Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática
2019

Miguel Oliveira Inocêncio

Co-processador da Transformada para o Codificador de Vídeo AV1

Transform Co-Processor for AV1 Video Coding Standard

Miguel Oliveira Inocêncio

Co-processador da Transformada para o Codificador de Vídeo AV1

Transform Co-Processor for AV1 Video Coding Standard

Dissertação de Mestrado apresentada à Universidade de Aveiro, para obtenção do grau de Mestre em Engenharia Eletrónica e de Telecomunicações, sob orientação científica do Professor Doutor António Navarro, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, com colaboração do Professor Luciano Agostini e do Video Technology Research Group - ViTech - da Universidade Federal de Pelotas - UFPel.

o júri / the jury

presidente / president

Prof. Dr. Armando José Formoso de Pinho
Professor Associado com Agregação da Universidade de Aveiro

vogais / examiners committee

Prof. Dr. Pedro António Amado Assunção
Professor Coordenador, Escola Superior de Tecnologia e Gestão de Leiria do Instituto Politécnico de Leiria (Arguente Principal)

Prof. Dr. António José Nunes Navarro Rodrigues
Professor Auxiliar da Universidade de Aveiro (Orientador)

agradecimentos / acknowledgements

À minha família por todo o apoio que deram ao longo destes 5 longos anos.
A todos os amigos e companheiros por terem feito parte destes 5 curtos anos.
Ao João, Edgar, Correia, Kevin, Diogo e à República de que vamos sempre fazer parte.
Ao Luís, Simão, Soares, Santos e ao segundo piso da Biblioteca, que conhecemos melhor que ninguém.
À Universidade de Aveiro, Instituto de Telecomunicações e membros docentes, por todo o conhecimento que me foi transmitido, e infraestruturas para o aplicar.
Ao Video Technology Research Group da Universidade Federal de Pelotas por todo o auxílio prestado durante esta dissertação.

Palavras-Chave Compressão de Vídeo, AV1, Transformadas, DCT, FPGA

Resumo

Esta dissertação apresenta o estudo efetuado sob o formato de compressão de vídeo *AV1*. A investigação realizada resultou em dados estatísticos referentes a diversas opções de codificação, tais como o kernel da transformada mais utilizado, os tamanhos de vetores utilizados, o número de bits utilizado nas aproximações de cossenos, entre outros. Com os resultados obtidos, foram implementadas medidas de otimização no codificador de referência, obtendo-se uma melhoria de 3% no tempo total de codificação, com uma redução de 81% na utilização de memória dedicada às aproximações do cosseno.

O algoritmo implementado em software foi de seguida descrito em VHDL, tendo sido obtidas duas soluções. A primeira permite um elevado grau de paralelização, obtendo todos os diferentes tamanhos de vetores transformados em 22 ciclos de relógio, sendo capaz de codificar vídeo FHD a 30 imagens por segundo, com uma frequência de operação de 187 MHz. A segunda minimiza a utilização de lógica, a custo de não permitir o cálculo de vários tamanhos de vetores simultaneamente. Esta última solução foi sintetizada e testada numa placa Nexys 4, ocupando 79.93% da área total da FPGA e 50 mW de potência consumida. No kit de hardware no qual foi implementada, esta arquitetura é capaz de processar vídeo HD a 30 imagens por segundo.

Keywords Video Coding, AV1, Transform Coding, DCT, FPGA

Abstract

This dissertation presents a study made of the video coding standard *AV1*. The research provides statistical results referring to various encoding options, such as the most commonly used *Transform* kernel, vector sizes, the number of bits used in cosine approximations, amongst others. With the gathered results, optimization measures were implemented on the reference encoder, achieving a 3% decrease in the total encoding time, with 81% reduction in the memory used to store cosine coefficients.

The algorithm implemented in software was then described in VHDL, obtaining two implementable architectures. The first allows a high degree of parallelization, obtaining all transformed vector sizes within 22 clock cycles, being able to maintain FHD video at 30 frames per second, at an operating frequency of 187 MHz. The second minimizes the amount of logic, although it does not allow the calculation of multiple vector sizes in parallel. This implementation was synthesized and tested on a Nexys 4 board, occupying 79.93% of total FPGA area and 50 mW consumption. On the hardware kit on which it was implemented, this architecture is able to process HD video at 30 frames per second.

Contents

List of Figures

List of Tables

Acronyms

ADST Asymetric Discrete Sine Transform

AOM Alliance for Open Media

ASIC Application Specific Integrated Circuit

AV1 AOM Video 1

CABAC Context Adaptative Binary Arithmetic Coding

CMOS Complementary metal–oxide–semiconductor

Codec Encoder-Decoder

CPU Central Processing Unit

CRT Cathode Ray Television

DCT Discrete Cosine Transform

DFT Discrete Fourier Transform

FFT Fast Fourier Transform

FPGA Field-Programmable Gate Array

fps Frames per Second

GPU Graphical Processing Unit

HEVC High Efficiency Video Coding

IC Integrated Circuit

JVT Joint Video Team

LUT Look Up Table

MCU Microcontroller

MM Matrix Multiplication

MPEG Motion Picture Experts Group

MSE Mean Square Error

PSNR Peak Signal to Noise Ratio

QP Quantization Parameter

RISC Reduced Instruction Set Computer

SDK Software Development Kit

TV Television

UART Universal asynchronous receiver/transmitter

UHD Ultra-High-Definition

VHDL Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (HDL)

VLC Variable Length Codes

VLSI Very Large Scale Integration

WGN White Gaussian Noise

WHT Walsh-Hadamard Transform

WNS Worst Negative Slack

Glossary

Bjontegaard-Delta rate (BD-rate) Objective quality metric, that evaluates the bitrate savings according to the obtained PSNR

Codec Encoder-Decoder. Also referred to the method of compressing and decompressing a video sequence

Floorplaning Synthesis and Implementation of an hardware design onto an FPGA, through the planning of logic gates and input/output attribution

Generic Parametric variable that allows for the easy parametrization of a design in VHDL

H.264/AVC Previous state of the art video codec from Joint Video Team (JVT), released in 2007. As to the writing of this work, it is the most used video compression algorithm

Interlaced scanning Technique used by televisions for broadcasting and displaying, where only odd or even numbered lines of a frame are transmitted/displayed at a time, alternately

JPEG Still image compression format, developed by the Joint Photographic Experts Group (JPEG)

libaom Reference software for AV1, released by Google in June 2018

Pixel Picture Element

Progressive scanning Technique used by more recent screens, where each frame is displayed as a whole, from top to bottom, and left to right

RGB Color space based on the addition of Red, Green and Blue components for complex color representation

VP8/VP9 Open-format video codecs developed by Google, released in 2008 and 2013, respectively

Nomenclature

$\boldsymbol{b}$ Bit

$\boldsymbol{B}$ Byte (8 bits)

$\lceil \mathbf{x} \rceil$ Round to the nearest integer, greater than x

$\lfloor \mathbf{x} \rfloor$ Round to the nearest integer, lower than x

$\mathbf{max}\,\mathbf{X}$ Largest value of matrix $\mathbf{X}$

$\mathbf{min}\,\mathbf{X}$ Smallest value of matrix $\mathbf{X}$

$\lfloor \mathbf{x} \rceil$ Round to the nearest integer

$\boldsymbol{x} << \boldsymbol{K}$ Shift x, k bits to the left

$\boldsymbol{x} >> \boldsymbol{K}$ Shift x, k bits to the right

$\vec{\boldsymbol{x}}$ Nth dimension vector

$\mathbf{X}$ Matrix

$\vec{\boldsymbol{\mathcal{X}}}$ Input vector $\vec{x}$ in the *Transform* domain

$\vec{\boldsymbol{x}}_{*}$ Restored version of vector $\vec{x}$

CHAPTER 1

Introduction

1.1 Background and Motivation

Since the spark of television research in 1887, a tremendous investment has been put into increasing the quality of images, cameras and screens that display them [1].

In the early years of mechanical Television (TV), this desire was pursued by making changes to the *Nipkow* disks [1], up to the decline of the mechanical TV, around the 1930's. The consequential rise of all-electronic TVs started with the capture of images with the same cathode tubes put into Cathode Ray Televisions (CRTs), with broadcasts of the live analog recordings, since there were no available methods of storing images, up to 1955, with the development of the open-reel magnetic tape [2].

The evolution of Complementary metal–oxide–semiconductor (CMOS) technologies, however, led to the downfall of cathode ray tubes, and to the rise of image capture to a digital sensor, that allowed better image captures and lower demands in terms of physical storage space. However, with the desire for higher fidelity video, the quantity of information captured also increased. Whether by increasing the sensor resolution, color bit depth or frame rate, the captured video sequences have increased its size throughout the years. For instance, for a video of 640×360 (now considered as a low resolution), at 30 Frames per Second (fps), considering each captured color (considering a RGB color space) is represented with 8 bits, there is approximately 166 Million bits per second (Mbps) of captured information. This means that a short 5 minute video would occupy more than 6 Giga Bytes (GB) of memory. This aspect gets more severe once higher resolutions are considered. For newer standards such as 4K Ultra-High-Definition (UHD) (3840×2160) or 8K UHD (7680×4320), under the same conditions, a ten minute video would occupy 448 GB and 1792 GB of raw data, respectively.

To further aggravate the situation, video consumption got massively adopted on the average consumer level, and continues to grow, both in the average number of watched hours by users and in the resolutions of the video, making the bandwidth dedicated to the visualization of video footage the highest between all other application. With the development of higher video sizes, increase of the average number of connected devices per user and overall market expansion through the number of consumers, this margin will continue to grow. In fact, according to *Cisco*, by 2022, up to 82% of global IP traffic will be dedicated to video [3, Trends 1 & 4].

This problem has led to the introduction of a new concept: *Video Compression* [2], which is the process of reducing the size of a video sequence, while still maintaining its playback

[1] Scanning disks used in mechanical televisions

[2] Also called *Video Coding*.

capabilities. This process is done by the Codec, which takes advantage of redundant information present on the raw data to reduce the size of the video, without heavily modifying the original picture or its quality.

The first form of video compression, Interlaced scanning, dates from 1940, and was purely analog. This solution was introduced with the intent of reducing the necessary broadcasting bandwidth for old CRTs, without decreasing the displayed fps. And even though this technique has been in use for over seventy years, it has proven to be so efficient that most TV channels today still use interlaced broadcasting.

However, analog television is now obsolete, as well as CRTs. The massive developments in Integrated Circuit (IC) fabrication led to the rise of the current digital era. Therefore, most screens (be it televisions, monitors or cellphones) use digital, Progressive scanning. As such, the use of analog compression techniques was not applicable. Accordingly, the evolution of digital video led to the development of digital compression techniques, such as the one presented in this work.

Being purely digital, these methodologies rely on computers and other processors to analyze data and apply the compression algorithms, making them very demanding processes from a computational standpoint. As expected, a high compression ratio is only obtainable by a high complexity algorithm, which also increases with the size of the video (more data leads to more analysis). Since in the early days of digital video, the used resolutions were lower than the ones used in the present days, the compression algorithms used were not very demanding. However as the pursuit for higher quality video continued, so did the necessity for better compression ratios, and therefore the computational needs also increased. Such complex softwares lead to a high power consumption from the processor executing it, making such implementations unsuitable for portable, battery limited applications, such as cellphones or laptops. Besides this huge factor, such softwares tend to be very slow, specially when a real time compression or decompression is desired.

To amend for these factors, and to increase the reachability of high quality video to as many users as possible, these applications needed to have a viable solution that did not compromise its usability. Accordingly, a new approach has been implemented on the most recent codecs. Besides the optimization of pure software compression/decompression solutions, there has been a great focus on the development of specialized hardware for such codecs. This solution could redress many of the previously presented problems, making them viable on mobile implementations, as well as other specialized appliances, since such co-processors usually present a better performance than generic CPUs. This tendency has already been verified on the implementation choices on recent smartphones [4, p. 14], as well as recent *Nvidia* Graphical Processing Unit (GPU) lineups [5].

Due to the differences between a certain compression algorithm and its predecessors, either through the changes to the bitstream or functioning principles, each time a new codec is released, there is a need to backup its development with a new set of hardware implementations. This makes the improvement of video compression techniques a continuous effort, in many engineering branches, as the technology needs to keep up with the demands of consumers, in a variety of applications.

Due to the broad access to video, and its influence in a variety of markets (besides video consumption itself), big companies have made investments on the improvement of video quality, and respective compression algorithms. These investments have provoked somewhat of a *"Codec War"*. Since 2010, several video compression algorithms have been deployed, and quickly replaced by a newer version, which presents better compression gains, at a lower

quality degradation, such as the replace of *VP8* (released in 2008) with *VP9* (2013).

1.2 Scope

AOM Video 1 (AV1) is the most recently released[3] video codec. It was developed as a Joint Development Foundation [6] project, under the name of Alliance for Open Media (AOM) [4]. This codec took the same objective as its main predecessor, *VP9*, which was to be an open source, royalty free alternative to Motion Picture Experts Group (MPEG)'s state of the art video codec, *High Efficiency Video Coding (HEVC)*.

Figure 1.1: *AV1* logo [7].

Upon release, *VP9* rivaled *HEVC's* performance. However, soon after, the market demanded higher compression performances, giving origin to the consortium of enterprises that now represent AOM, and to the development of *AV1*, in 2015. The first release of this coding format was made in March 2018, with the first release of its reference software, *libaom*, being made three months later, in June 2018.

Besides its main objectives, *AV1* was also developed with the intent of being easily implementable in hardware. Therefore, various design choices were made to make the algorithm low memory consuming, and highly parallelizable.

The desired compression performance was obtained at the cost of a highly complex algorithm (and reference software), that severely outperforms *VP9*, at the cost of much higher compression times [8].

Taken these factors, there is a high demand for dedicated hardware architectures, that can speed up the compression/decompression times and reach real-time usability on live-streaming applications, such as video-conferencing, live-content visualization, etc.

With this work, it is intended to perform a general study of the released software, and try to improve its performance. Due to the overall complexity of the topic, the focus relies on one of its composing blocks, the *Transformation Stage*. This is expected to be achieved through the simplification of the provided software, as well as the development of fast hardware architectures.

[3] Currently, there are other codecs being developed, without official bitstream release
[4] Further explained in Chapter 2

1.3 Outline

This dissertation is divided into five distinct Chapters. The current gives the reader a general overview of the video coding environment, as well as presenting this work's objectives.

Chapter 2 starts by conferring the basic principles behind the compression of video, through the explanation of the characteristics exploited in encoders, as well as the coding tools implemented in both these and in decoders. Towards the end of the chapter, *AV1* is described in higher detail.

In Chapter 3 the theoretical basis for the focus of this work is discussed, i.e., the *Transform Coding*. The final sections of this Chapter present the functioning behind *AV1*'s *Transform* stage, as well as data referring to encoding options.

With the foundations presented in the previous Chapters, in the fourth there are presented the software and hardware implementations of the designed *Transform* stages, as well as corresponding results.

This work finishes with Chapter 5, where some final considerations are carried out, as well as suggestions for future work.

References

[1] Mark Schubin. "What Sparked Video Research in 1877? The Overlooked Role of the Siemens Artificial Eye [Scanning Our Past]". *Proceedings of the IEEE* 105.3 (Mar. 2017), 568–576. ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.2017.2652998.

[2] Marco Jacobs and Jonah Probell. "A Brief History of Video Coding".

[3] *Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper*. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html.

[4] Scientiamobile. *Mobile Overview Report April – June 2018*.

[5] *Video Encode and Decode GPU Support Matrix*. https://developer.nvidia.com/video-encode-decode-gpu-support-matrix. Nov. 2016.

[6] *Joint Development Foundation*. http://www.jointdevelopment.org/.

[7] aomedia. *Home*. https://aomedia.org/.

[8] Dan Grois, Tung Nguyen, and Detlev Marpe. "Performance Comparison of AV1, JEM, VP9, and HEVC Encoders". *Applications of Digital Image Processing XL*. Ed. by Andrew G. Tescher. San Diego, United States: SPIE, Feb. 2018, 120. ISBN: 978-1-5106-1249-5 978-1-5106-1250-1. DOI: 10.1117/12.2283428.

CHAPTER 2

Video Compression Systems

2.1 Basic Principles

Video Compression Systems have been in development for approximately forty years, with the first video codec, *H.120*, being released in 1984. It was composed of basic operations, which did not correlate to good compression performances. This has lead to a quick downfall of its usage, being aggravated by the release of the *H.261* standard by 1988.

However, the building blocks on which later standards were based are the same as in the first generations, i.e., the strategies implemented on newer standards exploit the same *redundancies* as previous, less efficient, codecs.

By redundancies, it is meant disposable information to the playback of an image sequence. This concept is the key of video compression. Throughout the years, the enhancement of video codecs was based on the improvement of the algorithms which can reliably represent a video, while maintaining the least of the original information. In other words, the video sequence is analyzed for predictable/identifiable characteristics (e.g. the movement of a subject or the edge of an object), identifies strategies of predicting nearby pixel values through that information and removes the disposable. This process is mentioned as *redundancy removal*.

This way, to have a better understanding of the functioning behind video codecs, the mentioned redundancies are presented, as well as its origins. Most of such are due to the way humans perceive vision, being this the first topic of this Chapter.

2.1.1 Human Visual System

Most of the compressed/decompressed video nowadays is directed to content visualization by consumers, with the exception of some network-driven image processing applications, such as automatic video surveillance. Therefore, the compression of video sequences has the intent of making changes to the original data, without serious impact to the users' perception. This process is mentioned as the removal of the *Psychovisual redundancy* [2]. Therefore, a basic understanding of the visual system can clarify many of the design choices made in video compression applications, and why their use does not present much impact on the quality of the image, while greatly reducing its memory usage.

The image perception starts in the human eye, represented in Figure 2.1. Its different constituents accomplish different tasks, from focusing, to aperture control. Although their importance to the overall functioning of the eye, the part that matters most to the focus of this work is the innermost membrane, the retina.

Once the desired image is properly focused by the lens, an inverse version of it is shone on

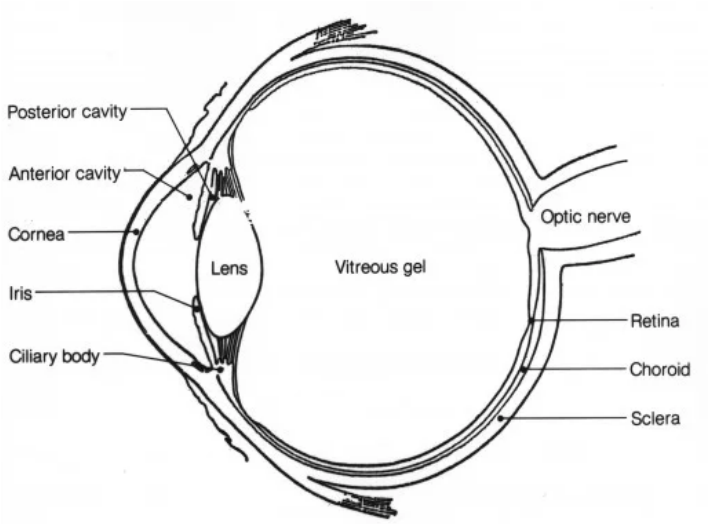


Figure 2.1: Representation of an human eye [3].

the aforementioned membrane, which is covered by two types of light sensitive cells, the *cones* and *rods*, which transform the observable image into a series of pulses, that get subsequently processed.

The *cones* are highly sensitive to color, being responsible for the *photonic* or *bright-light* vision. There are three different types, corresponding to the wavelength they are susceptible to. These are the S, M and L *cones*, being sensitive to, approximately, the blue, green and red light, respectively, making a somewhat similar capture to the RGB color system.

On the other side, *rods* are not stimulated by bright light, being more active on low illumination levels. This aspect makes them responsible for giving a rough overview of the field of view. This is called as *scotopic* or *dim-light* vision. These cells are more broadly spread across the retina comparing to the *cones*, which is also observable in the number of cells (approximately 6 million *cones*, to 100 million *rods*).

From this, it is already observable that the human visual system is more sensitive to differences on the luminosity, than to the color of an object [4], which is a starting point for compressing video, as will be shown later in this Chapter. However, many other opportunities come from the processing of the nerve signals, and the *psychovisual* perception that follows.

Although more sensitive to *luminance*, there is a threshold to which the difference between two objects — ΔI — cannot be discerned. This relation is mentioned as *contrast sensitivity function*, which is roughly approximated with the *Weber's Law*

$$\frac{\Delta I}{I} \approx constant \tag{2.1}$$

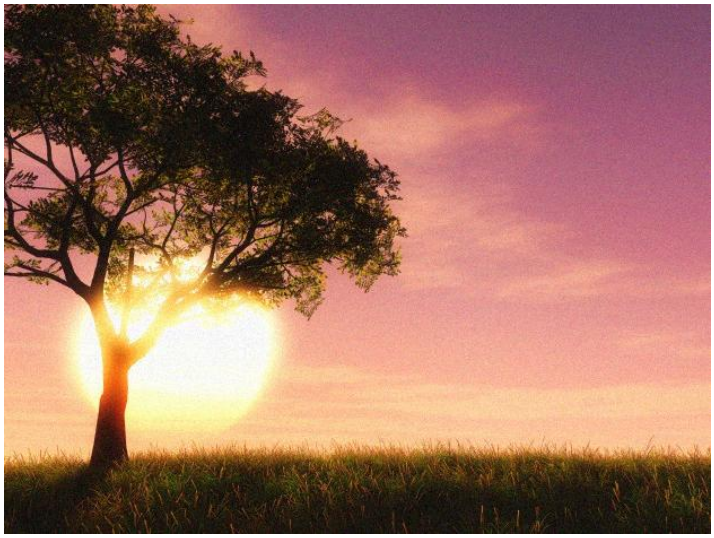
Analyzing this equation, it is possible to conclude that the darker an object is, the lower the difference in luminance needs to be to distinguish another object. Also, darker images tend to be more susceptible to compression artifacts.

Besides the luminance values, the spatial and temporal frequencies also represent an important role in the perception of such errors.

The Figure 2.2 gives an example of the dependency with spatial frequency. The first, 2.2a, represents the original image, which got corrupted with White Gaussian Noise (WGN),

represented in Figure 2.2b. As it is observable, these artifacts are less noticeable on the highly detailed areas (branches and leafs of the tree) than in the smooth ones (sky in the top right corner). The effect of *Weber's law* is also observed if we analyze the effect that the white noise as in the bright sun area, when compared to the darker areas.

(a)



(b)

Figure 2.2: Example of the effect of added noise on figure ((a) - Original Image [5]; (b) - Image with added WGN).

Temporal frequency dependency, although more challenging to exemplify, is easily understandable. On a sequence of frames with fast movements, either from the camera or the subject, the human eye does not have the ability to track details or other artifacts, while in slow moving scenes, it can easily identify errors.

