, with a minimal difference with respect to the previous ones are obtained simulating the test encoder containing the new code for the $DCT5_4$ only. The four points for the reference encoder remain the same of table 3.1, what are changing are only the four points of the test encoder. In this case these points are

QP	37	32	27	22
bit rate	366.9190	648.5651	1204.6175	2230.7944
PSNR	34.8960	37.3512	40.1083	43.2010

Table 3.1. The four points for the reference encoder. The input sequence used is Basketball Drill

QP	37	32	27	22
bit rate	367.3897	648.4381	1205.6222	2231.1230
PSNR	34.8970	37.3513	40.1118	43.2011

Table 3.2. The four points for the test encoder, with the new algorithm for $DCT5_{32}$ only

given in table 3.3. The rate distortion curve in this case is the same with the curve of figure 3.9.

QP	37	32	27	22
bit rate	367.4556	648.7460	1204.0825	2229.6794
PSNR	34.9024	37.3504	40.1060	43.1943

Table 3.3. The four points for the test encoder, with the new algorithm for $DCT5_4$ only

Finally the four points defining the rate distortion curve are given for the final test encoder, which will contain the new code for both $DCT5_4$ and $DCT5_{32}$. Again the results do not change much from the previous two cases, as given in table 3.4

QP	37	32	27	22
bit rate	367.3286	648.4571	1203.9437	2230.4222
PSNR	34.8951	37.3479	40.1042	43.1963

Table 3.4. The four points for the test encoder, with the complete final algorithm

Just for statistics, another simulation is done using a different input test sequence. The test sequence used in this case is again of C type, named *Horse Race*. The resulting rate distortion curves are given in figure 3.10. For more details, the four point are given in table 3.5 and table 3.6.

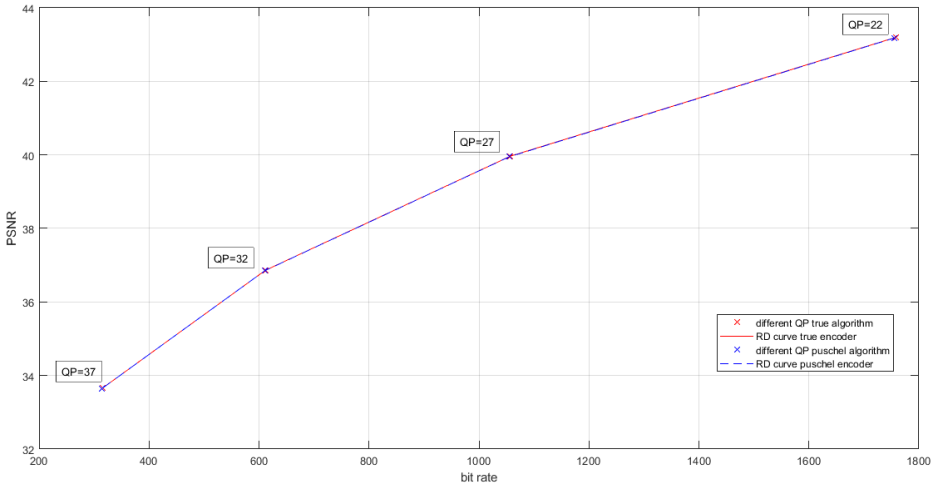

Figure 3.10. Rate distortion curve for reference and test encoder. Input sequence is named *Horse Race*

QP	37	32	27	22
bit rate	315.3521	611.4884	1056.7232	1758.3734
PSNR	33.6566	36.8599	39.9631	43.1979

Table 3.5. The four points for the reference encoder. The input sequence used is *Horse Race*

QP	37	32	27	22
bit rate	314.6266	610.3918	1055.3534	1756.1100
PSNR	33.6480	36.8481	39.9496	43.1818

Table 3.6. The four points for the test encoder. The input sequence used is *Horse Race*