These are some of the fragilities of the human visual system that get exploited during the compression of video. However, other *redundancies*, inherent from the captured images

themselves contribute to the reduction of the video size, as will be described in the following sections.

2.1.2 Redundancy Exploitation

Even though there are countless observable subjects and sceneries, it is unfair to think of a frame as a random sequence of pixels. Objects tend to represent clusters of pixels with roughly the same values, moving objects follow predictable directions, etc. Such characteristics represent *redundancies* that can be explored during the compression of said sequence.

2.1.2.1 Spatial Redundancy

Spatial redundancy comes from the similarity between neighboring pixels, on one frame. This aspect is easily verified through the autocorrelation of an image, as will be shown in the following example.

Taking Figure 2.2a and calculating its autocorrelation with various horizontal shifts, gives origin to the graph in graph in Figure 2.3.

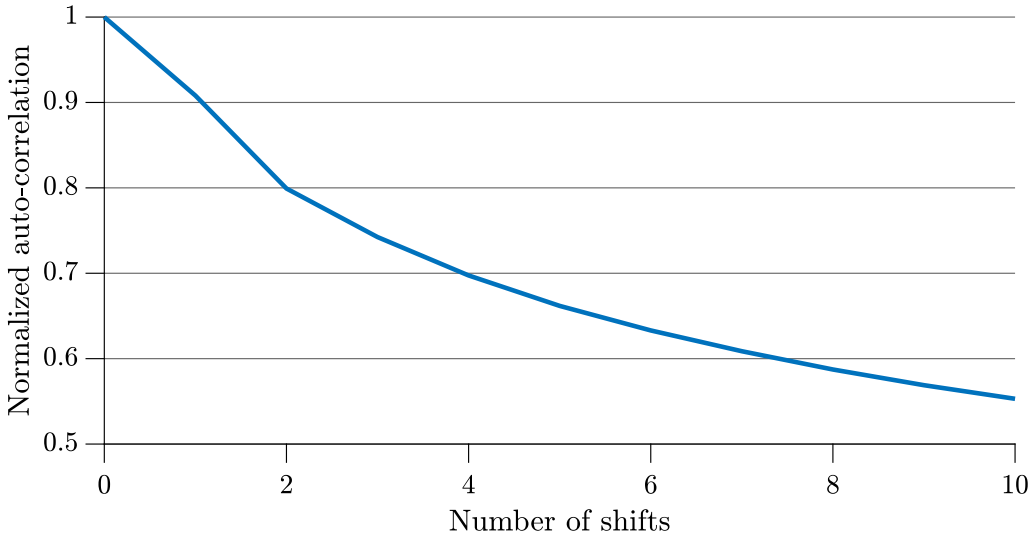


Figure 2.3: Autocorrelation of image 2.2a, with horizontal shifts.

As it is observable, for shorter shifts, the normalized autocorrelation is very close to one, since most of the de-correlation comes for mismatching edges. Although this relation varies depending on the image, it is safe to assume that it is very similar for the majority of the cases.

Such study gives a promising opportunity for compression, since it means that most pixels can be predicted from its neighbors. This aspect as lead to what is now known as *differential* or *predictive* coding.

On a video compression system, the spatial redundancy is considered in the *intra-prediction* block, which calculates pixels, or pixel blocks, through its surrounds.

2.1.2.2 Temporal Redundancy

As expected, a series of consecutive frames on the same subject, tend to be very similar between each other, especially if considered the 30 or 60fps desired nowadays.

Making a similar analysis to what was made in Section 2.1.2.1, a series of frames of the *Stefan* sequence [6] was considered, and the cross correlation between the first and the following nine was calculated, giving origin to the graph in Figure 2.4.

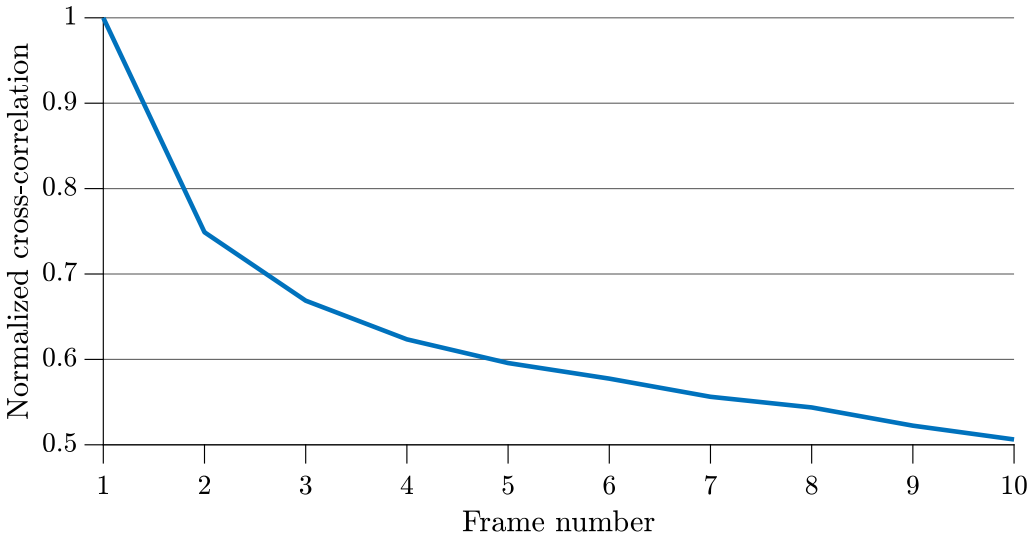


Figure 2.4: Cross-correlation between the first and following nine frames of the *Stefan* sequence.

Similarly to what happened in the previous example, the cross correlation between consecutive frames is very high. Even though for faster moving scenes this relation might not be as pronounced, its application on video coding greatly contributes to the compression verified in the latest codecs.

The codec takes advantage of this redundancy in the *inter-prediction* stage, which is composed by the *Motion Estimation* (ME) and *Motion Compensation* (MC) blocks. On this stage, blocks of pixels in nearby frames are analyzed for movement, predicting its position for following frames.

2.1.2.3 Psychovisual Redundancy

The redundancies presented in Section 2.1.1 are explored in various stages throughout the video encoder.

The first measure is the *chroma subsampling*, which takes advantage of the lower perception to color, discarding some of the *chroma* samples, depending on the subsampling chosen.

Typically, a pixel value is represented in one luminance and two chrominance values, on the *YCbCr* color space. The subsampling is defined in through the relation of luminance to chroma samples, being the most common the 4:4:4, 4:2:2 and 4:2:0 standards, represented in

Figure 2.5. In the first one, no chroma samples are discarded, which means that for each four luminance (Y) samples, there are an equal number of Cb and Cr samples. Correspondingly, in the second standard, for each four Y samples, only half of each color components are maintained. The last example, although its misleading term, means that only one in four chroma samples are kept.

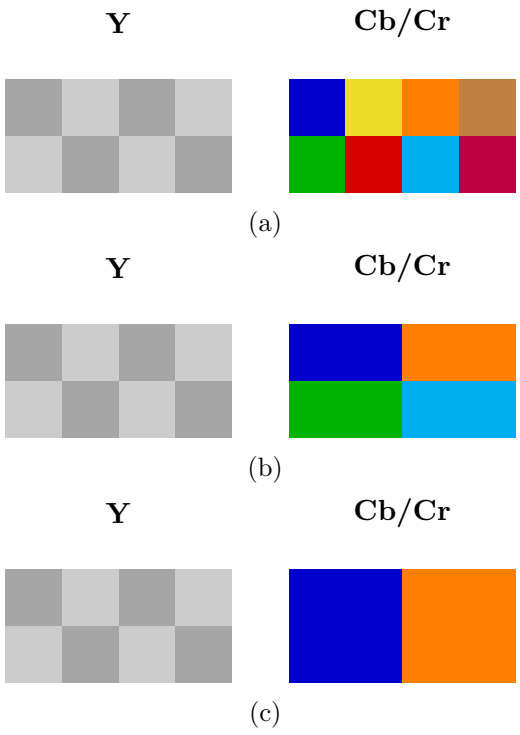


Figure 2.5: Representation of chroma subsampling ((a) - 4:4:4; (b) - 4:2:2; (c) - 4:2:0).

From the reduced sensitivity to details (or areas with high spatial frequency), the compression is explored in the *Transform* (T) and *Quantization* (Q) blocks. In the first stage, blocks of pixels are evaluated in their frequency components. These are then evaluated in the second stage, where the least significant ones get discarded. In the decoder, the image is reconstructed with the maintained coefficients, without much impact to the image quality. This process is further explained throughout the work.

On the Quantization block, some work was also developed to account for *Weber's law*, where the quantization depends on the average luminance value of the block. This concept was first introduced in *"Efficiency of a Model Human Image Code"* [7], and since then, experimented in various codecs, such as HEVC [8].

2.1.2.4 Coding Redundancy

Coding redundancy is directed to the method of representing information in the digital domain, i.e., the bits themselves, and how they are organized.

It is known that symbol probability plays a major role in information compression, across a wide variety of branches, and video is no exception. Taking this into account, codecs take

advantage of coding redundancy in the *Entropy Encoder* stage.

2.1.3 Basic Video Compression/Decompression System

From the basic principles of the previously mentioned blocks, it is possible to integrate them into two complete compression — *Encoder* — and decompression — *Decoder* — modules.

2.1.3.1 Encoder Model

The encoder's objective is to compress a video sequence, turning it into a readable *encoded bitstream*. To do this, the previously presented strategies get implemented on a system based on the schematic of Figure 2.6.

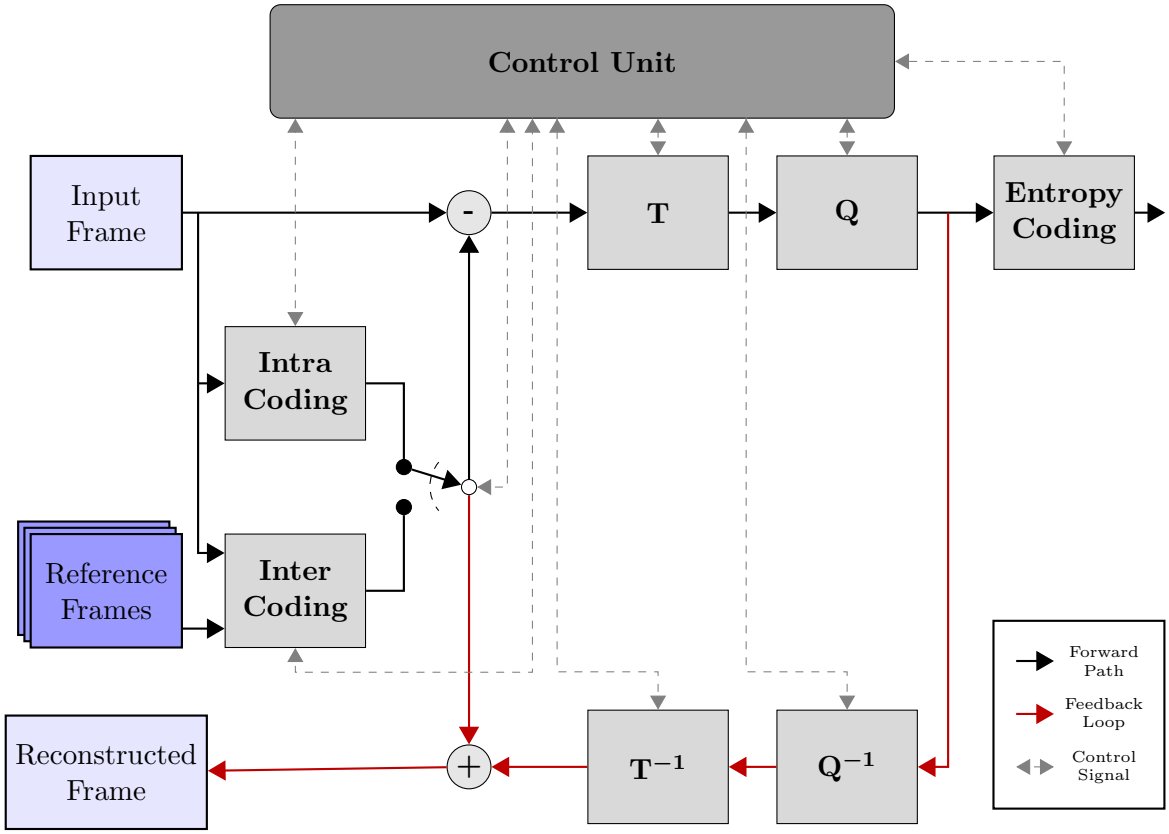


Figure 2.6: Simplified Basic Encoder Model.

The encoding process starts with the *Input Frame*, which can be of two types. *I Frames* are encoded using only the information present in themselves, i.e., using only *Intra Prediction/Coding*, while *P Frames* may use predictive coding from previously encoded frames [1].

[1]Most video codecs allow the encoding sequence to be different from the temporal sequence. This allows the currently encoding frame to use reference frames displayed after itself.

The input gets split into blocks, which get fed into the two main blocks of a video encoder: the *Intra* and *Inter* Prediction blocks.

The *Intra Coding* block, as mentioned previously, deals with the spatial redundancy, by predicting the current block from the pixels above and to the left of its upper and left edges. The prediction may be done with various algorithms, ranging from calculating the average from the reference pixels, to replicating these according to a certain direction. One such example is presented in Figure 2.7, where pixels B through H get spread across a 4×4 block, diagonally.

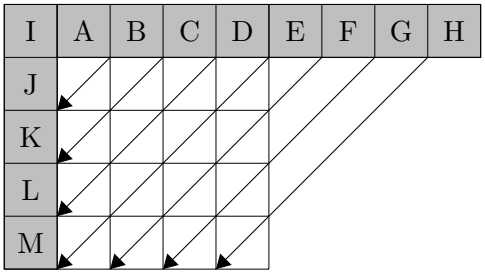


Figure 2.7: Directional Intra-prediction example.

Into the *Inter Coding* block, go two inputs. The currently encoding block, as well as a bank of previously encoded frames, named *Reference Frames*. Firstly, the frames inside the buffer get searched for blocks resembling the former input. Once found, this process generates a *motion vector*, corresponding to the difference between the position of the block found in the reference frame, and the position of the currently encoding block, as shown in Figure 2.8.

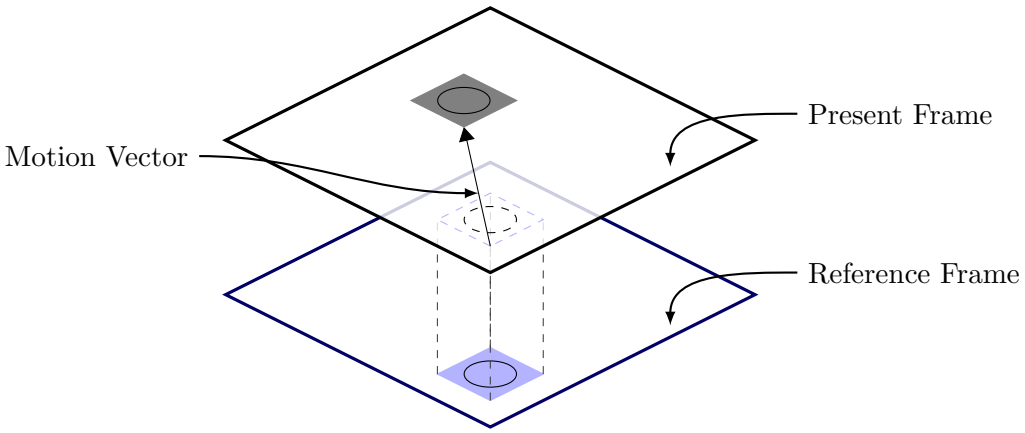


Figure 2.8: Inter-prediction example.

In most codecs, the motion vector has a precision below one pixel. This means that the matching block, from the reference frame, may be interpolated from existing pixels. This process is known as *sub-pixel interpolation*, which calculates virtual values between existing pixels.

After the prediction stage, the chosen output between the two processes, i.e., the predicted block, gets subtracted by the current one, giving origin to the *residue*. This corresponds to the pixel value differences between the original and predicted blocks. Lower *residues* indicate more efficient prediction stages.

The next stage, the *Forward Transform*, is the focus of this work. It takes the residue blocks, which may not be the same size of the prediction blocks, and evaluates them according to its spatial frequencies. Its output corresponds to a series of *coefficients*, that are related to the similarity — or *correlation* — between the input block and a series of *basis images*. This process is further explained in Chapter 3.

On the *Quantization* stage, the coefficients calculated in **T** get scaled according to a *Quantization Matrix*. This stage takes advantage of the eye's lower perception to details, and scales the higher frequency coefficients by a higher value, than the lower, more significant ones. In most of the transformed blocks, this leads to only a few low frequency components being maintained, while the others get nullified, since they are not relevant to the reconstruction of the image. Therefore, this stage is the the one that presents the higher loss, although the previously presented also introduce errors. In most of the encoding processes, this stage has the most direct impact on the obtained quality.

The wipe out of the least significant coefficients is particularly efficient when paired with the last stage before the output, the *Entropy Encoder*. On this block, **Q**'s output blocks get run sequentially via a *zig-zag scan*, which first passes through the lower frequency coefficients, followed by the higher frequency ones, as demonstrated by Figure 2.9.

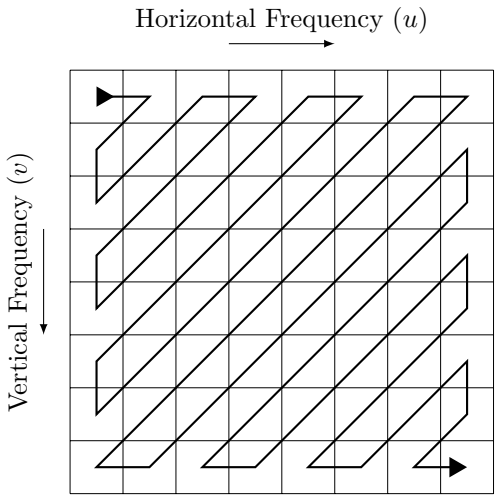


Figure 2.9: Demonstration of Zig-Zag Scan.

In most of the cases, this causes that the non-zero coefficients get read first, followed by a sequence of zeros. Such sequence benefits heavily of being encoded with Variable Length Codes (VLC), such as *Huffman Tree Codes* or Context Adaptative Binary Arithmetic Coding (CABAC). Off all the processes, this is the one that does not introduce further distortion into the encoded sequence, which is the reason it does not get included in the *feedback loop*.

The intent of this loop is to get an exact same copy of the frame reconstructed in the decoder. This reconstructed frame gets used as the reference for intra-prediction, or gets put into the reference frame buffer to be used in a later inter-prediction process.

The output of the encoder is the quantized coefficients, as well as the necessary information to recreate the encoded blocks, such as the type of prediction used, the transformation *kernel* [see p.28], quantization matrix, et al. These encoding parameters are the choices made by the *Control Unit*, which although represented by a block in Figure 2.6, may not be a local process, independent from all others.

Since *H.264*, most video codecs standardize the decoding process, specifically the allowed tools for reconstructing the video, and how to use them. This means that the encoding process is widely adaptable to the compression objectives, as long as the final product is a bitstream following the norms set on the codec's standard [9]. Therefore, the definition of a *Control Unit* is ambiguous in this context, since such unit can simply represent a set of parameters to be used throughout the encoding process[2], or an algorithm that can change between the different capabilities of the codec, in order to achieve an objective, such as a specific distortion rate, or not surpass a maximum bit rate. As expected, different objectives may lead to majorly different results, both in the output video, as well as in the used tools.

2.1.3.2 Decoder Model

As expected, the decoder (Figure 2.10) does the backwards operation of the encoder on Figure 2.6. It starts by analyzing the bitstream, separating the control information from the encoded and quantized coefficients.

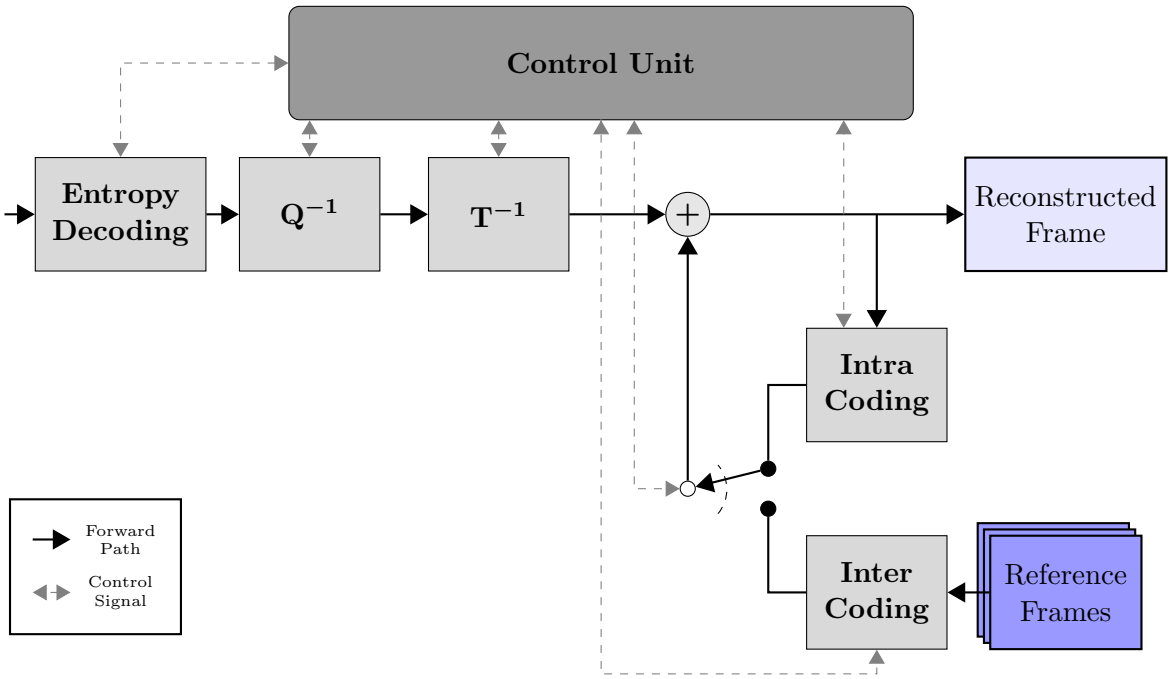


Figure 2.10: Simplified Basic Decoder Model.

Having the encoding choices performed by the encoder, the decoder returns the *coding redundancy* to the quantized coefficients, on the *Entropy Decoding* stage. This corresponds to a translation from the varying length code used in codification, back into the raw coefficients.

The *Inverse Quantization* rescales the maintained coefficients, resulting from the previous *Quantization* stage. With this, it is meant that the same quantization matrix used when dividing the transformed coefficients, in the encoder, is now multiplied by the quantized parameters. It must be kept in mind that this operation does not output an exact copy of

[2]One such example would be *lossless* compression modes, which use very a concise conditions on each stage, in order to get the least distortion.

the transformation coefficients, as a lot of information is permanently lost in **Q**. This process can be seen in Figure 2.11.

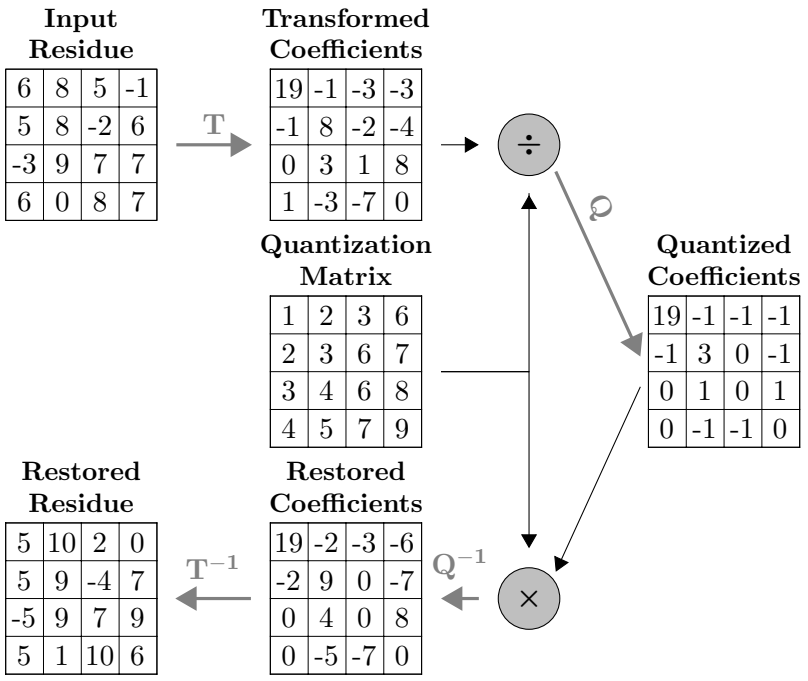


Figure 2.11: Processing of 4×4 residue block from *transformation* to restoring.

As can also be seen in this Figure, the *Inverse Transform* converts the coefficients back into spatial coordinates, therefore getting the restored residue. To obtain the final approximation of the block being decoded, this residue must be added to the same predicted block from the encoder. To do so, the *Intra* or *Inter Prediction* stages act according to the choices made in the encoding process, as to regenerate this block.

In the decoder, the *Control Unit* represents the process that organizes the different stages, according to the choices done in the encoding stage.

2.2 AV1

Being the focus of this work, in the following sections, *AV1* is discussed concerning its most relevant aspects, starting with its development process.

2.2.1 History and Development

The development of this codec started as a need to improve the bandwidth reduction of *VP9*. Therefore, the presentation of *AV1* starts by explaining the guidelines of its predecessor.

VP9 started with project *Webm*, created by *On2*, which got acquired by *Google* in 2010. This project had the objective of developing the first[3] open-source, royalty-free video codec.

[3]VP8 got openly released after the acquisition of the company, after closing the development process.

This got support from major video content producers, such as *YouTube*, *Netflix* and *Twitch*, since it represented large savings in licensing payments, from the use of MPEG's standards, which got aggravated from the difficult patenting terms of *HEVC* [12]. After release in 2013, *VP9* got adopted as *YouTube*'s default video codec for video's above *420p*, as well as other web-video consumption services, including *Facebook*.

In 2014, *Google* started working on the next generation of open-source video codecs, *VP10*. However, due to the large interest from other companies which already used the previous standard, in 2015, the *Alliance for Open Media* was created, and the the development made for this standard got inserted into *AV1*. Alongside *Google*, twelve other companies started *AOM*, including two which also had open video encoder projects, which also majorly contributed to the fast development of *AV1*: *Cisco*'s *Thor* and *Mozilla*'s *Daala*. As the time of writing, 42 companies are official members of *AOM*, englobing a wide range of markets, from video streaming services, to hardware producers.

Figure 2.12: *Alliance for Open Media* current members [13].

By 2016, *AV1* started, with the objective of reaching 30% bitrate decrease, in comparison to *VP9*. After the bitstream freeze in March 2018 and deployment of *libaom* soon later, this first objective was fulfilled. However, the compression performance did not atone for the very

high operation times of the reference software. This left a large margin for improvement, which quickly got explored with the development of other compression and decompression algorithms by the *AOM* members, such as *dav1d*, *rav1e*, *SVT-AV1*, among others. This parallel development gave origin to a competition among the corresponding teams, that benefited the adoption of the standard, since it brought a wide range of possibilities.

With the improvements verified on both encoders and decoders, *AV1* got progressively more adoption from the industry, getting support from most web browsers, as well as uploads of *AV1* encoded videos to streaming platforms [14].

Besides the advances in software solutions, shortly after the bitstream freeze, IC development companies started to develop hardware solutions. The focus started in hardware decoders for implementation in mobile devices, but some encoder solutions also have been announced. Although some claims of throughput up to *8k 60fps* have been made, third party performance tests still remain to be published [15, 16, 17, 18, 19].

2.2.2 Encoding Tools

Although the focus of this work revolves around the Transform stage, in this section, *AV1* is presented on its most relevant aspects. Some analogies are also made with *VP9*'s tools, as to justify the performance and complexity increases obtained with the most recent generation.

2.2.2.1 Partitioning

At the start of the encoding process, an input frame is divided into *superblocks*. These constitute the starting point of the compression of an image.

These blocks may be of 128×128 pixels, or 64×64. However, doing operations with such sizes would add complexity, as well as it would not prove to be efficient. Therefore, the *superblock* can be partitioned into various *prediction blocks*. These can range between 128×128 to 4×4, including rectangular blocks, with $2:1$ or $4:1$ ratios. The division of these blocks can be done recursively, where a square block divided into 4 square blocks can originate progressively smaller blocks, according to the schematic in Figure 2.13.

VP9 also included a recursive partitioning scheme, but the maximum block size is 64×64, and each block could only be divided with the $\times 4$ or $2:1$ ratios.

2.2.2.2 Intra-prediction

In Figure 2.7, it is presented one of the possible angles from the *directional prediction* mode of *Intra coding*. However, in *AV1*, this stage includes other prediction options, some being revised from previous generations, while others have never been implemented before.

On the *directional mode*, *AV1* improves massively from *VP9*, going from 8 directions to 56. This allows for better maintenance of details, especially on bigger blocks.

As to the *non-directional predictors*, *VP9* includes two different modes. In *DC*, the pixels within a block would get replicated as the average of its references. *True Motion* (TM) would calculate each pixel as the sum of the one above by the one to the left, and subtract the upper-left diagonal, i.e., $y_{(i,j)} = y_{(i,j-1)} + y_{(i-1,j)} - y_{(i-1,j-1)}$. In comparison, *AV1*'s *Smooth modes* are similar to the previous *DC*, but it has the possibility of calculating the weighted average

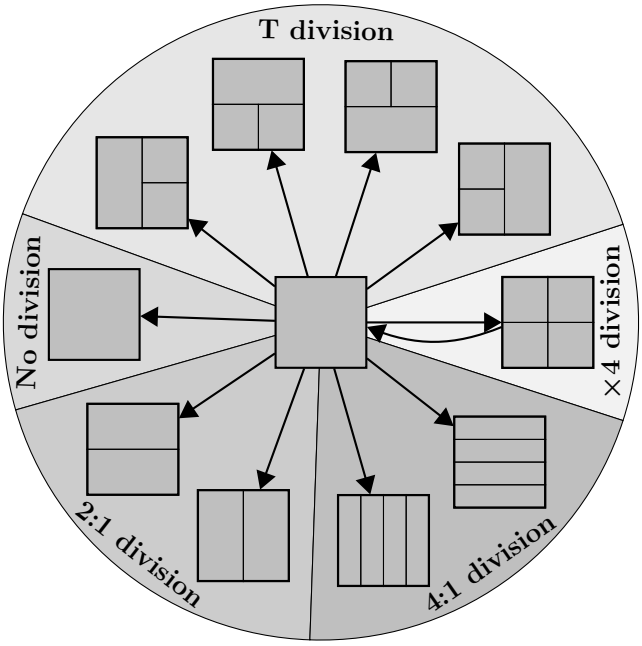


Figure 2.13: Description of the recursive partitioning scheme of *AV1*.

of the reference pixels, as well as using just one set of references, horizontal or vertical. *TM mode* gave place to *Paeth*, which makes various calculations similar to *TM*, then considering the most fitted prediction. An hardware architecture for this intra-predictor has already been implemented in *"A High Throughput Hardware Architecture Targeting the AV1 Paeth Intra Predictor"* [21].

Pallet mode also got revised and included in *AV1*. This mode is paired with other prediction techniques, limiting the pixel values to a set of possible colors. *Pallet* as well as *Intra-block copy* are especially designed for artificial video, such as video game footage, since these kind of videos contained a limited set of colors textures. *Intra-block copy* allows for the replication of a intra-predicted block, similarly to the process in inter prediction.

Finally, *AV1* introduces two new intra prediction modes that have not been implemented in previous generations. These are *Chroma from Luma* and *Recursive-filter Intra Prediction*. The first is easily understandable through its name. The chroma component of a block is calculated through the corresponding luminance values (see *"Predicting Chroma from Luma in AV1"* [22]). As to the later, it sub-divides a prediction block, and calculates each set of pixels using different filters.

2.2.2.3 Inter-prediction

This block got major innovations, as well as improvements to previous generations. Regarding the standard techniques, *AV1* improves in the number of motion vector estimation filters, going from two to four, as well as in the number of sub-pixel filters. While *VP9* allowed for three reference frames, the newer inter predictor allows to choose up to seven per frame, in a set of eight reference frames. This highly increases the necessary memory for encoding

and decoding, but allows for finer motion estimation.

As to innovations, *AV1* introduces *Warped motion*, which allows to shape the reference block on a trapezodial manner, *Global motion*, to easily shift an entire frame, as to deal with camera movements, and *Wedge mode* which allows to use different prediction schemes in the same block, among others.

Some works have already been published with advances to this stage, as well as hardware implementations [23, 24].

2.2.2.4 Transform

AV1 follows the innovations made in *VP9*, adding more transformation kernels. Besides the regularly implemented *Discrete Cosine Transform (DCT)*, the transformation blocks may now be transformed using Identity kernels or *Asymetric Discrete Sine Transform (ADST)* kernels, which can be implemented in two directions. These different options can be used independently in the columns and rows, giving origin to 16 different options of block transformations. This aspect is further explained in Chapter 3.

As to transform sizes, *AV1* allows for extra flexibility, not fixing any of the block dimensions to a certain value. This way, the block size can vary between 4×4 and 64×64, including rectangular blocks of $2:1$ and $4:1$ ratios.

2.2.2.5 Quantization

Although the simplest stage from the encoding/decoding process, *AV1* developed this stage by allowing a wider set of quantization matrixes to be used within the same frame, as well as updating the choosing criteria. While in *VP9* the *Quantization Parameter (QP)* [4] would be calculated considering the chroma components as one, now both channels (Cb and Cr) are considered independently.

Since *AV1* was targeted at web applications, one other innovation was added to this stage, which is an offset to the quantization matrixes. This is particularly effective on applications where a specific target bitrate is to be achieved.

2.2.2.6 In-loop Filtering

Although not represented in Figures 2.6 and 2.10, recent codecs include some kind of filtering to reduce compression artifacts. In *VP9* there was included a *Deblocking Filter*, which filtered the entire image, as to reduce the edging artifacts from prediction. *AV1* maintains this filter, reducing the necessary memory to implement it.

Besides the revision of the old filter, many others are added, such as the *Constrained Directional Enhancement Filter*, that filters the image directly on the prediction blocks' edges, with the same objective of the *Deblocking filter*. Some further explanation of these filters may be found in *"Film Grain Synthesis for AV1 Video Codec"* [25] and *"The Av1 Constrained Directional Enhancement Filter (Cdef)"* [26].

[4] Parameter that indicates the *quantization matrix* to use (higher values indicate more severe quantization).

2.2.3 Performance Analysis

AV1's decoding specification has not changed since the release of the standard and freeze of the bitstream. However, this is not verified on the implementations of the standard. Even *libaom*, which is intended to serve as a guideline for future implementations, has been severely improved since its release in June 2018.

The comparison of *AV1* throughout these developing months has been divided in two major categories: *Quality* and *Timing*. The first depends on the standard itself, and on how the encoding tools are able to compress the video, while maintaining its playback capabilities. Therefore, if the encoding objectives are maintained throughout the development of the encoders/decoders, this parameter should not vary. However, the same cannot be said of the *Timing Performance*, since as more efficient tools get released, it is expected that the time to encode/decode a video gets reduced, as to reach real-time usability.

According to Moscow State University [27], *AV1* achieved its objective of highly reducing the necessary bitrate. On this test, five *1080p* sequences have been encoded using different implementations of *H.264/AVC*, *H.265/HEVC*, *VP9* and *AV1*. The different softwares have been configured on a similar manner, as to encode the sequences with similar quality, and the average bitrate per codec was compared relatively to *H.264*. The results are presented in Figure 2.14.

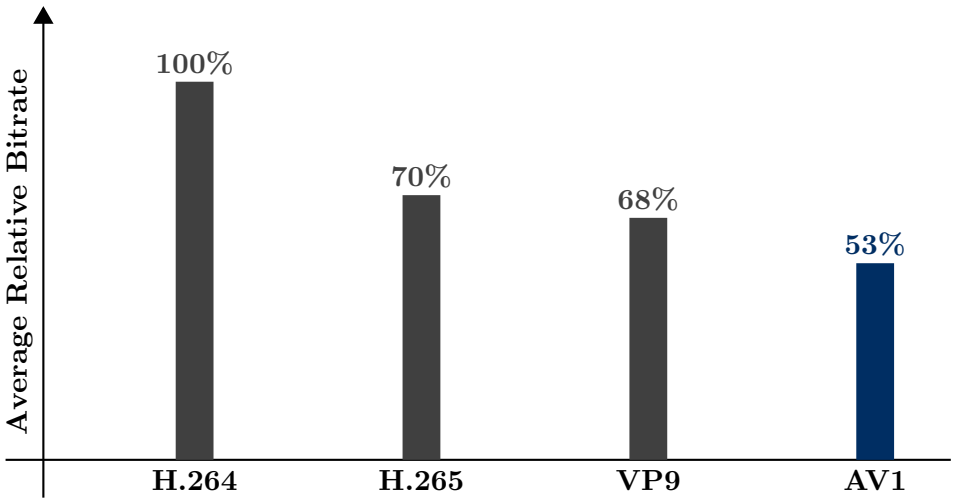


Figure 2.14: *AV1* bitrate savings [27].

These results may vary greatly with the performed tests, as the encoding tools may prove to be more adequate to certain types of videos. On a different test, performed by *Facebook* [28], *AV1* presents a higher performance than the one presented previously, as seen in table 2.1. Here, videos of various resolutions were encoded with *VP9* and *H.264* with equivalent parameters, and the obtained bitstreams are compared to *AV1*'s, according to Bjontegaard-Delta rate (BD-rate).

As it may be seen, as the resolution increases, so do the bitrate savings. This leads to believe that if the same test were to be performed with *4K* and *8K* sequences, higher performances would be verified.

As to the encoding times, in two articles from *Streaming Media* [29, 30], it is possible to

Table 2.1: BD-rate of *VP9* and *H.264* codecs, when compared to *AV1* (negative corresponds to bitrate savings) [28].

Codec	**Resolutions**				
	360p	**480p**	**720p**	**1080p**	**Average**
VP9	-29.5%	-32.5%	-32.3%	-35.9%	-32.5%
H.264	-43.4%	-49.3%	-51.2%	-57.9%	-50.3%

see the improvements made on *libaom*. In Table 2.2 there are presented the encoding times of a 5 second clip, shortly after the reference software was released, August 2018, and in March 2019, under the same conditions. Besides the performance of *libaom*, software for *H.264/AVC*, *H.265/HEVC* and *VP9* is also evaluated.

Table 2.2: Encoding times of different video encoders, and improvements on *AV1* [29, 30].

Codec	**Encoding Time (s)**	
	2018	**2019**
AV1	226 080	736
H.265	289	
VP9	226	
H.264	18	

From these results, it is possible to conclude that *AV1* is a promising codec. When quality and compression gains are considered, it is already verifiable that the codec presents better performances than its predecessors, in some cases even beating its objective of 30% improvement over *VP9*. However, when considering the timing issues, the results do not prove as optimistic. As the time of writing, the encoding solutions are still far away from a real-time usability. Although, as better software and hardware solutions get developed, this objective may be achieved in the near future.

References

[1] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Fourth edition. New York, NY: Pearson, 2018. ISBN: 978-0-13-335672-4.

[2] Yun Qing Shi and Huifang Sun. *Image and Video Compression for Multimedia Engineering: Fundamentals, Algorithms, and Standards*. 2. ed. Image Processing Series. Boca Raton, Fla.: CRC Press, 2008. ISBN: 978-0-8493-7364-0.

[3] Owlcation. *Anatomy of the Eye: Human Eye Anatomy*. https://owlcation.com/stem/Anatomy-of-the-Eye-Human-Eye-Anatomy.

[4] Kathy Mullen. "The Contrast Sensitivity of Human Color Vision to Red-Green and Blue-Yellow Chromatic Gratings". *The Journal of physiology* 359 (Mar. 1985), 381–400. DOI: 10.1113/jphysiol.1985.sp015591.

[5] *Freepik*. https://www.freepik.com.

[6] *YUV Sequences*. http://trace.eas.asu.edu/yuv/.

[7] Andrew Watson. "Efficiency of a Model Human Image Code". *Journal of the Optical Society of America. A, Optics and image science* 4 (Jan. 1988), 2401–17. DOI: 10.1364/JOSAA.4.002401.

[8] Kais Rouis and Chaker Larabi. "Perceptual Video Content Analysis and Application to HEVC Quantization Refinement". Nov. 2018, 1–6. DOI: 10.1109/EUVIP.2018.8611749.

[9] *AV1 Bitstream & Decoding Process Specification*. https://aomediacodec.github.io/av1-spec/av1-spec.pdf. 2019.

[10] Luciano Agostini. *Desenvolvimento de Arquiteturas de Alto Desempenho Dedicadas à Compressão de Vídeo Segundo o Padrão H.264/AVC*. 2007.

[11] Debargha Mukherjee. *AllThingsRTC 2019 - Opening Keynote - Past, Present and Future of AV1*. June 2019.

[12] Streaming Media. *HEVC Advance Patent Pool Creates Confusion, Lacks Transparency*. July 2015.

[13] aomedia. *Home*. https://aomedia.org/.

[14] Nathan Egge. "Latest Technical and Business Progress with AV1". *NAB Streaming Summit* (Apr. 2019), 59.

[15] *Allegro DVT Introduces the Industry First Real-Time AV1 Video Encoder Hardware IP for 4K/UHD Video Encoding Applications*. http://www.allegrodvt.com/allegro-dvt-introduces-the-industry-first-real-time-av1-video-encoder-hardware-ip-for-4kuhd-video-encoding-applications/. Apr. 2019.

[16] *NGCodec Announces AV1 Support and a 2X Performance Improvement in Broadcast Quality Live Video Encoding.* https://ngcodec.com/press-releases/2019-1-7-ngcodec-announces-av1-support-and-a-2x-performance-improvement-in-broadcast-quality-live-video-encoding.

[17] Anton Shilov. *Realtek Demonstrates RTD2893: A Platform for 8K Ultra HD TVs.* https://www.anandtech.com/show/14560/realtek-demonstrates-rtd2893-a-platform-for-8k-ultrahd-tvs.

[18] *Realtek Launches Worldwide First 4K UHD Set-Top Box SoC (RTD1311), Integrating AV1 Video Decoder and Multiple CAS Functions - REALTEK.* https://www.realtek.com/en/press-room/news-releases/item/realtek-launches-worldwide-first-4k-uhd-set-top-box-soc-rtd1311-integrating-av1-video-decoder-and-multiple-cas-functions.

[19] *Socionext Implements AV1 Encoder on FPGA over Cloud Service.* http://socionextus.com/pressreleases/socionext-implements-av1-encoder-over-cloud-service/. June 2018.

[20] Yue Chen et al. "An Overview of Core Coding Tools in the AV1 Video Codec". *2018 Picture Coding Symposium (PCS)*. San Francisco, CA: IEEE, June 2018, 41–45. ISBN: 978-1-5386-4160-6. DOI: 10.1109/PCS.2018.8456249.

[21] M. Corrêa et al. "A High Throughput Hardware Architecture Targeting the AV1 Paeth Intra Predictor". *2019 IEEE 10th Latin American Symposium on Circuits Systems (LASCAS)*. Feb. 2019, 93–96. DOI: 10.1109/LASCAS.2019.8667544.

[22] L. Trudeau, N. Egge, and D. Barr. "Predicting Chroma from Luma in AV1". *2018 Data Compression Conference*. Mar. 2018, 374–382. DOI: 10.1109/DCC.2018.00046.

[23] Z. Deng and I. Moccagatta. "Hardware-Friendly Inter Prediction Techniques for AV1 Video Coding". *2017 IEEE International Conference on Image Processing (ICIP)*. Sept. 2017, 948–952. DOI: 10.1109/ICIP.2017.8296421.

[24] R. Domanski et al. "High-Throughput Multifilter Interpolation Architecture for AV1 Motion Compensation". *IEEE Transactions on Circuits and Systems II: Express Briefs* 66.5 (May 2019), 883–887. ISSN: 1549-7747. DOI: 10.1109/TCSII.2019.2909705.

[25] A. Norkin and N. Birkbeck. "Film Grain Synthesis for AV1 Video Codec". *2018 Data Compression Conference*. Mar. 2018, 3–12. DOI: 10.1109/DCC.2018.00008.

[26] S. Midtskogen and J. Valin. "The Av1 Constrained Directional Enhancement Filter (Cdef)". *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Apr. 2018, 1193–1197. DOI: 10.1109/ICASSP.2018.8462021.

[27] Dr Dmitriy Vatolin et al. "MSU Codec Comparison 2018 Part IV: FullHD Content, High Quality Use Case" (2019), 42.

[28] Facebook Engineering. *AV1 Beats X264 and Libvpx-Vp9 in Practical Use Case*. https://engineering.fb.com/video-engineering/av1-beats-x264-and-libvpx-vp9-in-practical-use-case/. Apr. 2018.

[29] Streaming Media Magazine. *AV1: A First Look*. https://www.streamingmedia.com/Articles/ReadArticle.aspx?ArticleID=127133. Aug. 2018.

[30] *Good News: AV1 Encoding Times Drop to Near-Reasonable Levels*. https://www.streamingmedia.com/Articles/ReadArticle.aspx?ArticleID=130284. Mar. 2019.