Chapter 4

Developing the architecture

4.1 VHDL code

The final step in this thesis work is to create an architecture for the algorithm found, and try to see how this architecture behaves in terms of timing, area and power consumption. What has been tried to be implemented is an architecture which may work in two possible ways. If selected the $DCT5_{32}$ mode, given an input vector of 32 samples, it will compute the $DCT5_{32}$ and give an output vector of 32 samples. Each of the input samples are integers represented in **nb_in** bits complement of 2. The other mode to be selected is the $DCT5_4$ mode. To exploit what is given by the algorithm, i.e. the 5 blocks of $DCT5_4$ inside the $DCT5_{32}$, then an input of 20 samples is given, 4 samples for every $DCT5_4$. In this last case to introduce the inputs to the architecture the first 20 of the overall 32 are used. Also the outputs are given in the first 20 pins out of 32.
Another important point in the development of the architecture is the parallelism used for the coefficients. This number of bits is given as a parameter. So the architecture is parametric, with the number of bits used for the coefficients of the different stages given as generic constants. So when writing the architecture, considerations were done step by step about the parallelism, and how it increased with every block. The figure below tries to give a better picture of the architecture:

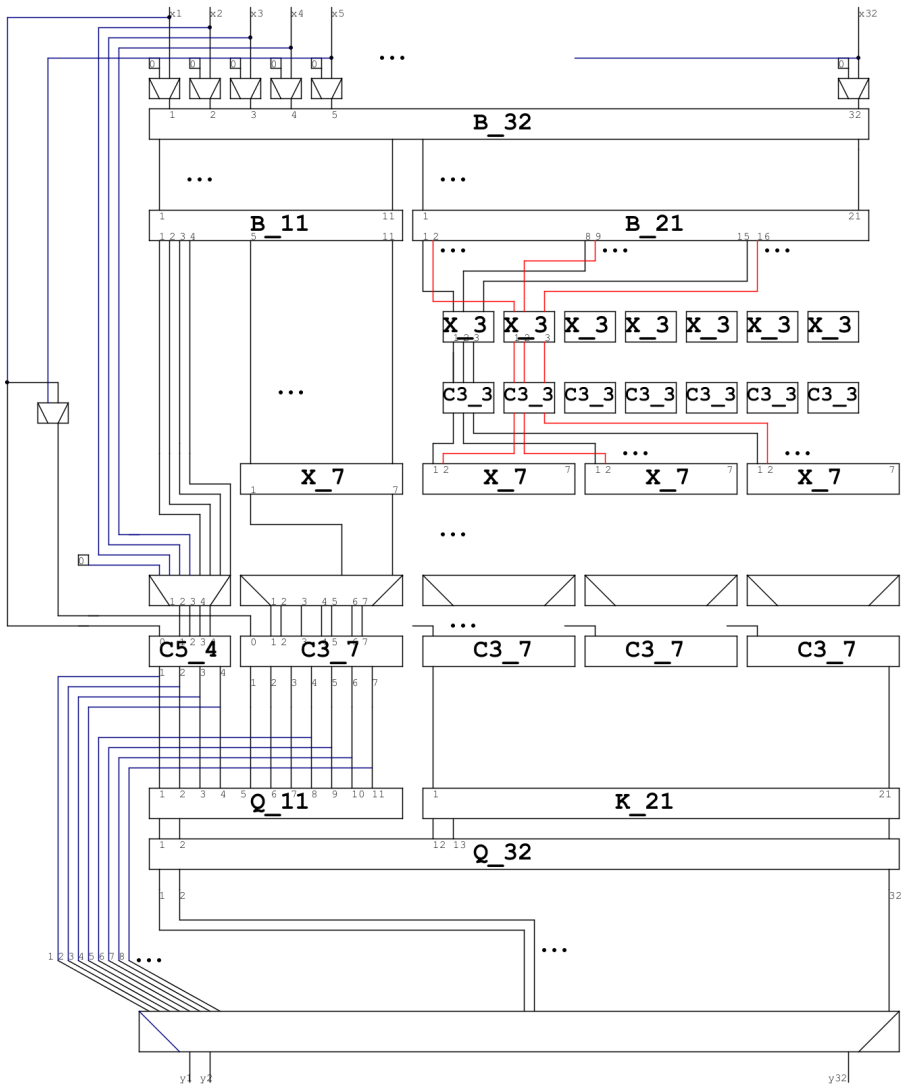

Figure 4.1. Top architecture

Starting with the blocks one by one, from input to output: The first entity to be considered is param_B_32 corresponding to matrix $B_{32}^{(C5)}$. The block is combinatorial and it can be written directly from the C or Matlab code, substituting for the sum and subtraction operation, the actual HW components.