CHAPTER 3

Video Coding Transforms

3.1 Introduction

As mentioned previously, the basic principle behind the compression of video, is the reduction of inter-pixel/inter-symbol correlation. The various integral blocks of a video compression system try to accomplish this objective through different strategies. The *Intra-frame* and *Inter-frame Prediction* exploit spatial and temporal correlation, respectively. Through the subtraction of the input by the output of one of these blocks, and the attainment of the *residue*, the next compression stage is made in the *Transform* block, which is the focus of this work.

The technique implemented by this process relies on the energy compaction in the frequency domain to reduce the correlation within a frame block, i.e. the input of the Transform block is evaluated on its main frequencies — the *transform coefficients* — on a spatial domain, similarly to the process executed on a *Fourier Transform*. Once each block is quantized on these coefficients, the compression is made with the removal of the least significant ones, on the *Quantization* stage.

The objective of this chapter is to give the reader a basic understanding of the theoretical basis behind said *Transformations*, as well as to introduce the most commonly used ones. Later, *libaom*'s *Transform stage* is presented, as well as some benchmarks of its performance.

3.2 Background

3.2.1 Basis Vector/Image Interpretation

A useful interpretation, and a good starting point to the study of this process, is to see it as the decomposition of an N length input, $\vec{g}$ as a set of basis vectors (in 1D transforms) or images/matrices (in 2D transforms). The transformation outputs , $\mathcal{G}_i$, can be seen as the weights of each basis vector/image, $\vec{e_i}$, that summed return the restored input, $\underset{*}{g}$, i.e.

$$\underset{*}{\vec{g}} = \sum_{i=0}^{N-1} \mathcal{G}_i \vec{e_i} \tag{3.1}$$

which means that the coefficients are related to the amount of correlation between the input and each basis component, and can be obtained with the *inner product* of the input and each

basis vector.

$$\mathcal{G}_i = \vec{e_i}^T \vec{g} \tag{3.2}$$

Since each input vector will have different correlation values between the various basis vectors, this operation accomplishes two main objectives:

- De-correlation of the input values
- Signaling of the most important basis vectors.

Considering a 2D image, $\mathbf{G}(x, y)$, and its corresponding transformed coefficients, $\mathcal{G}(u, v)$, where (x, y) are the pixel coordinates, and (u, v) are the corresponding coordinates in the transform domain, we can obtain an analogous version of Equation 3.2 as

$$\mathcal{G}(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \mathbf{G}(x, y) f(x, y, u, v) \tag{3.3}$$

Similarly, we can re-obtain the restored original picture

$$\underset{*}{\mathbf{G}}(x, y) = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} \mathcal{G}(u, v) i(x, y, u, v) \tag{3.4}$$

where $f(x, y, u, v)$ and $i(x, y, u, v)$ are the *forward* and *inverse transformation kernels*. To better explain the concept of these, first it is needed to introduce the two following concepts.

3.2.1.1 Separability

A useful characteristic of 2D Video Coding Transforms is its ability to be independently calculated between rows and columns. This means that given a 2D block as input, the transform coefficients can be calculated first with the *horizontal transform*, and then with the *vertical transform*, or vice-versa.

This aspect is applicable if the following conditions are applied

$$f(x, y, u, v) = f_1(x, u) f_2(y, v) \tag{3.5}$$

$$i(x, y, u, v) = i_1(x, u) i_2(y, v) \tag{3.6}$$

This means that the Equation 3.3 is reconstructed as 2 independent and sequential operations

$$\mathcal{G}_{temp}(x, v) = \sum_{y=0}^{N-1} \mathbf{G}(x, y) f_2(y, v) \tag{3.7}$$

$$\mathcal{G}(u, v) = \sum_{x=0}^{M-1} \mathcal{G}_{temp}(x, v) f_1(x, u) \tag{3.8}$$

On AV1, due to the various implemented transformation kernels, this aspect is severely explored, since the only way of implementing the combination of different 1D kernels, is to calculate them independently. This aspect is further explained with the following concept.

3.2.1.2 Symmetry

Taking Equation 3.5, a transformation kernel is said to be symmetric if

$$f_1(y, v) = f_2(x, u) \tag{3.9}$$

This characteristic is particularly useful because it makes the forward and inverse transformations expressible as matrix multiplications. Therefore, the equations 3.3 and 3.4 are represented, respectively, as

$$\mathcal{G} = F^T \mathbf{G} F \tag{3.10}$$

$$\underset{*}{\mathbf{G}} = I^T \mathcal{G} I \tag{3.11}$$

where F and I are the forward and inverse transform matrices. This aspect is only possible for square matrix, i.e., input blocks with the same height and width.

This concept is not exploited in AV1, since the use of different 1D transformation kernels, and rectangular block sizes ($M \neq N$) make the 2D transform asymmetric, and therefore, not executable as matrix multiplication. Consequently, the block transformation is made as 2 separate 1D operations, as shown previously.

Looking now at equation 3.4, we can interpret the inverse transformation kernel as a set of basis images, dependent of the (u, v) pair. By this, it is meant

$$\underset{*}{\mathbf{G}}(x, y) = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} \mathcal{G}(u, v) I_{u,v} \tag{3.12}$$

where

$$I_{u,v} = \begin{bmatrix} i(0,0,u,v) & i(0,1,u,v) & \dots & i(0,M-1,u,v) \\ i(1,0,u,v) & i(1,1,u,v) & \dots & i(1,M-1,u,v) \\ \vdots & \vdots & \ddots & \vdots \\ i(N-1,0,u,v) & i(N-1,1,u,v) & \dots & i(N-1,M-1,u,v) \end{bmatrix} \tag{3.13}$$

Therefore, the forward and inverse transformation process can be seen as the deconstruction of an input block, into a set of $M \cdot N$ basis images, dependent of the used transformation kernel. As expressed in Equations 3.5 and 3.6, this analogy can be made on a 1D space .

Given a general comprehension of the theoretical principles behind the *Transform* block, now the most common transformation kernels are introduced, with focus on the AV1 video codec.

3.3 Transformation Kernels

3.3.1 Discrete Fourier Transform (DFT)

Although it is not implemented in video coding, it is widely used in digital signal processing, and many of the used transformation kernels are approximations of this function.

It has it is roots on the *Fourier Transform*, whose forward and inverse transformations are expressed in Equations 3.14 and 3.15, respectively.

$$\mathcal{G}(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \mathbf{G}(x,y)e^{-j2\pi(ux+vy)}dx\,dy \tag{3.14}$$

$$\underset{*}{\mathbf{G}}(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \mathcal{G}(u,v)e^{j2\pi(ux+vy)}du\,dv \tag{3.15}$$

Once considered a finite number of points, the previous equations become

$$\mathcal{G}(u,v) = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} \mathbf{G}(x,y)e^{-j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)} \tag{3.16}$$

$$\underset{*}{\mathbf{G}}(x,y) = \sum_{u=0}^{M-1}\sum_{v=0}^{N-1} \mathcal{G}(u,v)e^{j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)} \tag{3.17}$$

which correspond to replacing the kernels in equations 3.3 and 3.4 with

$$f(x,y,u,v) = \frac{1}{MN}e^{-j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)} \tag{3.18}$$

$$i(x,y,u,v) = e^{j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)} \tag{3.19}$$

The position of the multiplication factor, $1/MN$, is irrelevant, and in some works is divided into two terms in the forward and inverse kernels, $1/M$ and $1/N$, or even $1/\sqrt{MN}$.

Because of the use of complex numbers, this operation tends require a high computational effort, whence its disuse in video coding.

3.3.2 Discrete Walsh-Hadamard Transform (WHT)

This transformation replaces the sum of sines and cosines of the DFT, alternating positive and negative 1's, depending on the binary representation of the inputs.

Considering the inputs of the transform to be represented with K bits, where $K-1$ is the most significant bit (b_{K-1}), the forward and inverse kernels are represented as

$$f(x,y,u,v) = i(x,y,u,v) = \frac{1}{\sqrt{MN}}(-1)^{\sum_{i=0}^{K-1}\lfloor b_i(x)p_i(u)+b_i(y)p_i(v)\rfloor} \tag{3.20}$$

where

$$\begin{aligned} p_0(u) &= b_{K-1}(u)\\ p_1(u) &= b_{K-1}(u) + b_{K-2}(u)\\ &\vdots\\ p_{K-1}(u) &= b_1(u) + b_0(u) \end{aligned} \tag{3.21}$$

3.3.3 Discrete Cosine Transform (DCT)

The most commonly used transform, the *DCT*, was published by N. Ahmed et al. in 1974 [3]. Since then, it has been adopted in a wide range of applications, being the only transform used in the first generations of video codecs, as well as in *still image compression*, being the basis of the *JPEG* standard.

It is frequently compared to the *DFT*, due to the similarity of their operation. However, as the name implies, the *DCT* relies on the cosine function to create its basis images, which is a *periodic* and *symmetrically even* function. Therefore, as mentioned by A. V. Oppenheim [4], *"Just as the DFT involves an implicit assumption of periodicity, the DCT involves implicit assumptions of both periodicity and even symmetry"*. This is easily observable once considered the equivalent process of both algorithms. Taking an L-point sequence, $g(n)$, the calculation of the *DFT* and *DCT* of such sequence is equivalent to the processes presented at Table 3.1.

Table 3.1: Similarity between the processes of the *DFT* and the *DCT*.

Step	***DFT***	***DCT***
1	Repeat $g(n)$ every L points, giving origin to $\widetilde{g}_L(n)$	Concatenate $g(n)$ with a flipped version of itself, creating a $2L$ sequence, $g_{2L}(n)$, and repeat it, giving origin to $\widetilde{g}_{2L}(n)$
2	Calculate the *Fourier* expansion of $\widetilde{g}_L$	Calculate the *Fourier* expansion of $\widetilde{g}_{2L}$
3	Keep the first L coefficients and set all others to 0	Keep the first L coefficients, and set all others to 0

The main reason behind the heavy adoption of the *DCT* is its great energy compaction on the lower frequencies, where most of the energy in a picture is packed. If the output of the first step of Table 3.1 is observed, this aspect is more easily understood. In Figure 3.1, a 4 point sequence, corresponding to the filled points, gets replicated throughout the discrete time axis, according to the corresponding transform.

Due to the back-to-head repetition seen in Figure 3.1a, there is a disruption every L points, which gives origin to high frequency components in the *Discrete Fourier Transform*. Therefore, the more continuous behavior obtained with the back-to-back repetition of the *DCT* gives origin to higher significance low frequency coefficients. However, there are many ways of creating a periodic and symmetric sequence from a finite number of points. This factor has led to the implementation of different versions of the *DCT*, which differ in minor details between themselves. These differences are consequence of the way the symmetry is obtained, which can be observed in Figures 3.1b to 3.1e. The represented implementations are referred to as *DCT-I* to *DCT-IV*, but other possibilities exist. Their definition depends on the overlapping of points when repeating each sequence.

Since the *DCT* in *AV1* is implemented in one dimension, the description of the following kernels is also made in 1D. Therefore, the dimension of the transform, L, is referring either to the blocks' width or height, depending if the operation is made to the rows or columns, respectively (M or N, previously).

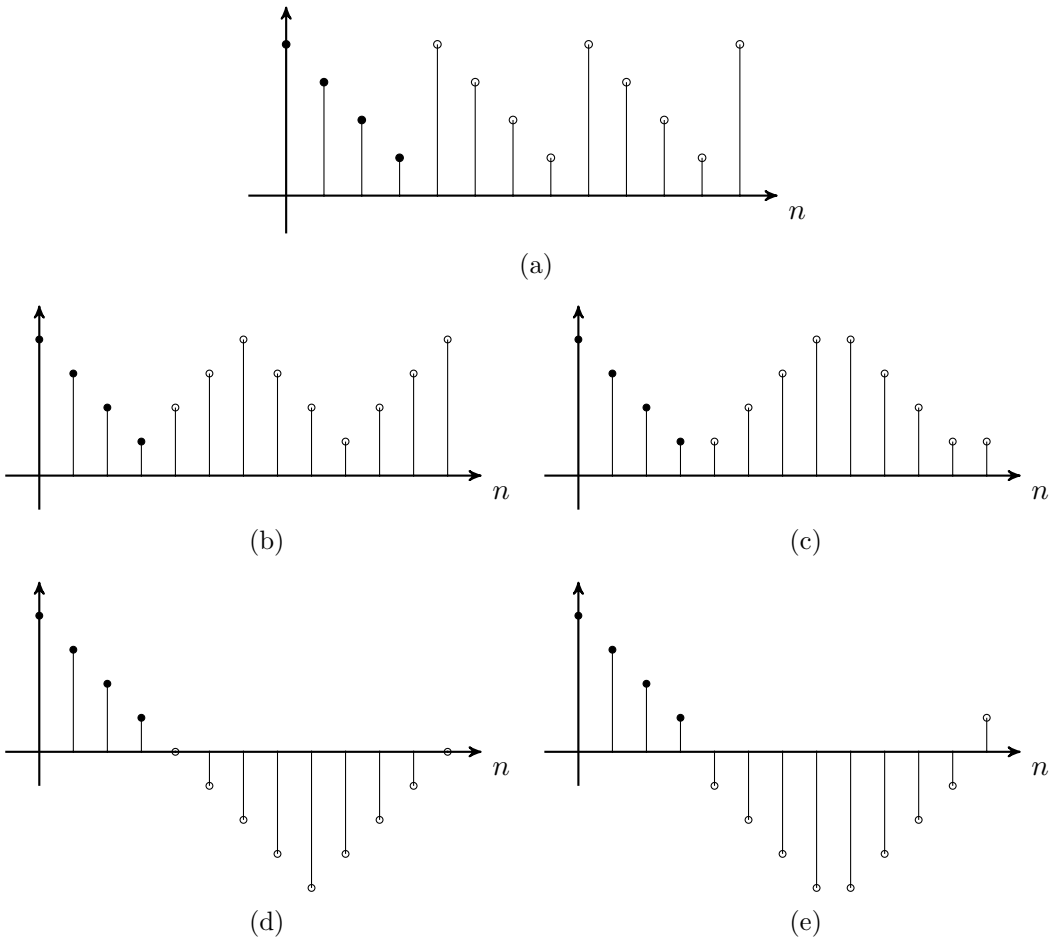


Figure 3.1: Sequences generated in the first step of Table 3.1for the DFT and different DCTs. Filled dots correspond to the original sequence ((a) - *DFT*; (b)) - *DCT-I*; (c)) - *DCT-II*; (d)) - *DCT-III*; (e)) - *DCT-IV*).

DCT-I The sequence created with first version of the DCT has overlapping points at $n = k(L-1),\ k = 0, 1, 2, ...$, making the overall period of the final sequence $2L - 2$.

$$f(x,u) = \frac{2}{L-1}\alpha(x)\cos\left(\frac{\pi x u}{L-1}\right) \tag{3.22}$$

where

$$\alpha(x) = \begin{cases} \frac{1}{2}, & x = 0 \vee x = N-1 \\ 1, & 1 \leq x \leq N-2 \end{cases} \tag{3.23}$$

The inverse transform becomes

$$i(x,u) = \alpha(u)\cos\left(\frac{xu\pi}{L-1}\right) \tag{3.24}$$

DCT-II Usually referred to as "the *DCT*", it is by far the most implemented version, being the only one mentioned in many of the studied works.

As seen in Figure 3.1c, this version has no overlap on the created sequence, making the period $2L$, and the points of symmetry $kL - \frac{1}{2}$.

$$f(x,u) = i(x,u) = \beta(u)\cos\left(\frac{(2x+1)u\pi}{2L}\right) \tag{3.25}$$

$$\beta(u) = \begin{cases} \sqrt{\frac{1}{L}}, & u = 0 \\ \sqrt{\frac{2}{L}}, & 1 \leq u \leq N-1 \end{cases} \tag{3.26}$$

DCT-III Named the *inverse* of DCT-II, due to the switch of the transform and pixel coordinates.

$$f(x,u) = i(x,u) = \beta(u)\cos\left(\frac{(2u+1)x\pi}{2L}\right) \tag{3.27}$$

$$\beta(u) = \begin{cases} \sqrt{\frac{1}{L}}, & u = 0 \\ \sqrt{\frac{2}{L}}, & 1 \leq u \leq N-1 \end{cases} \tag{3.28}$$

DCT-IV Is the basis of the *Modified Discrete Cosine Function (MDCT)*, where the input blocks overlap.

$$f(x,u) = i(x,u) = \sqrt{\frac{2}{L}}\cos\left(\frac{(2u+1)(2x+1)\pi}{4L}\right) \tag{3.29}$$

3.3.4 Discrete Sine Transform (DST)

Similarly to the DCT, there is also the possibility to represent a finite sequence as a sum of discrete *sine* functions, giving origin to the *DST*. Contrarily to the former presented transform, this variant uses sinusoidal functions to generate its basis images, which gives origin to *odd symmetric* sequences.

In the same way as its *even* counterpart, there are various different ways off accomplishing such symmetry, which also gives origin to eight different variations of this Transform. However, due to its misuse over the DCT, only the *DST-II* is presented.

$$f(x,u) = i(x,u) = \sqrt{\frac{2}{L+1}} \sin\left(\frac{(j+1)(u+1)\pi}{L+1}\right) \tag{3.30}$$

Equivalently to what happens with the DFT, the odd symmetry of this function gives origin to discontinuities, which are undesirable when coding video blocks, since they lead to less significant low frequency coefficients, and therefore higher quantization errors.

3.3.5 Asymmetric Discrete Sine Transform (ADST)

The symmetric behavior of previous transforms lead to better performance on evenly spread residue blocks, i.e. when the pixel values post-subtraction (and before transformation) have roughly the same value across the whole block.

However, due to the directional spatial prediction, the residue on one boundary of the block may differ from the others, since the chosen direction for prediction may prove more efficient on one section of the block. This leads to worse energy compression, when using transforms like the *DCT* or *WHT*.

In order to address this problem, VP9 introduced a new transform called *Asymmetric Discrete Sine Transform (ADST)*, which corresponds to an alternative implementation of the DST with the addition of frequency and phase shifts.

This enhancement provides the developer with a high degree of liberty, since the basis images can be adapted with the variation of the shifts. On AV1, there is only one ADST implementation per block size. However this transformation can be done in two directions, i. e., the input vector can be transformed front-to-back and vice-versa. *AOMedia* named these transforms *ADST* and *Flip-ADST*, according to the direction of the input vector.

3.4 *Libaom*'s Integer Transformations

In battery driven applications, computing power plays an important role. Consequently, any approach that leads to lower computational costs, without compromising the image quality, tends to get incorporated into a video codec.

When considering the *Transform Stage*, a widely adopted approach is the use of *integer transforms*. The objective of such operations is to maintain the features of floating point implementations, but severely reducing the complexity, decreasing the necessary operations to arithmetic additions and integer multiplications. In many cases, the latter are implemented with bitwise shifts and additions.

From the transforms presented throughout section 3.3, there have been several methods of developing integer counterparts. Most of the fast implementations are based in either *Fast Fourier Transform* algorithms or in the *Walsh-Hadamard Transform* [10, 11]. Since the objective of this work was to develop a *Transform Co-processor* for *libaom*, the focus of this section resolves around these kernels.

3.4.1 Functioning and Implementation

The first analysis of this section was made through the study of the transformation stage of the reference software. Its main workflow is represented in Figure 3.2.

This stage is controlled by a configuration set, which is chosen according to the desired encoding objectives. These parameters control the transformation block's width and height (`size_col` and `size_row` [1], respectively), the transformation kernels to use in the rows and columns, the precision to use in the sine and/or cosine coefficient approximations, as well as other parameters for overflow control. Associated to the transformation kernel chosen, the variables `ud_flip` and `lr_flip` are also set. The first one is set to *1* if the block's columns are to be transformed with the *Flip-ADST* kernel. If such choice is applied to the rows, the second variable is, likewise, set to *1*. These variables control if the input rows are flipped vertically, and/or if the coefficients resulting from the column transformation are flipped horizontally [2].

The choosing of these parameters will not be addressed in this work, since *AV1* allows for a great deal of maneuverability to the designer, as to adjust each encoder/decoder pair to the desired application. In this regard, *libaom* allows for a high number of configuration options, that dramatically change the parameters chosen in the transformation stage, as well as in the rest of the system.

Throughout the represented process, many of the operations are done with sequential, iterative processes, e.g., the input vector selection or the flipping operations. Such operations would greatly benefit of a hardware implementation, since they are easily parallelizable, as the objectives of *AV1* suggested. However, on this work, the focus relies of the T block, i.e., the transformation kernel itself.

Independently of the chosen transform, the operation is done sequently, in various stages. In each of these, the corresponding intermediary coefficients get calculated as function of two of the previous calculated coefficients. These, in most of the stages, are multiplied by a specific integer approximation of a cosine/sine value. Such approximations, as mentioned previously, depend on the number of bits on which they are represented.

The arrays on which the calculated cosine and sine values are stored, `cospi` and `sinpi`, respectively, are bi-dimensional. The first dimension, `K`, has 7 positions, corresponding to 10 to 16 bits approximations. The second dimension, `n`, has 64 positions for `cospi` and 5 for `sinpi`, representing the first quadrant of the trigonometric circle. Each position is calculated according to Equations 3.31 and 3.32, where n represents the position in the array, and K corresponds to the number of bits. Therefore, `cospi[K][0]` corresponds to $\cos(0)$, `cospi[K][63]` is $\cos(63\pi/128)$, and the following positions can also be associated to a certain angle.

$$\texttt{cospi[K][n]} = \left\lfloor 2^K \cos\left(\frac{n\pi}{128}\right)\right\rceil \tag{3.31}$$

$$\texttt{sinpi[K][n]} = 2^K \left\lfloor \frac{2}{3}\sqrt{2}\sin\left(\frac{n\pi}{9}\right)\right\rceil \tag{3.32}$$

The `sinpi` array is only used in the shortest length of the *ADST*, which is the reason it only has five positions. All other versions of this kernel use `cospi` to get the desired value.

[1] Each number does not correspond to the number of elements in columns and rows, but rather to the number of rows and columns.

[2] Here, the notation of *horizontally* or *vertically* is set considering a matrix input block. In the 1D transform implemented in *libaom*, this just means that what would be the last coefficient is now the first, and so on.