There are defined 32 input ports, **nb_in** bits each. The input number of bits is given as a parameter. To write everything in a more concise way a vector notation was tried for the 32 inputs, creating a type array of **std_logic_vector** unconstrained. This type had to be used in every entity definition, so it was defined in a package. Using this kind of construct, an unconstrained array of unconstrained arrays, is possible only in VHDL 2008. No problems occurred when simulating the model,

but when doing the synthesis, *Synopsys Design Compiler* did not support that kind of construct. That is why the less concise notation is used to define every port separately. About the output ports, looking at their notation, they are been divided in two groups. This is done keeping in consideration the following stages of the algorithm(the block diagonal matrix with blocks of 11 and 21 points).

Moving on to the description of the components used, an adder component was defined to do the addition of three numbers. In the code this is given by **adder_2**. The adder_2 component does two normal additions(given by **param_adder** components). This was done just to write everything in a more compact way. It is used for computing the first 11 outputs of **B_32**, since they need 3 additions. The thing to be noticed about this block is the number of output bits. For every addition one extra bit has to be used to not have overflow. Since at maximum two additions are done for every output, it is expected that the output parallelism is nb_in+2. Looking at the port definition, actually the z output ports are defined with nb_in+3 bits. The extra bit is due to the left shift done for every output. Returning back to the Matlab code, when doing the simulation of the algorithm, the floor operation when doing the division by 2, introduced a wrong final result. This was because when small numbers were used, the floor would truncate everything to 0. To resolve the problem, the division by 2 should be removed. To do that everything is to be left shifted by 1(meaning a multiplication by 2). For the final result of the algorithm it is important to right shift the result by 1 to get something correct. For the last 21 outputs the method is straightforward, the only difference is that in this case a max of 1 subtraction is done, giving a parallelism of nb_in+2. The code for B_11 is written in a similar way. Still the algorithm of reference is the same $B_{3m+2}^{(C5)}$, what is changing is only the number of input and output ports. Again also in this case the output is divided in two. A group of 4 outputs and another of 7 with respectively a parallelism of nb+3 and nb+2.

An important difference between B_11 and B_32 is that B_11 includes also an adder to compute the first output. The algorithm defined an addition between x(1) and x(2m+2), but it was not included in B_32 since in this case x(1)=0 always(because of the normalized algorithm). In the case of B_11 x(1) is not zero.

```
--first output
y11(1)(0)<='0';
y11(1)(nb_in downto 1)<=x(22);
y11(1)(nb_in+2 downto nb_in+1)<=(nb_in+2 downto nb_in+1=>x(22)(
    nb_in-1));

--uses x(1) in the computation
x22_ext<=std_logic_vector(resize(x(22), nb_in+1));
y21(1)<=std_logic_vector(resize(-signed(x22_ext), nb_in+2)) ;
```

Listing 4.1. first output computation B_32

For y11(1), since x(1) is zero, the output will be just $2 \cdot x(22)$. The first row of the code shown in listing 4.1 does exactly that, a multiplication by 2 or a left shift by 1. Then the third row of the code does just the bit extension. For the y21(1), the output will be just x(22) with the changed sign. But an extra step is needed to eliminate the risk of overflow, so an extension of the parallelism by one bit is done. On the other hand, for B_11, x(1) is not zero any more, meaning that instead of a simple change of sign, an addition or a subtraction is needed, as shown from the code below:

```
a1: param_adder
generic map(N=>nb+1)
port map(A=>x(1), B=>x(8), S=>y4(1)(nb+1 downto 1));
y4(1)(0)<='0';
y4(1)(nb+2)<=y4(1)(nb+1);

---...

x_8<=std_logic_vector(resize(signed(x(8)), nb+1));
left_shift(0)<='0';
left_shift(nb downto 1)<=x(1);

a12: param_subtractor
generic map(N=>nb+2)
port map(A=>left_shift, B=>x_8, S=>y7(1));
```

Listing 4.2. first output computation B_11

B_21 code corresponds to the $B_{3,m}^{(C3)}$ algorithm. Transforming from C to vhdl becomes straightforward. Again also for this code, a new component is created, **param_sub_add**, grouping together a subtractor and an adder.

After defining the code for the pre-addition matrices the components containing the fast algorithms can be written. As said in the chapter 2, about the Matlab code, for the first stages of $X_n^{(C3)}$ no truncation has to be done. When writing the architecture two blocks have been defined. X_3 corresponds to $X_n^{(C3)}$ of 3 samples. It will be the block preceding $DCT3_3$. Meanwhile X_7 is the one for 7 samples, so it will precede the $DCT3_7$ blocks. Since the X_3 block is always the first block in the cascade then its output should not be truncated.
The actual parallelism used inside the block for the coefficients is nb2+1. One extra bit is added due to the fact that if considered the values of those coefficients there

are few of them with values near to 1. When shifted with nb2-1 an overflow may happen in this case giving an erroneous result. This is why it is chosen to extend the parallelism. As said before no truncation has been applied to the output. For the case of X_7 some extra considerations have to be done, since the same block will be used when both it needs truncation and when it does not need truncation.

```
1
2 ...
3
4 y(1)<=std_logic_vector(resize(signed(x(1)), nb_out+1))when
      first_dct3='0' else
5 std_logic_vector(to_signed((2**(nb2-1)-1)*to_integer(signed(x(1)
      )), nb_out+1));
6
7 ...
8
9 c_int<=std_logic_vector(to_signed(integer(sign(c)*floor((sign(c)
      *c)*real(2**(nb2-1))+0.5)), nb2+1));
10
11 ...
12
13 m_c<=std_logic_vector(resize(shift_right(signed(prod_c), (nb2-1)
      ), nb_out)) when first_dct3='0' else
14 std_logic_vector(resize(signed(prod_c), nb_out));
```

Listing 4.3. X_7

The difference in this code is when doing the renormalization of the coefficients. It shifts right with nb2-1 only in the case when it is not the first $DCT3_7$. In this case the signal before elaborated by the X_7 block should pass through the X_3 block described previously. Since this last block did not do the truncation at the end, that truncation is actually done at this point in the algorithm. So this block will be instantiated 4 times in the architecture. Once it will be instantiated as a block doing the truncation(this when first_dct3=1) and three other times will be instantiated as a block not doing the truncation. Since the block is the same, then a parameter should be defined for the output parallelism. In this way the output parallelism is decided when instantiating the component. This nb_out parameter it is not strictly necessary for X_3, but it is used just to have a similarity of the code.

In the case of $DCT3_3$, since the input vector has already had a left shift by the preceding block, and this block itself will introduce another left shift, then in this case it is possible to right shift the result. It is chosen to do the right shift with nb3-1, which is the same shift introduced when transforming the coefficients

of $DCT3_3$ into integers. It should be noted that the first coefficient is stored in nb3 bits while the second in nb3+1 since this coefficient is larger than one, and so using nb3 bits would have introduced an overflow. The code below does the truncation.

```
m1<=std_logic_vector(resize(signed(mul_ext1(nb+nb3-1 downto nb3
    -1)), nb+3));

m2<=mul_ext2(nb+nb3 downto nb3-1);
```