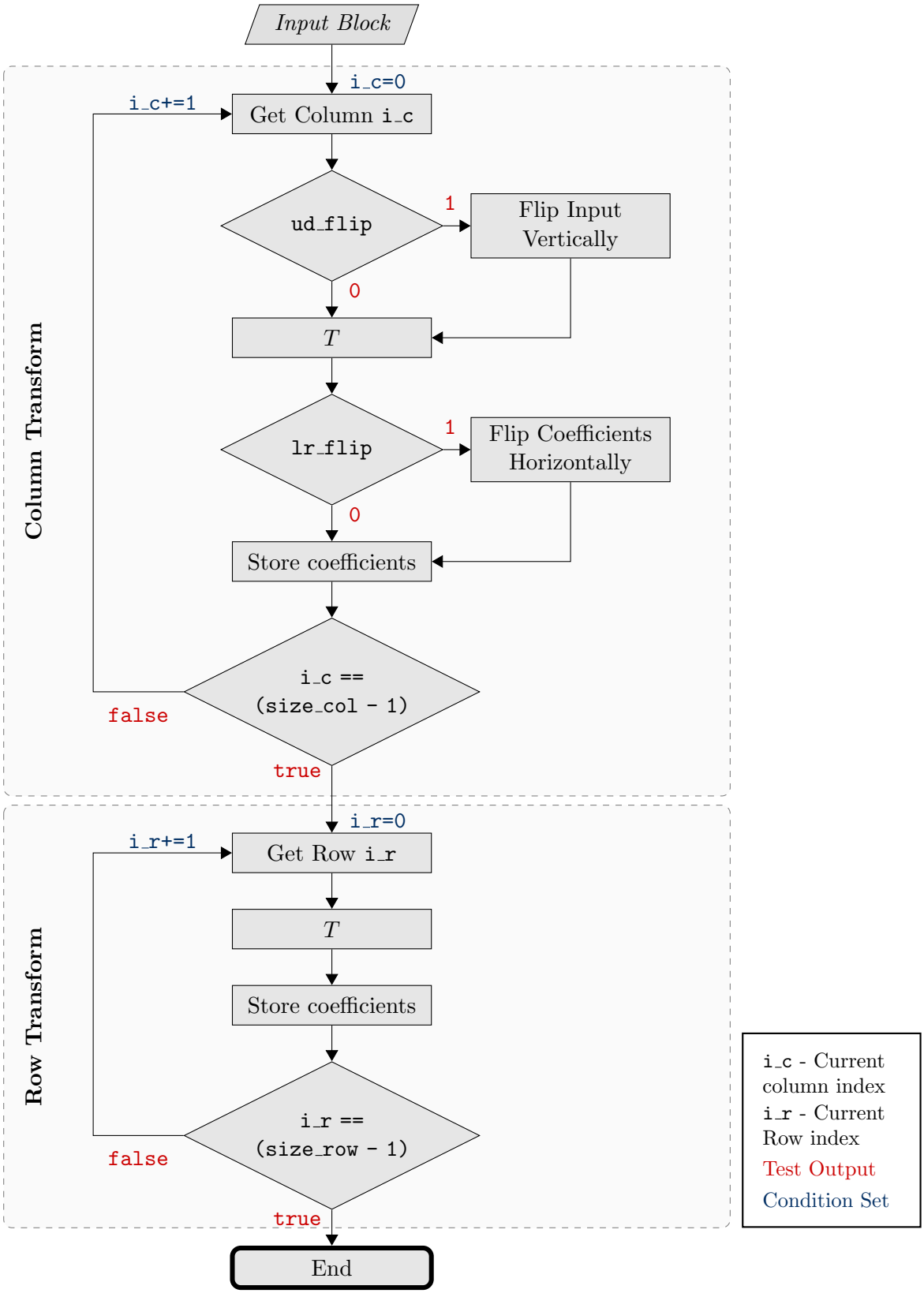


Figure 3.2: Flowchart of the Transform Stage on *libaom*.

Most of the intermediary coefficients inside each stage are calculated with the function `half_btf`, which performs the operation represented in Equation 3.33. This function takes the two previously calculated coefficients, two values from the previously introduced approximations arrays, as well as the number of bits used to represent these, and maps the result from the multiplications and sum of the first inputs to the desired number of bits.

$$\texttt{half\_btf(w0, in0, w1, in1, K)} \;\widehat{=}\; \left\lfloor \frac{w_0 \cdot in_0 + w_1 \cdot in_1 + 2^{K-1}}{2^K} \right\rfloor \tag{3.33}$$

Although the code implementation is sequential, the *8 length* transformation kernels are represented in Figures 3.5 and 3.6 as parallel block diagrams, with the diverse stages in series. While *AV1* accepts *transform block* sizes varying between 4 and 64, the method of transformation is similar between the different sizes.

Both pictures start with the input vector components, i.e., `x0` to `x7`. The following sum's represent the addition of the two pointing values, in case the that the arrow guiding these does not present any further coefficient. If such is verified, the operation to be realized is the one presented in Equation 3.33. The value near each arrow is referred to the equivalent `cospi` position, that multiplies by the result coming from the arrow's origin. Figure 3.3 presents a visual aid for the following schematics.

Both *DCT* and *ADST* are implemented using the method firstly described in 1977 by Wen-Hsiung Chen et. al, in *"A Fast Computational Algorithm for the Discrete Cosine Transform"* [12]. This approach consists of sequential *butterfly rotations*, which correspond to the various rotations obtained with the additions and subtractions of nodes of opposite ends. These operations are easily parallelizable, making this approach widely used in most hardware *DCT* implementations to this day ([13, 14, 15, 16]).

The identity transforms, *IDTX*, are the simplest of the ones implemented in *libaom*, since they consist of a scale factor, which varies throughout the transform sizes. On the 4 and 16 length transforms, the scaling factor includes a 12-bit integer approximation of the square root of 2, which is calculated through

$$N_{\sqrt{2}} = \left\lfloor 2^{12}\sqrt{2} \right\rceil = 5793 \tag{3.34}$$

Being so, the input also suffers an additional mapping, similar to the operation in Equation 3.33. These operations are demonstrated in Figure 3.4.

With the forward transformations explained and represented graphically, it is easily understandable that the corresponding inverses correspond to the backwards operation in Figures 3.4, 3.5 and 3.6. With this, it is meant that only the direction of the arrows change, and the corresponding procedure is done right-to-left, i.e., the output's position, `y`, is now the input.

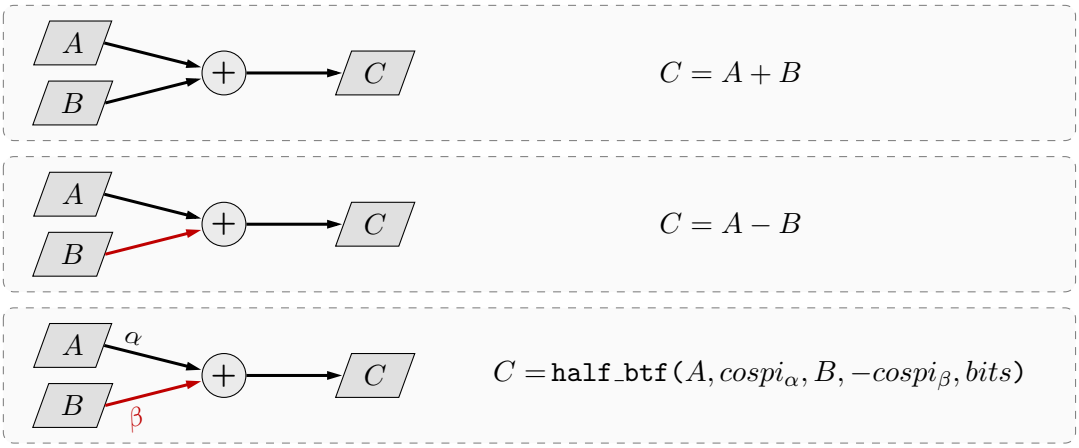


Figure 3.3: Graphical aid for Figures 3.5 and 3.6.

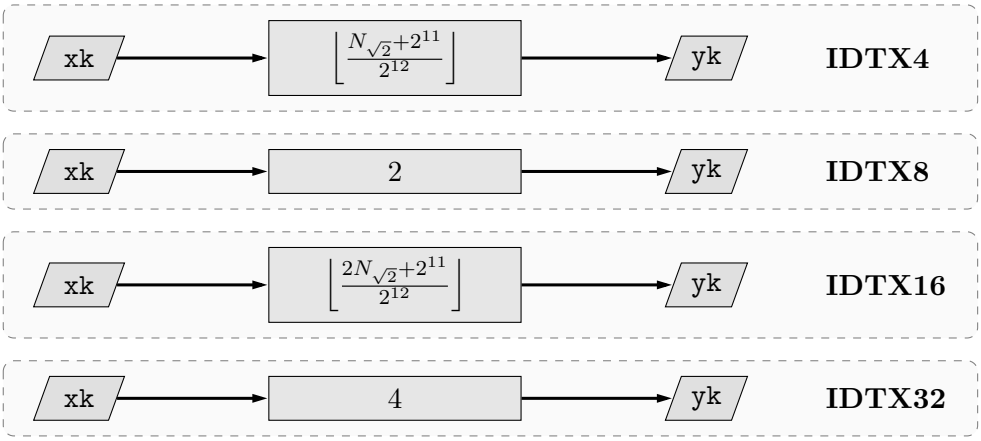


Figure 3.4: Description of the Identity transforms in *libaom*.

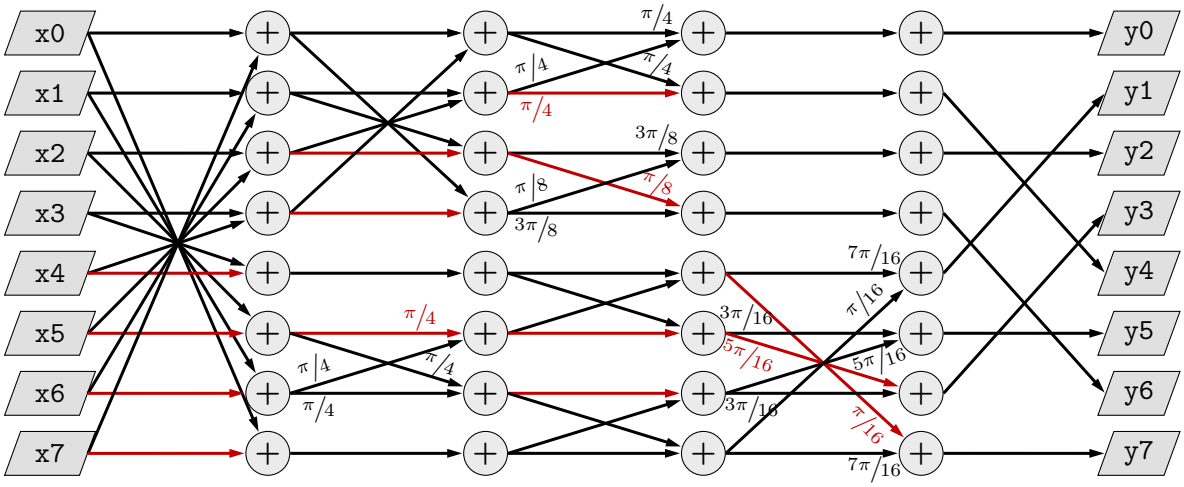


Figure 3.5: Block diagram of *libaom*'s Integer DCT.

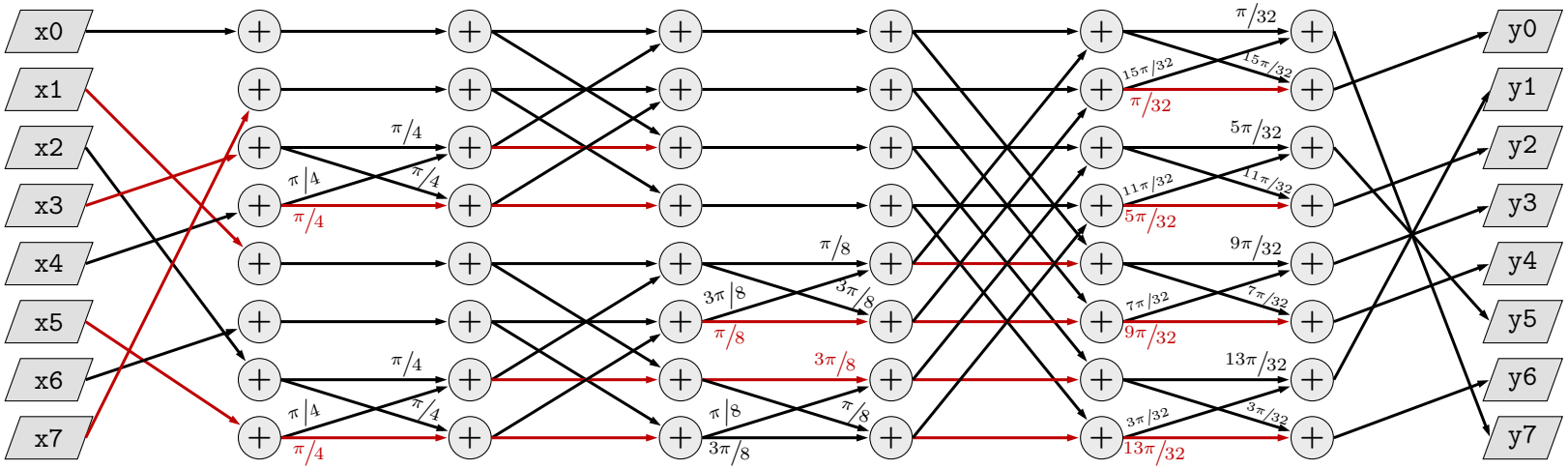


Figure 3.6: Block diagram of *libaom*'s Integer ADST.

3.4.2 Performance and Statistics Analysis

From the obtained understanding of the tools and characteristics of the *Transform* stage, various tests were performed in order to gather information on what were the most commonly used options, and their corresponding impact on encoding performance.

With these tests, it was intended to get to know:

- the time spent per encode in the *Transform* stage;
- used kernels and vector sizes;
- bits used on cosine approximation.

In order to achieve this, *libaom* was modified in order to retrieve these statistics, without impacting the encoding performance. As an additional measure, the timing tests were performed independently from the others, in order to get an accurate result of the encoding performance, since the writing of the transformation options would cause a major impact to the overall duration of the encode.

The tests were performed on different sequences with various resolutions and encoding difficulties. These were obtained in *Xiph*'s test media [17], and are presented in Table 3.2.

Table 3.2: Sequences used for testing.

Resolution			**Sequence Name**
Label	**Height**	**Width**	
CIF	288	352	*Waterfall* *Flower* *Bridge Close*
HD	720	1280	*Ducks take off* *Parkrun* *Shields*
FHD	1080	1920	*Parkjoy* *Dinner* *Factory*
UHD	2160	3840	*Into tree* *Old Town Cross* *Crowd Run*

As mentioned previously in Section 2.1.3.1, the definition of the encoding objectives greatly impacts the performance of the encoder, both in terms of compression gain, and obtained video quality. These objectives are typically defined according to a certain performance or quality metric. *AV1* reference encoder, *aomenc*, provides four different metrics, corresponding to constant or constrained quality (*q* and *cq*) and constant or varying bitrate (*cbr* and *vbr*). These options allow for the easy adaptation of each encode to the corresponding use case. For instance, for a network application, a certain bit rate objective, or range, is more adequate

than the definition of a certain quality, since in this case the obtained bit rate may depend on the scene.

For the performed tests, the variation of used tools and performance was evaluated by varying the desired objective on a constant quality mode, which is controlled through a subjective parameter, `cq-level`. This may vary between 0 and 63, corresponding the latter to the lowest quality encode. Three different quality parameters were tested, *60, 25,* and *5*, corresponding to a *Low*, *Medium* and *High* quality sets, respectively.

Besides the quality objectives, the encoder was also configured to use the highest computing power, `cpu-used=8`, as well as a single pass encode, `passes=1`. These options were set in order to get the lowest encoding times, since their impact on the obtained quality did not justify the added complexity.

The resulting command for configuring *aomenc* for the developed tests, encoding the first 10 frames of each video, becomes

```
./aomenc <INPUT-FILE> -h <HEIGHT> -w <WIDTH> -o <OUTPUT-FILE> --limit=10 -p 1 --
    cpu-used=8 --i420 --q-hist=64 --end-usage=q --cq-level=<CQ-LEVEL>
```

The results presented in the following sections are derived from the test encodes, which were performed on a *Ryzen 7 2700*, clocked at 3.9GHz.

3.4.2.1 Timing Analysis

On the performed *AV1* timing analysis, the main aspect to evaluate was the impact of the *Transform* stage on the total encoding time, for different quality thresholds.

For a certain resolution, it is expected that the time spent on this stage remains approximately constant, regardless of the desired quality. Therefore, the higher the total encoding time, the least impact the *Transform* stage would have.

These results were verified in the performed tests, and are represented in Figure 3.7.

In this Figure, each tall bar corresponds to the average time spent per encode, in each of the resolutions, varying the desired quality objective. The corresponding smaller bar represents the percentage of time spent during transformation, also expressed numerically by the number on top of each bar.

As anticipated, the desired quality of each encode plays an important role in the necessary total time, majorly considering the *Low* to *Medium* quality objectives, since the latter, in average, spends *double* the time of the former. However, once considered the *High* quality set, the encoder takes 14% more time to encode the same sequence on a *Medium* quality objective.

Although these results represent an interesting analysis from a performance standpoint, for the focus of this work, the most relevant analysis comes from the percentage of time spent during the *Transform* stage. As expected, this time stays roughly the same, independently of the quality objective. However its impact to the total encoding time decreases as the quality increases. On average, the encode spends 17.3% on the transformation.

This aspect leads to conclude that there is a necessity to develop fast and efficient architectures for the *Transform* block, since it corresponds to a relevant percentage of the total encode time, regardless of the quality. And although lower quality encodes could benefit more of such improvements when comparing to higher qualities, both cases would gain, since such architecture could be used on a high variety of encoders.

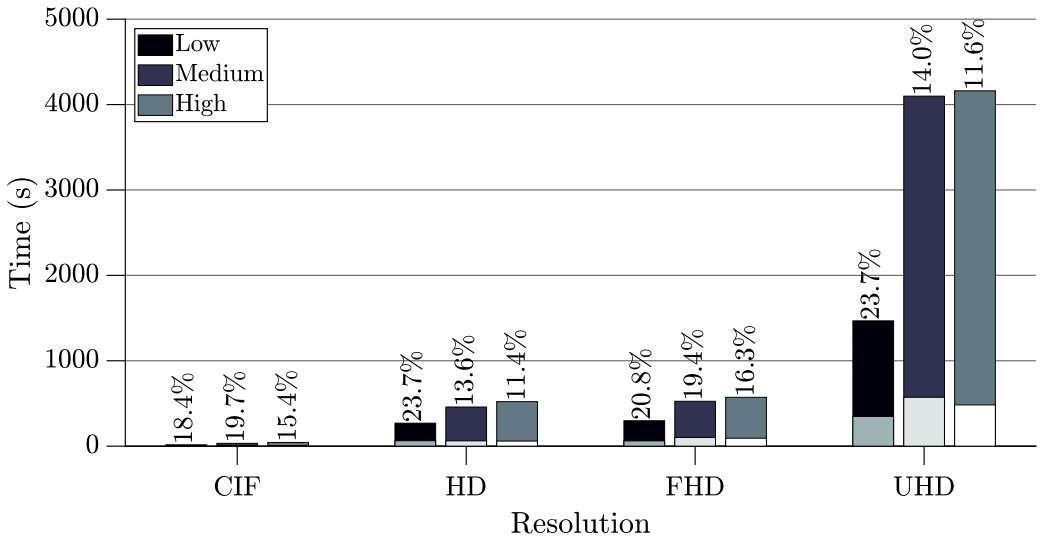


Figure 3.7: Average encoding and transform time per resolution, on different quality objectives (dark colours represent total encoding time, while lighter are the respective time spent on *Transform* stage).

3.4.2.2 Configuration Set Analysis

This analysis is divided in different segments, dedicated to each of the transform options analyzed. Some of which, although not entirely relevant for the aim of this work, may prove useful for the implementation of different architectures.

On the distribution of used kernels verified in Figure 3.8, the most relevant aspect is the clear dominance of the *DCT* among the others. However, as quality increases, the distribution starts to spread out.

A contrary behavior is verified on the transform vector size. As seen in Figure 3.9, as quality increases, the smaller vector sizes (namely the 4 length vector), get used more frequently. This is easily understandable, as smaller blocks present lower losses during the *Quantization* stage.

Although *AV1* supports asymmetric transform blocks, in Figure 3.10 it is possible to verify that the encoder, in most of the block transformations, does not take advantage of such, using square blocks, as well as the same kernel for both directions. This behavior remains similar throughout the different resolutions.

This aspect may prove as a starting point for improving the *Transform* stage, since, as mentioned previously, symmetric transforms may be implemented with *Matrix Multiplication (MM)*. Therefore, using fast *MM* architectures for symmetric blocks, and complementing with the algorithm implemented in *libaom* for asymmetric blocks, the transform time may decrease.

The final analyzed transform option is the number of bits used in the cosine representation. In Figure 3.11, the distribution is represented for the different quality objectives.

Various conclusions can be derived from this data. Firstly, as expected, as quality increases, so does the number of bits for cosine representation, as seen by the increase of the

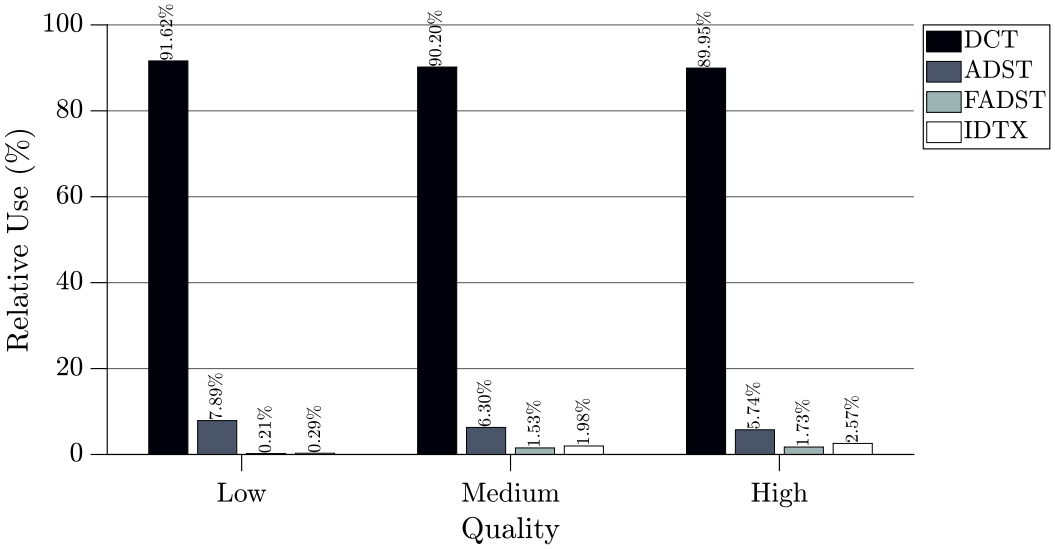


Figure 3.8: Average distribution of used kernels, for all resolutions, according to the quality threshold.