Listing 4.4. DCT3_3 truncation

What remains to be defined now are the last blocks in the cascade, that is the blocks containing $DCT5_4$. $DCT5_4$ is the block used when working both in 32 samples mode and in 20 samples mode. The two sets of coefficients have been defined, the one for $DCT5_4$ standalone and the other one for when it is used inside $DCT5_{32}$. An initial process is used to select between the two sets of coefficients. **sel_dct**=0 selects the first case while **sel_dct**=1 selects the other case with coefficients of $DCT5_{32}$. In this last case to normalize the coefficients, as in the software 3.1, a multiplication with $2^{10.5}$ should be used. But for the normalized algorithm, the multiplication with C(6) will be in cascade with the other multiplications meaning that it introduces an extra shift of the coefficient, i.e. a shift of the result. To obtain the correct value then it should be renormalized, by dividing again with the constant $2^{10.5}$. Since doing a floating point division is heavy from the computational point of view, then only for the last coefficient the normalization constant used is 2^{11} instead of $2^{10.5}$. In this way when trying to rescale back the result at the end by dividing with the normalization constant, it can be simply done by a right shift. This last operation is given by the code:

```
last_mul: process(sel_dct, m6_ext, a13)
begin
        if sel_dct='0' then
                y1<=std_logic_vector(resize(signed(m6_ext(nb_out
   +nb1_32 downto nb1_4-1)), nb_out+4));
        else
                if first='1' then
                        y1<=std_logic_vector(resize(signed(
   m6_ext(nb_out+nb1_32 downto nb1_32-1)), nb_out+4));
                else
                        y1<=std_logic_vector(resize(signed(a13),
    nb_out+4));
                end if;
        end if;
end process last_mul;
```

Listing 4.5. DCT5_4 first output resizing

The two cases have been separated, when computing $DCT5_{32}$ a different normalization constant is used, meaning that this same normalization constant will be used to renormalize it back. nb1_4 and nb1_32 are defined in the software 3.1 and have respectively the values 9 and 12. A separate case is selected when it is not considered the first $DCT5_4$ of the algorithm, meaning **first**=0. Since the normalization coefficient $\frac{1}{\sqrt{2}}$ is not used when **first**=0, as a result also when doing the renormalization it should not be divided by that constant. Another important piece of code for this block is the part which does the renormalization from the cascaded multiplications. $DCT5_4$ is the block used before almost every output element. It is preceded by different matrices. The first $DCT5_4$ is preceded only by the two B matrices. Each of these matrices does a multiplication by 2. In order to have correct results for the outputs of $DCT5_4$ it is important to divide with the same amount that has been multiplied. So for the first $DCT5_4$ the constant to be divided is 4, or in another way it can be said that it should be right shifted by 2. In the case of first $DCT3_7$, the block X_7 is preceding it. Since this block will introduce a left shift by nb2-1, then it is important that the $DCT5_4$ block does the right shift of the same amount, to bring the result at the correct order of magnitude. It should be noted also that in this case there are also the two B matrices preceding everything. This means that also in this case a right shift by 2 has to be done. So in overall, the total right shift amounts to nb2+1. For the last 3 $DCT5_4$, again the only block preceding them that has to be considered is X_7. The multiplication with the normalization constants introduced by the blocks X_3 and dct3_3 have already been taken care of. What is changing in this case is the fact that instead of having two B matrices preceding, there is only one B_32 which introduces the multiplications by 2, while B_C3 does not introduce normalization multiplications. So for the case of the final 3 $DCT5_4$ the right shift to be done is by nb2. The part of code which does all what has been explained here is written below in listing 4.6:

```
1 gen_not_first: if first='0' generate
2         gen_first_dct3: if first_dct3='1' generate
3                 m_norm(0)<=std_logic_vector(resize( shift_right(
    signed(mul_ext0),(nb2+1)), nb_out));
4                 m_norm(1)<=std_logic_vector(resize( shift_right(
    signed(mul_ext1),(nb2+1)), nb_out));
5                 m_norm(2)<=std_logic_vector(resize( shift_right(
    signed(mul_ext2),(nb2+1)), nb_out));
```

```
6                   m_norm(3)<=std_logic_vector(resize( shift_right(
    signed(mul_ext3),(nb2+1)), nb_out));
7                   m_norm(4)<=std_logic_vector(resize( shift_right(
    signed(mul_ext4),(nb2+1)), nb_out));
8           end generate gen_first_dct3;
9           gen_other_dct3: if first_dct3='0' generate
10                  m_norm(0)<=std_logic_vector(resize( shift_right(
    signed(mul_ext0), nb2), nb_out));
11                  m_norm(1)<=std_logic_vector(resize( shift_right(
    signed(mul_ext1), nb2), nb_out));
12                  m_norm(2)<=std_logic_vector(resize( shift_right(
    signed(mul_ext2), nb2), nb_out));
13                  m_norm(3)<=std_logic_vector(resize( shift_right(
    signed(mul_ext3), nb2), nb_out));
14                  m_norm(4)<=std_logic_vector(resize( shift_right(
    signed(mul_ext4), nb2), nb_out));
15          end generate gen_other_dct3;
16  end generate gen_not_first;
17  gen_first: if first='1' generate
18          m_norm(0)<=std_logic_vector(resize( shift_right(signed(
    mul_ext0), 2), nb_out));
19          m_norm(1)<=std_logic_vector(resize( shift_right(signed(
    mul_ext1), 2), nb_out));
20          m_norm(2)<=std_logic_vector(resize( shift_right(signed(
    mul_ext2), 2), nb_out));
21          m_norm(3)<=std_logic_vector(resize( shift_right(signed(
    mul_ext3), 2), nb_out));
22          m_norm(4)<=std_logic_vector(resize( shift_right(signed(
    mul_ext4), 2), nb_out));
23  end generate gen_first;
```

Listing 4.6. right shift to obtain the correct results

Finally for the second mode of operation, when only $DCT5_4$ are being computed, this problem does not exist since there are no preceding blocks which have coefficients to be normalized. So in this case the result has simply to be extended into the same number of bits as for the $DCT5_{32}$ case. After having explained all the used components, the top entity called **param_datapath** has simply a structural architecture doing the instantiation of all this components. For this architecture there are two modes of operation, but still it will use only the blocks of the $DCT5_{32}$ algorithm. Which means some multiplexers are needed to supply the inputs to the $DCT5_4$ and $DCT3_7$ blocks. Basically the top level architecture will have 32 inputs each of nb0 bits for the input samples, and 32 outputs with nb0+nb1_32+21. Since

this is a parametric architecture, the number of bits for the coefficients it is not known a priori, so the worst case has always to be considered when computing the parallelism of every block. This explains the large number obtained as output parallelism. The final input is **sel_dct** which is of binary type, used as the select signal when doing the multiplexing. When **sel_dct**=1 the architecture is expected to work as $DCT5_{32}$, so it expects 32 inputs and will calculate 32 outputs. While for the other mode of operation, it needs only 20 inputs and will calculate 20 outputs. The inputs are expected to be given in the first 20 consecutive ports, the other 12 inputs will be ignored. Actually to reduce the power consumption during this mode of operation, since only the 5 $DCT5_4$ will be used, the inputs to the $DCT5_{32}$ can be supplied to 0 in this case, since the blocks preceding the 5 $DCT5_4$ will not be used. Some considerations have to be done for the inputs of $DCT3_7$. The first stage of the $DCT3_7$ is the **Q** matrix from [4.9] factorization. It will do a permutation of the inputs, and afterwards the **D** matrix introduced by [4.8] does also a sign change. Keeping this in mind the assignment to the outputs of this block is given as:

```
1 y7_dct1_4 <=( 3=>std_logic_vector(resize(-signed(x(6)), nb0+nb2
    +7)),
2                              5=>std_logic_vector(resize(signed(x(7))
    , nb0+nb2+7)),
3                              7=>std_logic_vector(resize(-signed(x(8)
    ), nb0+nb2+7)),
4                              others=>(others=>'0'));
```

Listing 4.7. dct input assignment

The code shown in listing 4.7 corresponds to the first $DCT3_7$, for the other $DCT3_7$ blocks the input parallelism will change accordingly. Also for the outputs of the architecture a multiplexing operation has to be done. When **sel_dct**=1 then the final permutation matrices have been implemented by doing the corresponding signal assignments for every output port. When **sel_dct**=0 no permutations have to be done, the outputs of $DCT5_4$ and $DCT3_7$ will be connected to the first 20 outputs of the architecture. The final 12 outputs will be left as high impedance. This brings an end to the description of the architecture written for the block computing both the $DCT5_{32}$ and 5 $DCT5_4$. Finally input and output registers are added to the datapath to create the top level entity. This registers will help finding the frequency of the circuit. The next section will show the simulation and synthesis results for the given architecture.