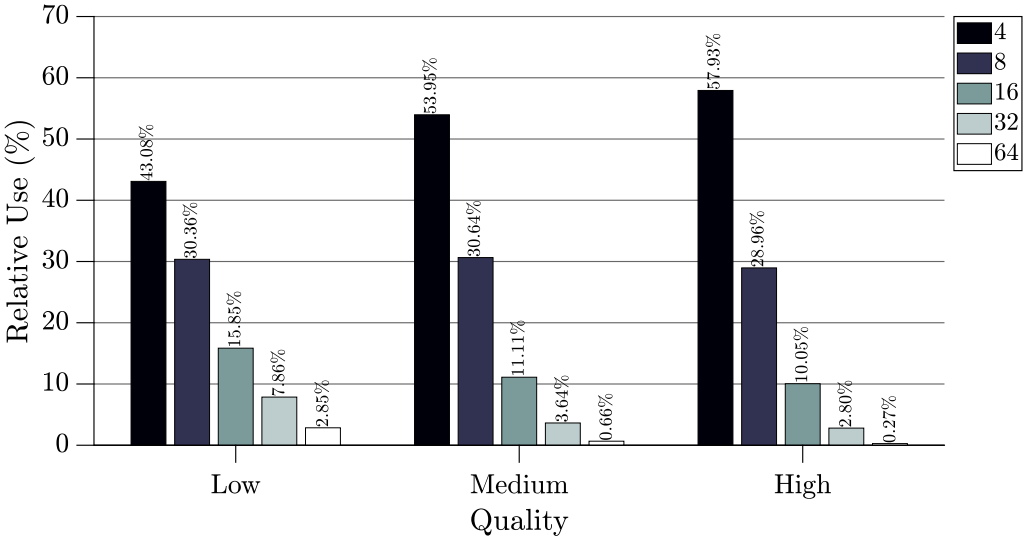


Figure 3.9: Average distribution of vector sizes, for all resolutions, according to the quality threshold.

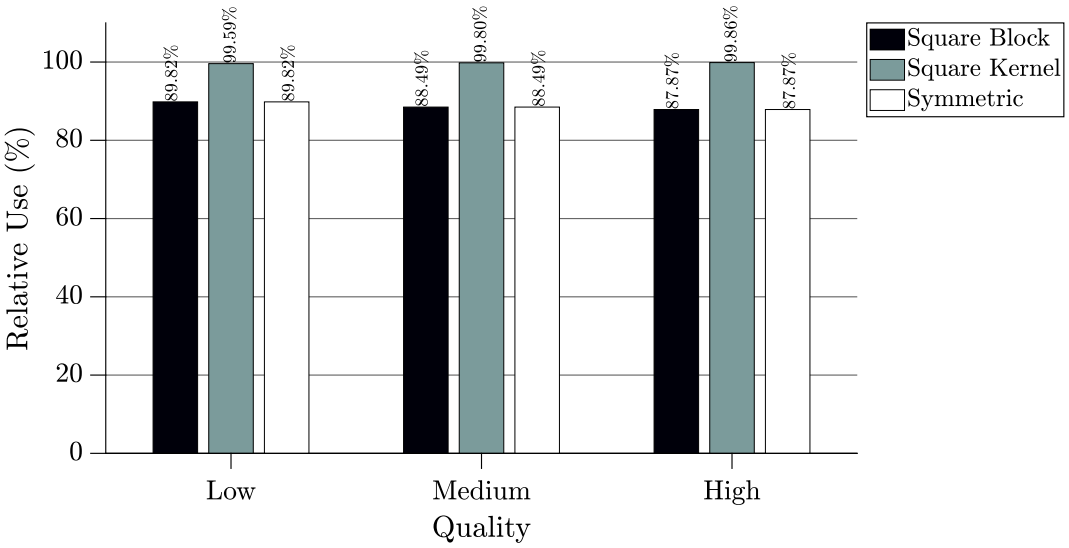


Figure 3.10: Use of square blocks, same kernel for rows and columns, and symmetric kernels, according to the quality threshold.

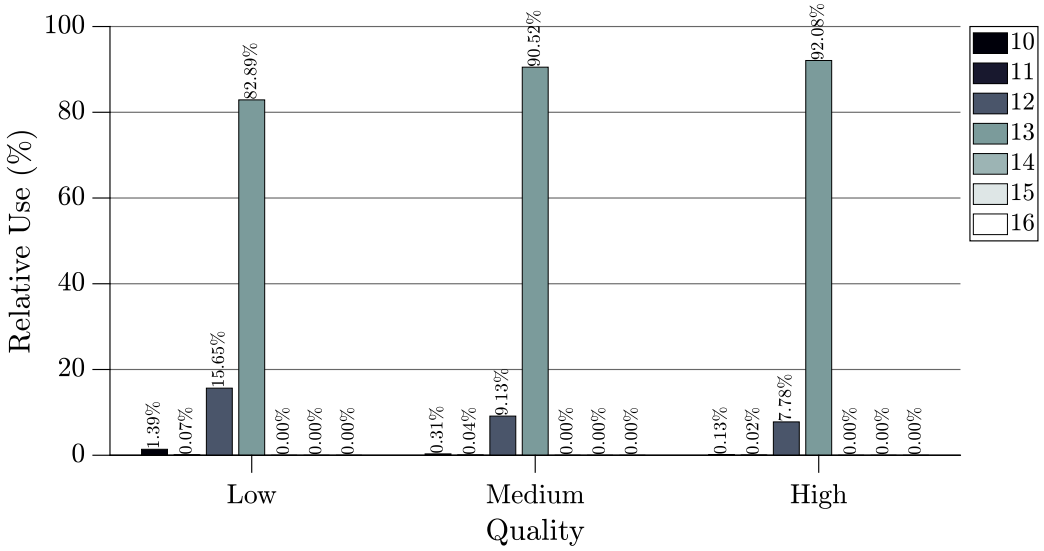


Figure 3.11: Different number of bits used on the cosine approximations, throughout different quality sets.

percentage on the *13 bit* representation.

However, most representations hardly get used, since most of the transformations use *13, 12, 10* and, very infrequently, *11 bits*, without using any of the higher representations (on the performed tests).

Analyzing this information, it may be thought that the number of bits used in the cosine contributes for the overall quality of the compressed sequence. In the following section, this hypothesis is tested.

3.4.2.3 Quality Analysis

In this test, besides evaluating the obtained quality for each tested `cq-level`, the impact of the number of bits in cosine representation also was measured.

To evaluate the impact of the number of bits used in the cosine approximations, *aomenc* was modified to force either 10 or 16 bits throughout the encoding operation, for both forward (T) and inverse transformations (T^{-1}). *Aomdec* (reference decoder) was not modified, since it acts according to the specified *Bitstream Decoding Format* [18]. Nonetheless, it uses 12 bit representation, regardless of the choices made in the decoder.

From the gathered reconstructed sequences, $\underset{*}{\mathbf{G}}$, the Peak Signal to Noise Ratio (PSNR) of each one was calculated, according to Equation 3.35.

$$PSNR = 10\ log_{10}\left(\frac{255^2}{MSE}\right) \tag{3.35}$$

MSE corresponds to the *Mean Squared Error* of the reconstructed video. Considering a single $M \times N$ monochrome frame, this error is given by Equation 3.36.

$$MSE = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}\left(\mathbf{G}_{\mathbf{x,y}} - \underset{*}{\mathbf{G}}_{\mathbf{x,y}}\right)^2 \tag{3.36}$$

However, since the test revolves around a sequence of K reconstructed frames, with three chroma channels per bit, c, MSE becomes

$$MSE = \frac{1}{3KMN}\sum_{k=0}^{K-1}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}\sum_{c=0}^{3-1}\left(\mathbf{G}_{\mathbf{k,x,y,c}} - \underset{*}{\mathbf{G}}_{\mathbf{k,x,y,c}}\right)^2 \tag{3.37}$$

The workflow of the performed test is represented in Figure 3.12.

The average results from all resolutions, for each of the quality objectives is represented in Figure 3.13.

As expected, as `cq-level` increases, so does the obtained quality. Also, considering the encoding time differences verified in Section 3.4.2.1, the smaller PSNR gap between *Medium* and *High* qualities was also expected. However, the difference between these two parameters depends on the encoded video, as shown in Figure 3.14, where the difference between *Medium* and *High* encodes is $2dB$.

The differences between obtained PSNRs can be easily explained through analysis of the *Quantization* stage in each quality objective. Looking at the distribution of the *Quantizer* throughout the different `cq-level`'s (Figure 3.15), it is possible to verify that this stage deeply adapts to the desired quality objective, increasing QP for lower qualities.

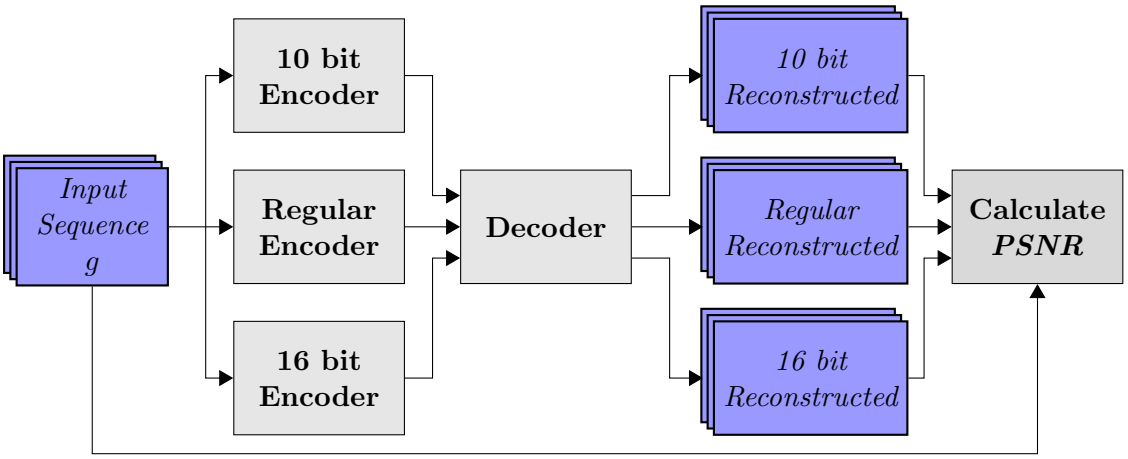


Figure 3.12: Description of the test for comparing impact of number of bits in cosine approximations.

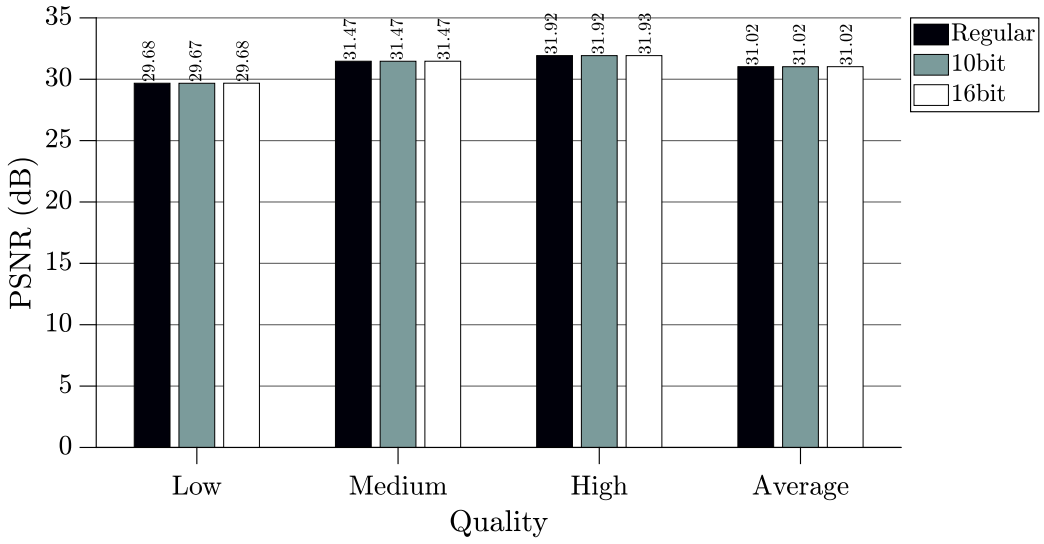


Figure 3.13: Obtained quality for each of the quality objectives, and comparison with different cosine bits approximation.

(a)

(b)

(c)

Figure 3.14: Detail of *Parkjoy* encodes, through different quality objectives (`cq-level =` (a) - 60; (b) - 25; (c) - 5).

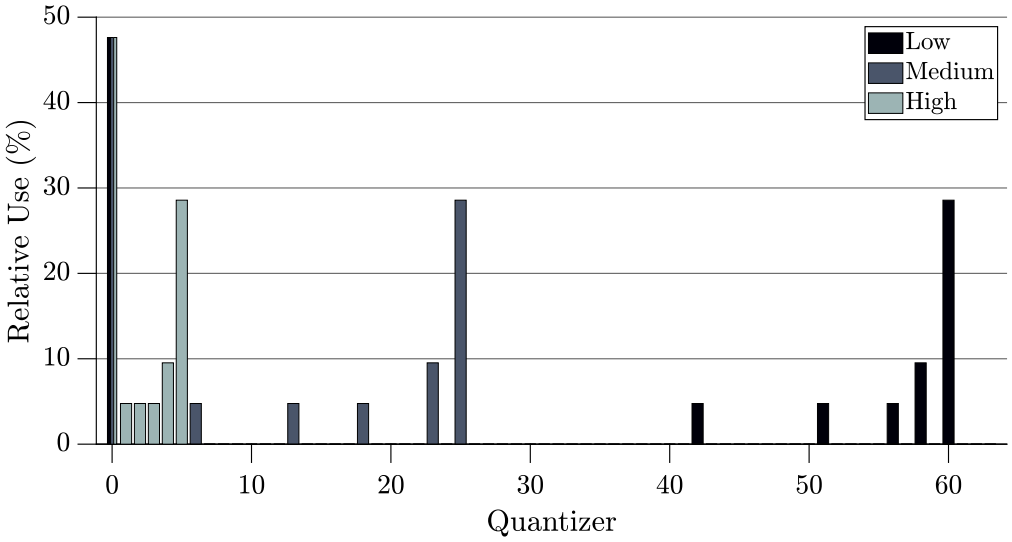


Figure 3.15: *Quantizer* distribution on different quality objectives.

However, the differences in the used QP's do not justify the increased encoding time throughout the different quality objectives, since the *Quantization* stage's complexity should not vary too much depending on the desired quality, similarly to what happens on the *Transform* stage. The time difference is mainly caused in the higher complexity blocks, the *Inter* and *Intra Prediction* stages, as the encoder adapts the processes in these blocks depending on the desired quality. For instance, there is no need for the encoder to make a exceptionally precise prediction, when most of the transform coefficients are discarded.

Considering now the obtained quality for each of the three different encodes (*Regular*, *10 bit* and *16 bit*), it is possible to observe that the number of bits on cosine approximation does not contribute to the obtained quality, regardless of the desired objective, contrary to what was verified in Figure 3.11. Accordingly, it would be safe to assume that the cosine approximations could be fixed on a certain number of bits, without major impact to the video quality.

This factor presents a major point for exploring faster *Transform* block architectures. The architectures in Figures 3.5 and 3.6 are highly dependent on the `half_btf` function (equation 3.33), which is adaptable to the number of bits used for cosine, in each block transformation. However, with the use of a fixed number of bits, this function could be simplified, since the multiplications and divisions performed in it could be implemented with a fixed number of shifts and additions.

References

[1] Yun Qing Shi and Huifang Sun. *Image and Video Compression for Multimedia Engineering: Fundamentals, Algorithms, and Standards*. 2. ed. Image Processing Series. Boca Raton, Fla.: CRC Press, 2008. ISBN: 978-0-8493-7364-0.

[2] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Fourth edition. New York, NY: Pearson, 2018. ISBN: 978-0-13-335672-4.

[3] N. Ahmed, T. Natarajan, and K. R. Rao. "Discrete Cosine Transform". *IEEE Transactions on Computers* C-23.1 (Jan. 1974), 90–93. ISSN: 0018-9340. DOI: 10.1109/T-C.1974.223784.

[4] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. 2nd ed. Upper Saddler River, NJ: Prentice Hall, 1998. ISBN: 978-0-13-754920-7.

[5] *Discrete Cosine Transform - MATLAB Dct*. https://www.mathworks.com/help/signal/ref/dct.html.

[6] William K. Pratt. *Digital Image Processing: PIKS Inside*. 3. ed. A Wiley-Interscience Publication. New York: Wiley, 2001. ISBN: 978-0-471-37407-7.

[7] Jingning Han, Yaowu Xu, and Debargha Mukherjee. "A Butterfly Structured Design of the Hybrid Transform Coding Scheme". *2013 Picture Coding Symposium (PCS)*. San Jose, CA, USA: IEEE, Dec. 2013, 17–20. ISBN: 978-1-4799-0294-1 978-1-4799-0292-7. DOI: 10.1109/PCS.2013.6737672.

[8] Benny Bing. *Next-Generation Video Coding and Streaming*. Hoboken: Wiley, 2015. ISBN: 978-1-119-13332-2 978-1-119-13333-9.

[9] Soo-Chang Pei and Jian-Juin Ding. "The Integer Transforms Analogous to Discrete Trigonometric Transforms". *IEEE Transactions on Signal Processing* 48.12 (Dec. 2000), 3345–3364. ISSN: 1053-587X. DOI: 10.1109/78.886998.

[10] S. Wolter et al. "Parallel Architectures for 8*8 Discrete Cosine Transforms". *[Proceedings] 1992 IEEE International Symposium on Circuits and Systems*. Vol. 1. San Diego, CA, USA: IEEE, 1992, 149–152. ISBN: 978-0-7803-0593-9. DOI: 10.1109/ISCAS.1992.229992.

[11] Yonghong Zeng et al. "Integer DCTs and Fast Algorithms". *IEEE Trans. Signal Process.* 49.11 (Nov./2001), 2774–2782. ISSN: 1053587X. DOI: 10.1109/78.960425.

[12] Wen-Hsiung Chen, C. Smith, and S. Fralick. "A Fast Computational Algorithm for the Discrete Cosine Transform". *IEEE Transactions on Communications* 25.9 (Sept. 1977), 1004–1009. ISSN: 0090-6778. DOI: 10.1109/TCOM.1977.1093941.

[13] Sun Song and Qi Haibing. "A Pipelining Hardware Implementation of H.264 Based on FPGA". *2010 International Conference on Intelligent Computation Technology and Automation*. Vol. 1. May 2010, 299–302. DOI: 10.1109/ICICTA.2010.401.

[14] Pankaj Kumar Srivastava and Prof. Anil Kumar Jakkani. "FPGA Implementation of Pipelined 8×8 2-D DCT and IDCT Structure for H.264 Protocol". *2018 3rd International Conference for Convergence in Technology (I2CT)*. Apr. 2018, 1–6. DOI: 10.1109/I2CT.2018.8529352.

[15] G Ravi Teja et al. "Verilog Implementation of Fully Pipelined and Multiplierless 2D DCT/IDCT JPEG Architecture". *2015 Online International Conference on Green Engineering and Technologies (IC-GET)*. Nov. 2015, 1–5. DOI: 10.1109/GET.2015.7453819.

[16] P Subramanian and A Sagar Chaitanya Reddy. "VLSI Implementation of Fully Pipelined Multiplierless 2D DCT/IDCT Architecture for JPEG". *IEEE 10th INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING PROCEEDINGS*. Oct. 2010, 401–404. DOI: 10.1109/ICOSP.2010.5657181.

[17] *Xiph.Org :: Test Media*. https://media.xiph.org/.

[18] *AV1 Bitstream & Decoding Process Specification*. https://aomediacodec.github.io/av1-spec/av1-spec.pdf. 2019.

CHAPTER 4

Developed Architectures

4.1 Software Implementations

The previous chapter presented some characteristics of the current state of *libaom*'s *Transform* stage which might compromise its performance, the most relevant being the unnecessary flexibility in the representation of cosine approximations.

In order to undertake these opportunities, and improve the overall encoder performance, new architectures for the studied stage were developed. The first approach was to study possible simplifications of the reference software, through the development and testing of alternative approaches for the provided functions.

The developed implementations tackled the forward DCT, as it was the *kernel* that would have the highest impact on encoder performance. As the $IDCT$ is shared between encoder and decoder, and due to the added complexity, no changes were done to this block, as it acts with accordance with the established standard, as mentioned previously.

All the developed architectures and corresponding tests were written in C programming language, as to maintain the simple integration into *libaom*.

4.1.1 Matrix Multiplication Implementation

The first test was the application of the simplest integer DCT, done by the multiplication of the input vector by a scaled up version of the transform matrix, $\mathbf{F}$, firstly shown in Equation 3.25.

The original integer transform matrix is shown in Equation 4.1.

$$\mathbf{F}_{x,u} = \beta(u)\cos\left(\frac{(2x+1)u\pi}{2L}\right),\ 0 \leq u, x < L$$

$$\Downarrow$$

$$\mathbf{F} = \sqrt{\frac{2}{L}}\begin{bmatrix} \sqrt{\frac{1}{2}} & \sqrt{\frac{1}{2}} & \cdots & \sqrt{\frac{1}{2}} \\ \cos\left(\frac{\pi}{2L}\right) & \cos\left(\frac{3\pi}{2L}\right) & \dots & \cos\left(\frac{(2(L-1)+1)\pi}{2L}\right) \\ \vdots & \vdots & \ddots & \vdots \\ \cos\left(\frac{(L-1)\pi}{2L}\right) & \cos\left(\frac{3(L-1)\pi}{2L}\right) & \dots & \cos\left(\frac{(2(L-1)+1)(L-1)\pi}{2L}\right) \end{bmatrix} \tag{4.1}$$

As mentioned previously, the floating point coefficients bring a number of disadvantages on a hardware implementation, from increased calculation overheads, to encoder/decoder mismatches.

In order to address these problems, a scale and rounding operation was performed, as shown in Equation 4.1, where K represents the number of bits of the scaled coefficients.

$$\lfloor \mathbf{F}_K \rceil = \left\lfloor 2^K \mathbf{F} \right\rceil \tag{4.2}$$

However, due to the rectangular block sizes allowed in *AV1*, the factor $\sqrt{2/L}$ is not considered in the kernels themselves. Instead, the transformed outputs get scaled at a later stage. This way, the implemented transform matrix is

$$\begin{aligned} \lfloor \mathbf{F}_K \rceil &= \left\lfloor 2^K \sqrt{\frac{L}{2}} \mathbf{F} \right\rceil \\ &= \left\lfloor 2^K \begin{bmatrix} \sqrt{\frac{1}{2}} & \sqrt{\frac{1}{2}} & \cdots & \sqrt{\frac{1}{2}} \\ \cos\left(\frac{\pi}{2L}\right) & \cos\left(\frac{3\pi}{2L}\right) & \cdots & \cos\left(\frac{(2(L-1)+1)\pi}{2L}\right) \\ \vdots & \vdots & \ddots & \vdots \\ \cos\left(\frac{(L-1)\pi}{2L}\right) & \cos\left(\frac{3(L-1)\pi}{2L}\right) & \cdots & \cos\left(\frac{(2(L-1)+1)(L-1)\pi}{2L}\right) \end{bmatrix} \right\rfloor \end{aligned} \tag{4.3}$$

This way, the transformed outputs are calculated through

$$\vec{\mathcal{G}} = [\lfloor \mathbf{F}_K \rceil \vec{g}] >> K \tag{4.4}$$

For an L length vector, the calculation of the transformed vector implies L^2 additions and L^2 multiplications, which leads to the main disadvantage of such implementation. For larger vectors, this operation becomes too demanding in terms of memory and complexity.