4.2 verification and synthesis

To simulate the design, a testbench is created, containing three other extra blocks:

- A clock generator component which will contain inside a process for generating the clock, and it also forces the values for **RSTn** and **sel_dct** signals.

- A data maker component which will read the input samples from a text file, and generate the outputs as *std_logic_vector*. It will also generate an **END_SIM** signal 10 clock cycles after the end of file is reached.

- A data sink component which will write the outputs of the algorithm into an output file, with the same format of the input file.

Two pair of this blocks have been created. Each one of them will simulate the architecture in one of the two operating modes. In the $DCT5_{32}$ mode input and output file will have rows of 32 integers each, while in the other mode of operation the input and output files will have rows of 4 integers each. The format of this files has been made similar to the format of the Matlab files created by the Matlab code discussed in the previous chapter. Actually the input file is the same *samples* file generated by Matlab. While for the output text file it will be compared with the Matlab output text file by a Matlab scrip. This Matlab scripts simply does the difference element by element of the two files to see if they are equal. After running the simulation and comparing the two text files together, the results are as shown by the figure 4.2. It clearly shows that the two files are equal, meaning the VHDL architecture is a correct representation of the Matlab algorithm.

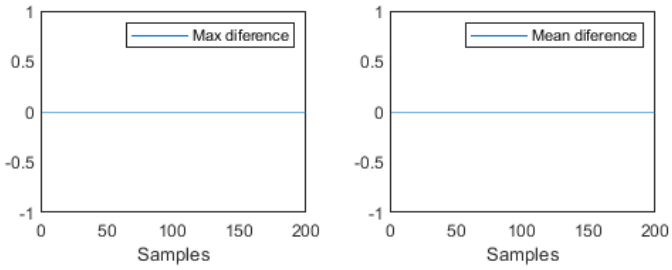

Figure 4.2. Matlab vs Vhdl difference mode 32

In a similar way it is done the simulation for $DCT5_4$. Also in this case the VHDL was correct as shown in figure 4.3:

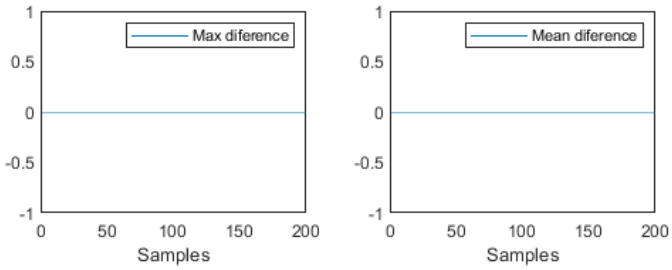

Figure 4.3. Matlab vs Vhdl difference mode 4

After verifying the correctness of the VHDL, the architecture can now be synthesized with the **Synopsys Design Compiler**. The used library for the synthesis is **umc65**. In a first stage the design has been compiled to obtain the time and area reports. The minimum working period found is **5.6ns**, which means a working frequency of 178.57MHz. The results about the area are reported in the tables below:

Combinational area	255112.202867
Noncombinational area	14879.880079
Total cell area	269992.082946

Table 4.1. Total area results

	% total
B_32	2.4
B_11	0.6
B_21	1.5
X_3(7 blocks)	6.3
dct3_3(7 blocks)	8.9
dct5_4	3.2
first_X_7	2.0
first_dct3_7	8.1
dct5	5.2
inv_dst7	1.6
X_7_2_9	5.9
second_dct3_7	15.2
dct5	9.8
inv_dst7	3.2
X_7_4_9	4.0
third_dct3_7	12.4
dct5	7.6
inv_dst7	2.9
X_7_8_9	5.7
fourth_dct3_7	15.2
dct5	9.7
inv_dst7	3.2

Table 4.2. Separate blocks area results

Looking at the results, it can be noticed that last blocks in the algorithm, the ones which would have also a higher parallelism, will also occupy more area. Also

module	implementation	count	estimated area	% of cell area
DW01_add	pparch	165	62428.9733	23.1
DW01_sub	pparch	140	47037.4714	17.4
DW_mult_tc	apparch	38	13132.4291	4.9
DW_mult_tc	pparch	111	95388.0269	35.3
Total:		454	217986.9007	80.7