One other negative aspect of such implementation is that, due to the variation of the transform matrix's coefficients, the obtained error in the rounding and scaling operation also varies with the vector size. The quantization[1] error, Δ_K, can be calculated as

$$\Delta_K = \frac{\max\left(\sqrt{\frac{L}{2}}\mathbf{F}\right) - \min\left(\sqrt{\frac{L}{2}}\mathbf{F}\right)}{2^K} \tag{4.5}$$

As it was proven in the previous Chapter that the number of bits in the cosine representation would not greatly impact the quality of the video, the developed architectures used 8 bits for the scaling operation, as to decrease the overhead of the implemented multiplications and shifts. The impact of this choice was evaluated at a later stage.

To evaluate the performance of this first implementation, a test was performed to measure and compare the elapsed time for both the described architecture, and the corresponding equivalent from *aomenc*. This test injected a fixed sequence of 1 million input vectors into each of the *DCT*'s, measuring the elapsed cpu time in the operation. The results are in Table 4.1.

From these it is easily observable why the encoder's implemented transforms follow the *butterfly* scheme. Although from sizes 4 to 32 the proposed implementation is faster than the current version of *libaom*, the largest transform is slower. This factor, added to the error variation from the scaling operation makes this implementation quite damaging for the overall encoder performance, especially on a constant quality objective, as shown in Table

[1]Here, quantization refers to the scaling and rounding operation, and not to the the Q stage in an encoder.

Table 4.1: Comparison of execution time between *aomenc*'s DCT and the matrix multiplication implementation.

Vector Size	**Execution Time (ms)**	
	aomenc's	**MM**
4	75	36 (−52%)
8	179	66 (−63%)
16	405	174 (−57%)
32	1039	686 (−33%)
64	3288	3590 (+9%)

4.2. Here, there are presented the timing results of an encoding test, where one encode was made with the standard *aomenc*, the other had the proposed matrix multiplication *DCT*'s. The test encoded the first 15 frames of the *Parkrun* HD sequence, with two different quality objectives. After compression, the encoded video was decoded with *aomdec*, calculating the PSNR of the output video.

Table 4.2: *aomenc*'s encoding time with original vs implemented *DCT*.

cq-level	**Measure**	**Execution Time (ms)**	
		Original	**MM**
60	**Total time (s)**	466.5	530.8
	***Trans.* time (s)**	45.0	104.2
	PSNR (dB)	32.39	32.38
5	**Total time (s)**	814.1	835.3
	***Trans.* time (s)**	60.4	98.4
	PSNR (dB)	34.88	34.86

As it is observable, to maintain a similar encoding quality, the encoder spends up to 13.8% more time per encode, making such architecture unreliable for implementation on *aomenc*.

Taking this into account, a new approach was employed, using the same *butterfly* scheme as *libaom*'s transforms.

4.1.2 Alternative *Butterfly* Implementation

AV1's reference *Transform stage* follows the aforementioned architecture for the DCT, in addition to expanding its use into the *ADST*. In this scheme's publishing paper [1], the authors gave good reasons for the heavy adoption of this implementation, claiming that "*The number of computational steps has been shown to be less than 1/6 of the conventional DCT algorithm employing a 2-sided FFT*".

This is achieved through a pipelined implementation of the previously shown matrix multiplication, where each stage is calculated as function of the previously calculated intermediary

coefficients, as shown in Figures 3.5 and 3.6. Besides the reduction of complexity, this implementation also uses a fixed bank of *cosine* coefficients (corresponding to `cospi` in *libaom*), limited between $\cos(\pi/2) < \alpha \leq \cos(0)$, i.e., $0 < \alpha \leq 1$ [1]. This way, the quantization error produced by the rounding and scaling operation is constant for all vector sizes.

To improve upon the reference *DCT*, the developed architecture implemented a similar approach to the previous *MM* architecture, using 8 bits for the scaling of the *cosine* approximations. These values, cos_{Apr}, were generated using the same method as `cospi`, i.e.

$$cos_{Apr} = \left\lfloor 2^8 \cdot \cos\left(\frac{k\pi}{128}\right) \right\rceil,\ 0 \leq k \leq 63 \tag{4.6}$$

Comparing the quantization step to the worst case for *libaom*'s *DCT*, with 10 bits

$$\Delta_{10} = \frac{1-0}{2^{10}} \approx 0.98 \cdot 10^{-3} \tag{4.7}$$

$$\Delta_{8} = \frac{1-0}{2^{8}} \approx 3.9 \cdot 10^{-3} \tag{4.8}$$

then it is possible to calculate the *Mean Squared Quantization Error* through Equation 4.9

$$\begin{gathered} MSE_{Kq} = \frac{\Delta_K^2}{12} \\ \Downarrow \\ MSE_{10q} \approx 79.5 \cdot 10^{-9} \\ MSE_{8q} \approx 1271.6 \cdot 10^{-9} = 16 \cdot MSE_{10q} \end{gathered} \tag{4.9}$$

Although this value might seem discouraging, most of the error introduced in this stage is irrelevant once considered the error created by the encoder's *Quantization* block. With that said, the impact of this approximation might compromise the encoder on higher quality (i.e., lower *Quantizer*) objectives.

The implemented *DCT* also deviates from the reference software in the multiplication of the cosine coefficients. The latter version calculated most of the intermediary functions with the `half_btf` function (Equation 3.33), which, because the additional $2^{(K-1)}$, performs a *rounding* ($\lfloor \mathbf{x} \rceil$) operation, as shown in Equation 4.10

$$\begin{gathered} \texttt{half\_btf(w0, in0, w1, in1, K)} \equiv (w_0 \cdot in_0 + w_1 \cdot in_1 + (1 << (K-1))) >> (K) \\ \Downarrow \\ \left\lfloor \frac{w_0 \cdot in_0 + w_1 \cdot in_1}{2^K} \right\rceil \end{gathered} \tag{4.10}$$

On the implemented architecture, the rescaling did not include the additional factor, and was made just with the right shifting ($x >> K$) by 8 bits, corresponding to the *flooring* operation ($\lfloor \mathbf{x} \rfloor$).

The constructed *DCT* architecture underwent the first test as the previous implementation, giving origin to the results from Table 4.3.

As shown, the developed architecture is, on average, 60% faster for all vector sizes. This is due to the removal of the memory accessing overheads imposed by the access to the `cospi` array, as well the simplification of the performed operations.

[1] Zero is excluded from the set, as matrix $\mathbf{F}$ will not have any null element, for any L

Table 4.3: Comparison of execution time between *aomenc*'s DCT and the alternative *butterfly* implementation.

Vector Size	**Execution Time (ms)**	
	aomenc's	**Alternative *butterfly***
4	75	37 (−51%)
8	179	66 (−63%)
16	405	149 (−63%)
32	1039	355 (−65%)
64	3288	1362 (−58%)

However, the reliability of this implementation depends on whether the encoder can maintain the desired quality with the increased error introduced by the quantization of the *cosine* approximations. To verify this factor, the tests presented in Section 3.4.2 were repeated, once with the original encoder, and other with the encoder with the described alternative version of the *DCT*. The obtained quality and timing results are presented in Figures 4.1 and 4.2, respectively. In the latter, the dotted lines correspond to time with the original encoder. The percentage above each bar represent the time difference taken by the alternative encoder, relative to the original.

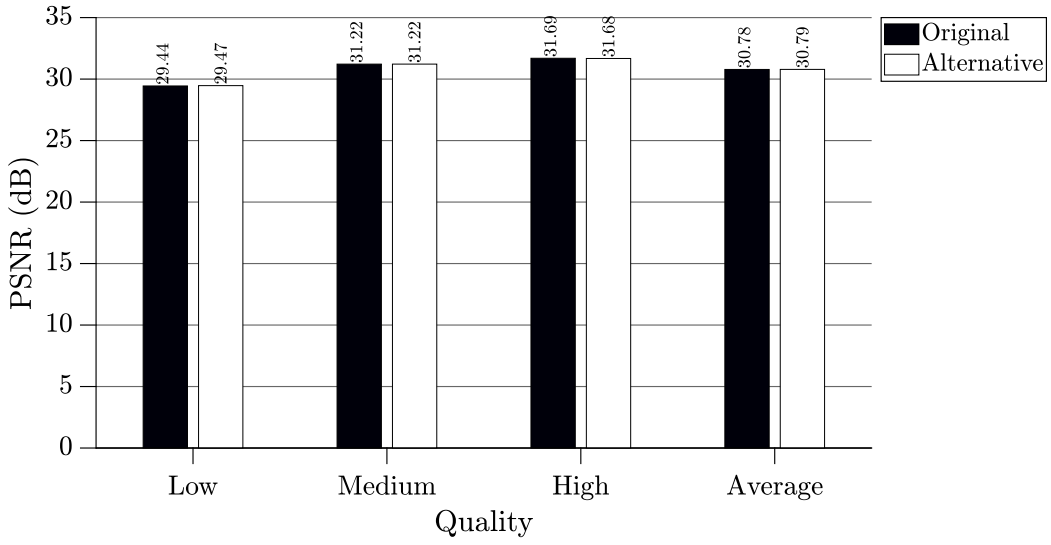


Figure 4.1: Obtained quality with original vs alternative *DCT* implementation.

As shown, with the performed changes, the encoding time was, in average, reduced by 2.9% for all quality objectives, while maintaining the output PSNR, making this a suitable *DCT* implementation for an *AV1* encoder. Although the performance improvement is rather diminishing once considered the full encoding cycle, it is a step toward a possible realtime encoding implementation.

However, the full impact of the applied changes is most noticeable on a hardware im-

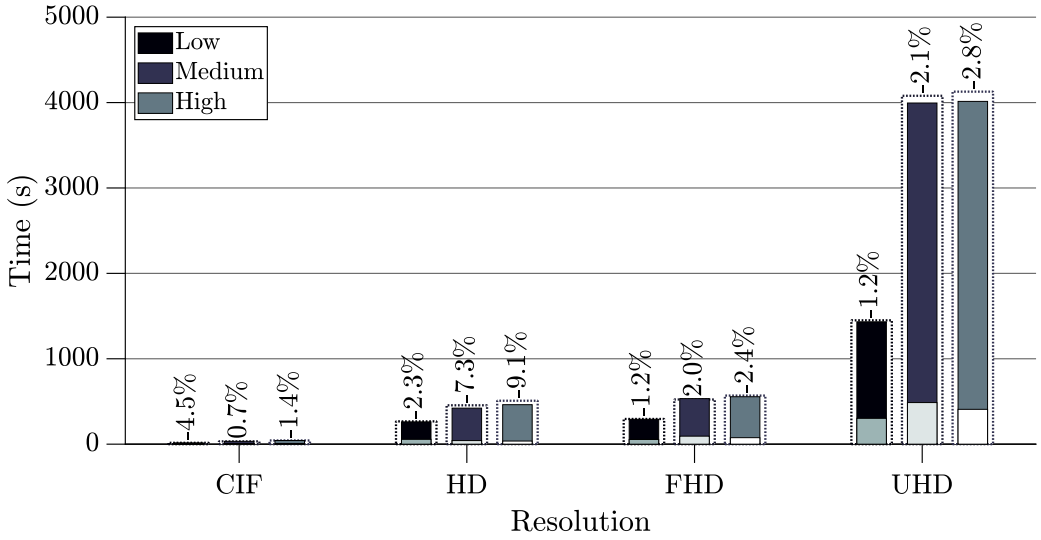


Figure 4.2: Encoding time with original vs alternative *DCT* implementation.

plementation. With the reduction of the cosine approximations, the memory used in the developed *DCT*s is highly reduced. Considering $M_{\texttt{cospi}}$ as the number of bytes (B) used for storing the original cosine approximation vector,

$$M_{\texttt{cospi}} = \frac{64 \cdot (10 + 11 + 12 + 13 + 14 + 15 + 16)}{8} = 728\,B \tag{4.11}$$

On the other hand, with the new implementation, only $64\,B$ would be needed to store these approximations, corresponding to an 81% reduction.

Nonetheless, both *libaom*'s version and the developed could heavily benefit from parallelization, as is described in the following section.

4.2 Hardware Implementations

On the subject of hardware development, Field-Programmable Gate Arrays (FPGAs) have gained massive popularity within developers. Their ease of use, added to the applicability in certain designs as caused them to gain massive popularity for both prototyping and product implementation, since they present relatively smaller *times-to-market* over other options. However, as usual, they come with compromises, namely the increased cost over specialized circuits, as well as the lower performances when compared to such [3].

Therefore, when it comes to hardware development, the best compromise would be to maintain the applicability of FPGAs in the prototyping stage, and easily migrate the tested designs into Application Specific Integrated Circuits (ASICs). This way, the first development stage could be done on a massively adaptable platform, and once concluded, the design could be implemented as a specialized design, without compromising its performance.

To achieve this, in recent years, there have been major developments in software platforms that allow for the synthesis of FPGA designs as ASICs, such as *Cadence*'s *Genus* or *Synopsys'*

Design Compiler [4, 5]. These softwares have been implemented in a variety of branches, when hardware development is necessary, namely, in video coding.

This way, for the development of this work, the chosen development platform was *Xilinx*'s *Vivado*, due to the wide availability of its FPGAs, as well as for the wide support from its community. However, there should be kept in mind that to achieve the full performance of the developed designs, a specialized hardware implementation is desired.

With an efficient algorithm for each of the supported vector sizes, the first objective was to develop a hardware architecture to implement each of the 1D DCTs individually, and group them on a single block at a later stage.

Two different architectures were developed. The first implements each of the DCT blocks individually, while the latter uses sub-blocks of each DCT, in order to achieve the final result. These architectures are explained in the following Sections.

With this approach, it was hopped to reach an architecture that englobed all the DCT kernels, allowing to easily chose between each of them, depending on the desired choices made in the beginning of the transform stage.

4.2.1 *Individual 1D DCTs* Design

The hardware implementations followed the same scheme as the corresponding software counterparts. By this it is meant that the flow of the input towards the output is done in individual and sequential stages. The main difference between these implementations is that in hardware, all of the intermediary signals within each stage are calculated in parallel.

However, in order to achieve an efficient hardware implementation, some additional measures must be taken into consideration, mainly when considering the multiplication of signals by the cosine coefficients and re-scaling.

Consider an hypothetical operation performed in the software version, where two intermediary signals, `x1` and `x2`, get multiplied by some constant, added, and finally rescaled. In C, this operation is easily described in a single line of code, as shown in Figure 4.3. However, to perform the same operation on an hardware descriptive language, some additional steps must be taken. The seemingly simple operation done in software must be deconstructed in various sequential steps, controlled by a clock signal. The operation shown in this Figure is repeated throughout the various DCT implementations in hardware, making it the key to the development of the parallel architectures.

Due to advances in *VHDL* compilers and supporting libraries, both multiplication, shifts and additions are easily described, on a similar manner to a higher level language. Although on previous generations there would be some added benefits of implementing a multiplication by shifting and adding an input, as shown in Equation 4.12, the improvements done in most recent years allow for similar architectures to be implemented, with less effort.

$$\texttt{15 * x1} \equiv \texttt{(x1<<3) + (x1<<2) + (x1<<1) + x1} \tag{4.12}$$

Taking these measures into consideration, the development of the 1D transforms becomes similar to all vector sizes. The software implementations are composed of alternating stages of simple summing operations, with more complex multiplying, sum and shift cycles. Therefore, the hardware counterparts are composed of three different blocks:

- *Summing Stages* where the inputs get added according to the previously shown butterfly schemes;

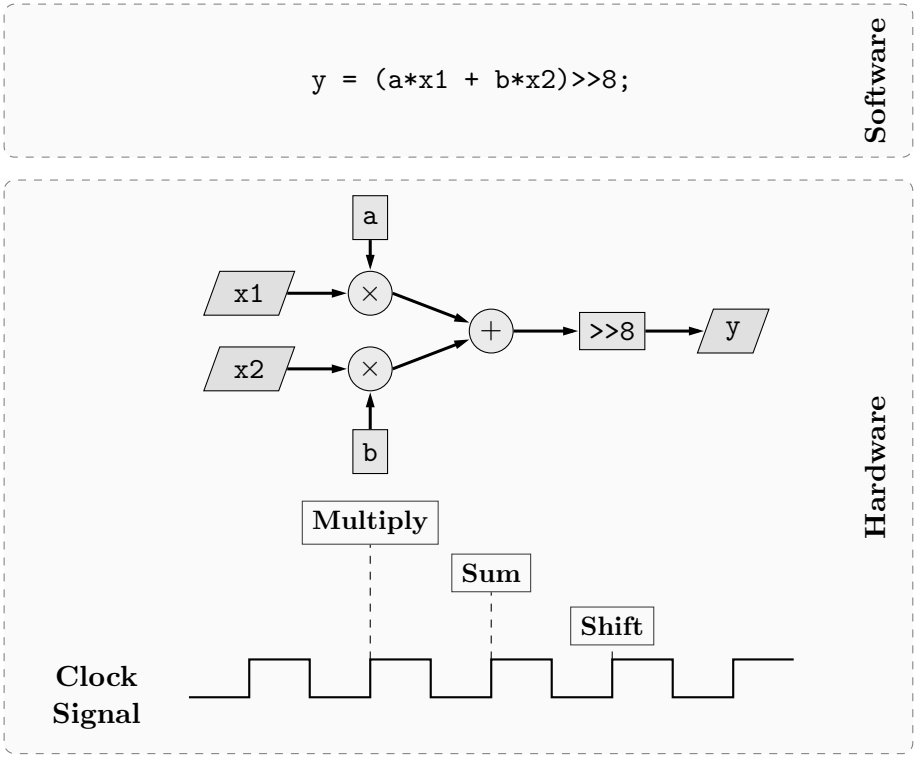


Figure 4.3: Comparison between software and hardware implementation of multiplication, sum and re-scaling.

- *Multiplier Stages*, which multiply the necessary inputs by the corresponding cosine coefficients;
- *Shift Stages* that rescale the coefficients.

Although these blocks are unique between transform sizes, and even within the same *DCT*, the operations performed within are similar between all the vector sizes.

In order to ensure the correct pipelining of the *Transform* process, each stage is controlled by an *enable* flag, `en`, which signals the start of the block's process. Once it is concluded, the block outputs an indicator, `valOut`, that acts as the enable for the following stage, creating a daisy chain of stages. The last stage's `valOut` acts as the indication of the conclusion of the *Transform* operation.

All blocks are controlled by the same *clock* and *reset* signals. The first triggers the internal processes on its ascending flank. The latter signals all internal registers to be put to 0 (its initial stage).

A simplified version of **DCT4**'s hardware implementation is represented in Figure 4.4. In here, the direction of the arrow represents if the corresponding signal is a *input* or *output*. The numbering of the output coefficients is done accordingly to the software implementation.

To simplify the development of the hardware architectures, all signals and internal registers are represented, as this measure allows to easily interconnect the developed modules. Nonetheless, all modules are configurable through the modification of a Generic parameter.

On a post-prototyping stage, to achieve an optimal utilization of resources, both inputs,

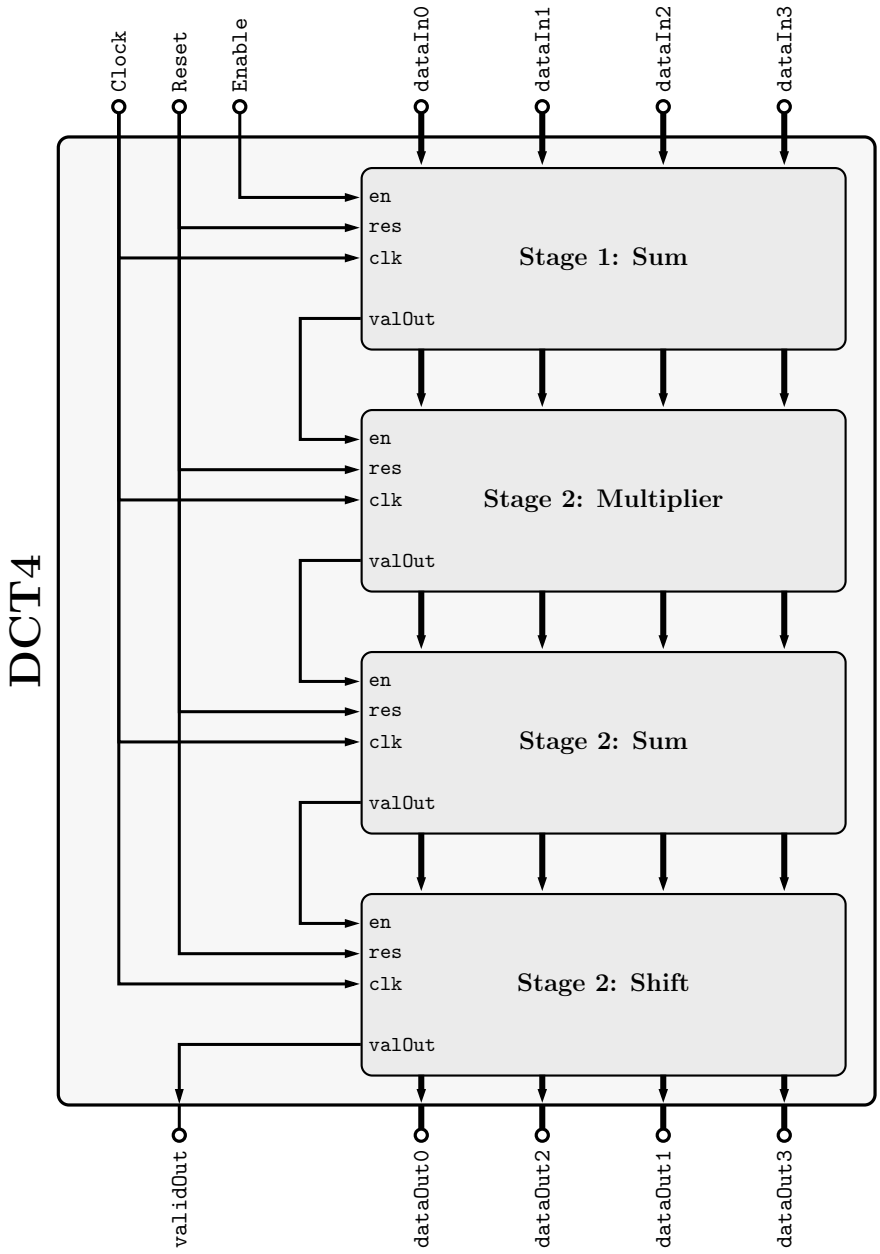


Figure 4.4: 1D **DCT4** hardware implementation.

outputs and internal signals should be shortened to the minimum length.