Table 4.3. Different implementation area results

the multipliers which have a large parallelism will occupy in an estimated way the same area with the additions and substractions even though their number is 1/3 of the later. To obtain the power report, the design has to be compiled using the switching activity of the nodes. So as a primary step the **saif** file has to be created. The results after the simulation are shown below. There are two considered set of results, one obtained using the *compile* command of **Synopsys**, and the other using the *compile_ultra*. With the *compile* command, the minimum period achieved was **7.56ns**. The power report in this case gave the results:

internal power	23.072 uW
switching power	15.442 uW
leakage power	46.112 uW
total power	84.626 uW

Table 4.4. Compile power results 4 mode

internal power	184.8792 uW
switching power	143.2767 uW
leakage power	45.7650 uW
total power	373.9 uW

Table 4.5. Compile power results 32 mode

internal power	4.0440 uW
switching power	1.8444 uW
leakage power	26.7426 uW
total power	32.631 uW

Table 4.6. Compile ultra power results 4 mode

internal power	3.4335 uW
switching power	1.2054 uW
leakage power	26.9203 uW
total power	31.559 uW

Table 4.7. Compile ultra power results 32 mode

Comparing the values for the two cases, it can be clearly seen that *compile_ultra* does a high optimization of the design by reducing the power almost 10 times. This reduction happens even though the frequency of the circuit in this case is still higher. The *compile_ultra* does a reduction of the total dynamic power of the cell of almost 100 times in the $DCT5_{32}$ case. The dominant power component in this case though is the leakage power, which is still reduced compared to compile command result, but not with the same order as the previous one. Using the synthesis obtained with *compile_ultra*, the power consumption of the architecture during the two modes of operation is approximately 32 uW.

Bibliography

[4.1] K. Sayood, *Introduction to Data Compression*. Morgan Kaufmann, 2012.

[4.2] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1649–1668, Dec 2012.

[4.3] T. Biatek, V. Lorcy, P. Castel, and P. Philippe, "Low-complexity adaptive multiple transforms for post-HEVC video coding," in *Picture Coding Symposium*, 2016, pp. 1–5.

[4.4] X. Zhao, J. Chen, M. Karczewicz, X. Li, and C. Wei-Jung, "Enhanced Multiple Transform for Video Coding," in *Proc. 2016 Data Compression Conference*, 2016, pp. 73–82.

[4.5] M. Puschel and J. M. F. Moura, "Algebraic Signal Processing Theory: Cooley-Tukey Type Algorithms for DCTs and DSTs," *IEEE Trans. Signal Process.*, vol. 56, no. 4, pp. 1502–1521, April 2008.

[4.6] Joint Collaborative Team on Video Coding (JCT-VC) of ITU-T SG16 WP3 and ISO/IEC JTC1/SC29/WG11, *HM-16.6-JEM-4.0 Reference Software Model*. [Online]. Available: https://jvet.hhi.fraunhofer.de/svn/svn_HMJEMSoftware/tags/HM-16.6-JEM-4.0/

[4.7] M. Puschel and J. M. F. Moura, "Algebraic Signal Processing Theory."

[4.8] M. Masera, M. Martina, and G. Masera, "Odd Type DCT/DST for Video Coding: Relationships and Low-Complexity Implementations," *IEEE Trans. Signal Process.*, November 2017.

[4.9] Y. A. Reznik, "Relationship between DCT-II, DCT-VI, and DST-VII transforms," in *Proc. 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 5642–5646.

[4.10] X. Shao and S. G. Johnson, "Type-II/III DCT/DST algorithms with reduced number of arithmetic operations," *Signal Processing*, vol. 88, no. 6, pp. 1553–1564, 2008.

[4.11] M. T. Heideman, "Computation of an odd-length DCT from a real-valued DFT of the same length," *IEEE Trans. Signal Process.*, vol. 40, no. 1, pp. 54–61, Jan 1992.

[4.12] S. Winograd, "On Computing the Discrete Fourier Transform," *Mathematics of computation*, vol. 32, no. 141, pp. 175–199, 1978.