As a final measure to simplify the development process, the *kernels* are implemented using the smaller sizes as a constituting block. As shown in Wen-Hsiung Chen's work [1], all transform sizes greater than 4 englobe the same sequence of operations as the smaller counterparts, on one subset of its intermediary coefficients. This way, each of the smaller *1D-DCT* blocks may be inserted into the size immediately above it.

As an example, **DCT8**'s hardware implementation is represented in Figure 4.5. There, it is observable that the architecture is similarly composed of the same blocks as the previous implementation. However, after the first summing stage, the first four intermediary coefficients are input into **DCT4**.

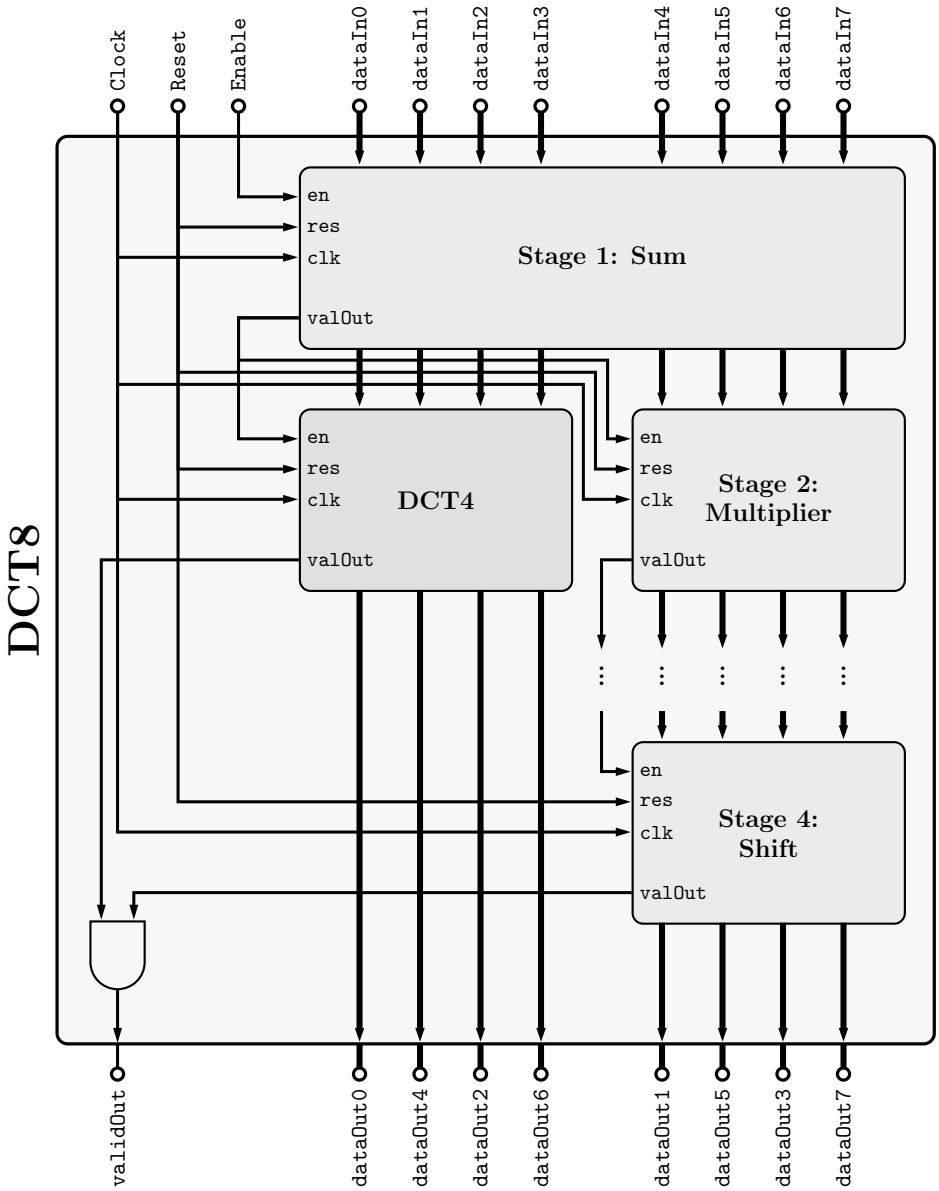


Figure 4.5: Simplified 1D **DCT8** hardware implementation, with inclusion of **DCT4**.

In the same manner, **DCT16** includes **DCT8**, which, as shown, also includes the four input version. This approach causes the smaller blocks to be repeated throughout the various larger architectures, making this approach highly inefficient from the chip's utilization standpoint. However, it brings the possibility to calculate each of the five *DCT* sizes simultaneously, which is a highly desired characteristic on an encoder. As its objective is to encode each frame in the most efficient manner, the encoder tests various options, as to find the best for the current block. This behavior is present in the various stages, including the *Transform*. Therefore, for an hardware encoder to be efficient, it must allow for the parallelization of encoding options.

One such implementation was implemented, as described in Figure 4.6.

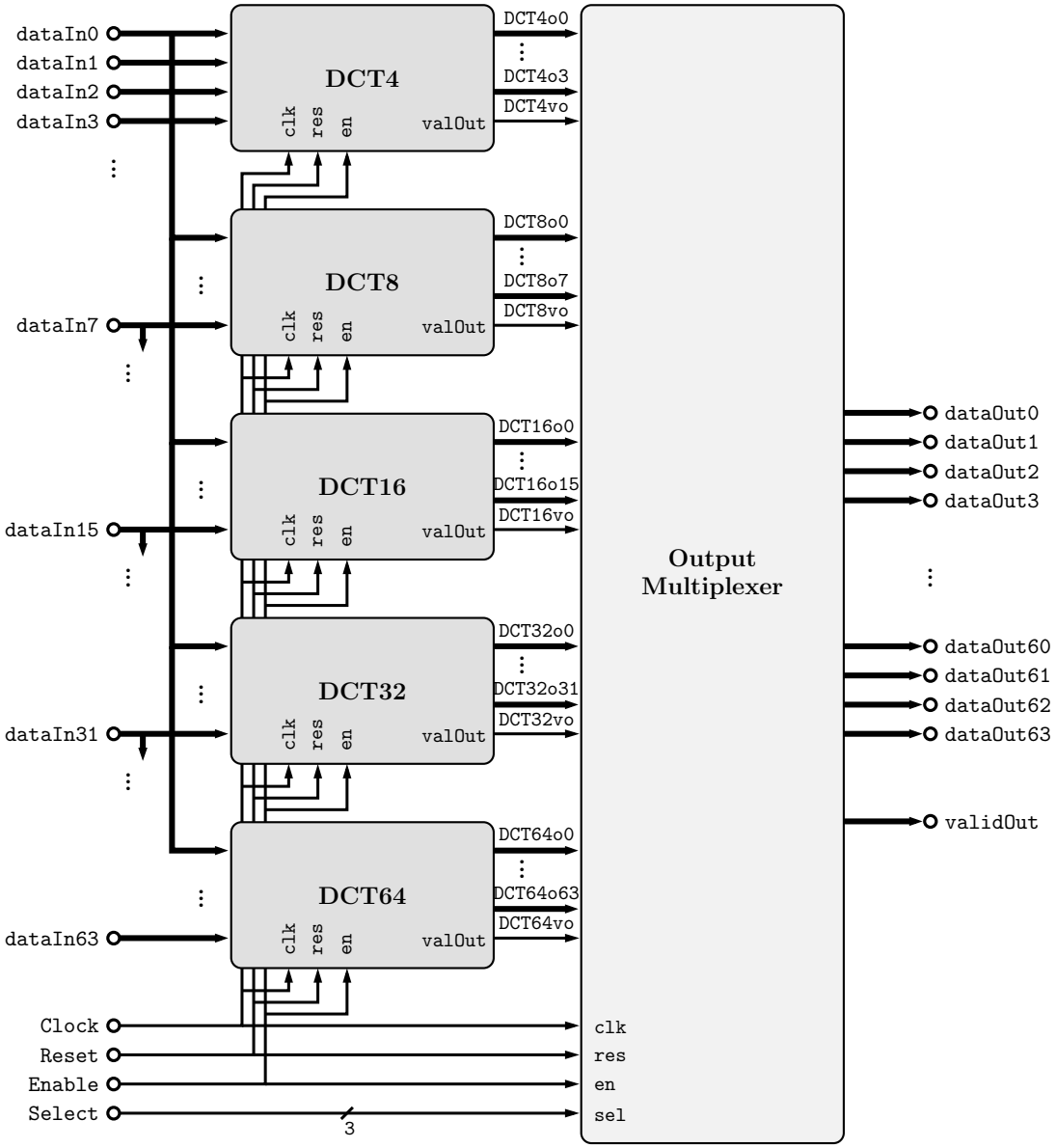


Figure 4.6: First version of the complete *DCT* wrapper.

Besides the previously shown 32 bit `dataIn`'s, `Clock`, `Reset` and `Enable`, this implemen-

tation adds an additional `Select` input.

As shown, this wrapper uses all the individual kernels independently, depending on the **Output Multiplexer** to conduct the correct output, according to the selected *DCT*.

To validate this design, a VHDL test bench was built and simulated in *Vivado*. It generates a vector of 32 bit integers, as well as the four control signals, injects them into the developed architecture, and receives the outputs. In Figure 4.7 there is represented one of the timing tests made, where the selected size was 8. It shows the internal signals for the full architecture, as well as the selected block, **DCT8**.

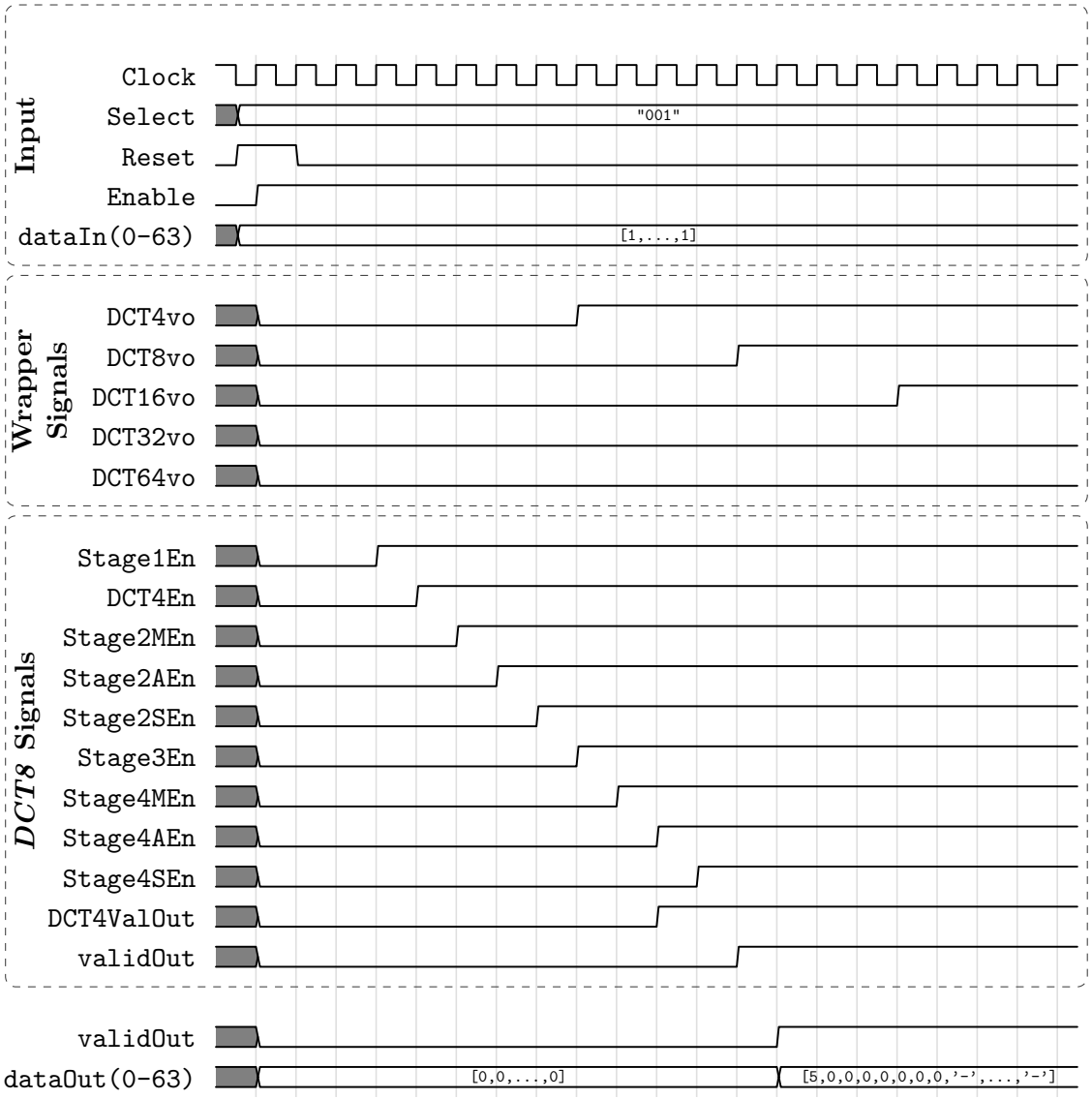


Figure 4.7: Timing diagram for a test run on the first *DCT* wrapper.

From this test, it is observable that the desired functioning of the internal stages is achieved. The input gets sequentially pipelined through the various stages, getting a valid output 10 clock cycles after the enabling of the system. This behavior was also verified for the other vector sizes, although the delay until getting the output varies from *DCT* to *DCT*,

between 6 clock cycles (**DCT4**) to 22 (**DCT64**).

Considering these results, it is possible to calculate the necessary frequency of operation, in order for this architecture to maintain a given frame rate at a specific resolution.

To obtain the necessary number of transformation blocks to process each second, the desired frame rate must be multiplied by the number of 4×4 blocks within a frame of a given resolution (as to obtain the worst case scenario). Considering then that the presented results are referring to 1D vectors of length L, the number of clock cycles to process a 2D $L \times L$ block (N_{2D}) is given by

$$N_{2D} = 2 \cdot (L \cdot N_{1D}) \tag{4.13}$$

The necessary frequency of operation for frequencies between HD and 8K at 30 fps is represented in Table 4.4.

Table 4.4: Necessary frequency of operation to obtain real-time encoding at 30 frames per second.

Resolution	**Frequency (MHz)**
1280 × 720	83
1920 × 1080	187
3840 × 2160	746
7680 × 4320	2986

With these results, several illations can be retrieved. For lower resolutions, this implementation can easily provide operable frame rates, as most devices can easily operate at 83 MHz. However, the same cannot be said of the necessary 3 GHz for 8K video. These results give an idea of the necessity to develop efficient architectures, that can take advantage of high parallelization in hardware, if high resolutions are desired. In the same manner, such architectures hardly could be implemented in FPGAs, since these usually present lower clock speeds than ASICs.

Such architectures would need to process several input vectors at once, as to process several transformation blocks simultaneously.

The developed hardware was then synthesized considering the *Artix 7* FPGA family, obtaining the utilization results from Table 4.5.

4.2.2 *Interdependent 1D DCTs* Design

This next version's main objective was the reduction of necessary resources for FPGA implementation. The strategy taken was to avoid the repetition of the internal DCT stages, i.e., the final architecture should have a single instance of each of the previously shown internal stages throughout the entire design.

To achieve this, each individual **DCTX** apart from **DCT4** was divided in sections. **DCTX_P1** is composed of the first stage of each individual kernel, corresponding to the first summing/rotation. Correspondingly, **DCTX_P2** groups all the following stages, from multiplications, sums and shifts, apart from the steps taken by the smaller $\mathbf{DCT^{X/2}}$ im-

Table 4.5: First developed architecture's utilization in number of LUTs and Registers.

DCT Size	Utilization	
	Slice LUTs	Slice Registers
4	1125	636
8	2428	2087
16	7103	5702
32	19148	14257
64	45996	34146
Wrapper	75805	58370

plementations. Figure 4.8 gives an exemplification of this sectioning, for the previous implementation of **DCT8**. All bocks are controlled by the same clock and reset signals. For simplification purposes, these signals wont be displayed in the following Figures.

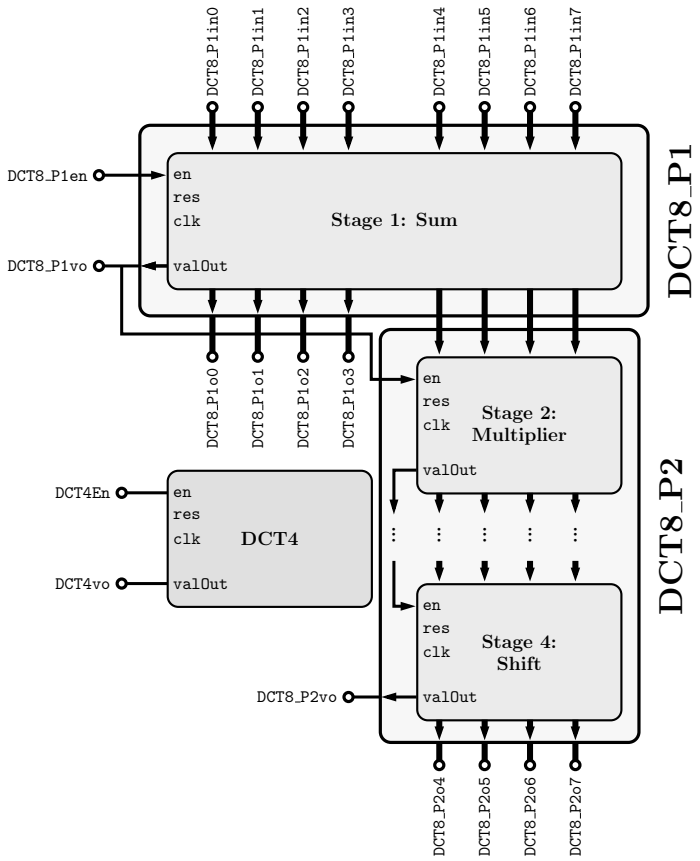


Figure 4.8: Exemplification of the individual kernel's division for the second implementation.

In Appendices B and C there are presented the implemented VHDL descriptions for both *DCT8* blocks. The following blocks follow the same architecture, varying on the multiplication coefficients and number of stages.

It is important to note that **DCT8_P2** inputs and `en` are hardwired to the bottom four outputs and `valOut` of **DCT8_P1**, respectively. Similar connections are done throughout the whole architecture, as each of the **DCTX_P2** stages will always process outputs from the corresponding **P1** stages. However, the latter may be injected with the systems `dataIn`'s, or with intermediary coefficients from previous stages.

Depending on the selected size, the input data is injected into one of the **P1** stages or directly into **DCT4**. The interconnection of the internal blocks, and corresponding flow of the intermediary coefficients is dealt by a arrangement of multiplexers, that differ from the previous implementation. In this case, this block has a much higher input/output count, as it must control which signals go into each stage, to generate the correct final coefficients as if the `dataIn`'s passed through a single **DCT** block. Besides the coefficients, also the intermediary enable signals are dependent of the central unit, since the interconnection of the `valOut/en` pairs is dependent on the selected vector size.

A simplified version of the achieved architecture is represented in Figure 4.10, where no clock or reset signals are represented. As mentioned previously, the system is composed of a single instance of each of the intermediary stages. This makes the achieved architecture similar to a single **DCT64** block from the first version. In other words, the largest block from the previous implementation had the same set of internal blocks as the newer *Wrapper*, but instantiated in a different manner. In this case, since all the internal intermediary points are accessible by the **Coefficient Multiplexer**, all the smaller transformations can be calculated with a single kernel of size 64. Figure 4.9 demonstrates this behavior. Depending on the selected vector size, the data is sequenced by different blocks, as shown by the different colored lines. The dotted lines represent the enable signals which will be activated at some point of the transformation process.

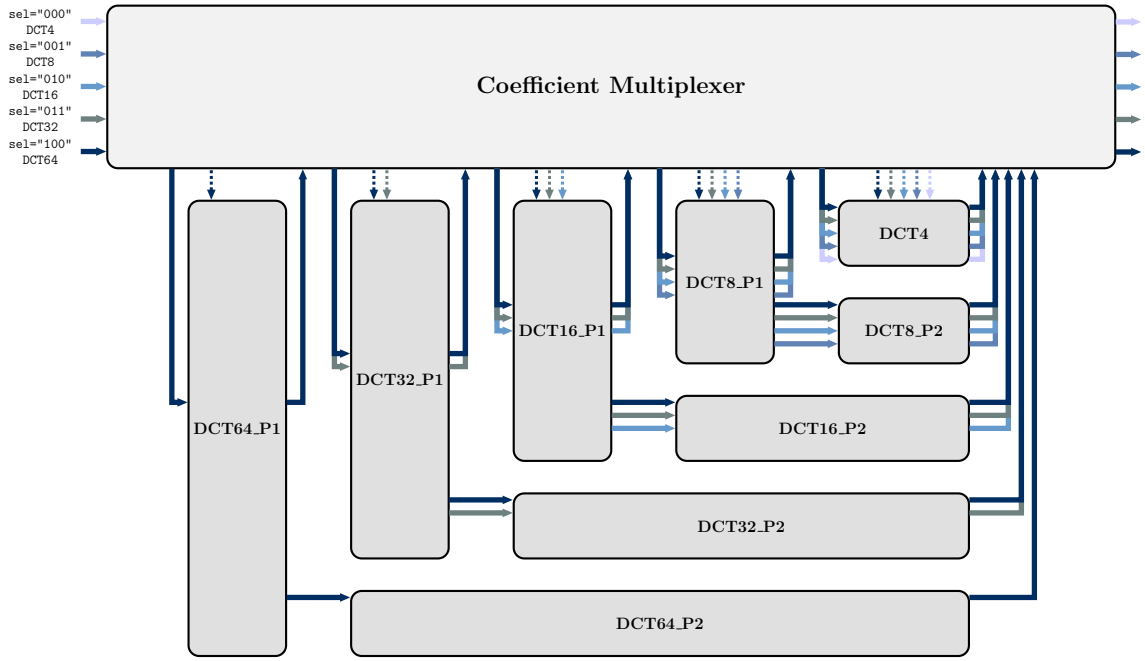


Figure 4.9: Flow of `dataIn` according to the selected vector size.

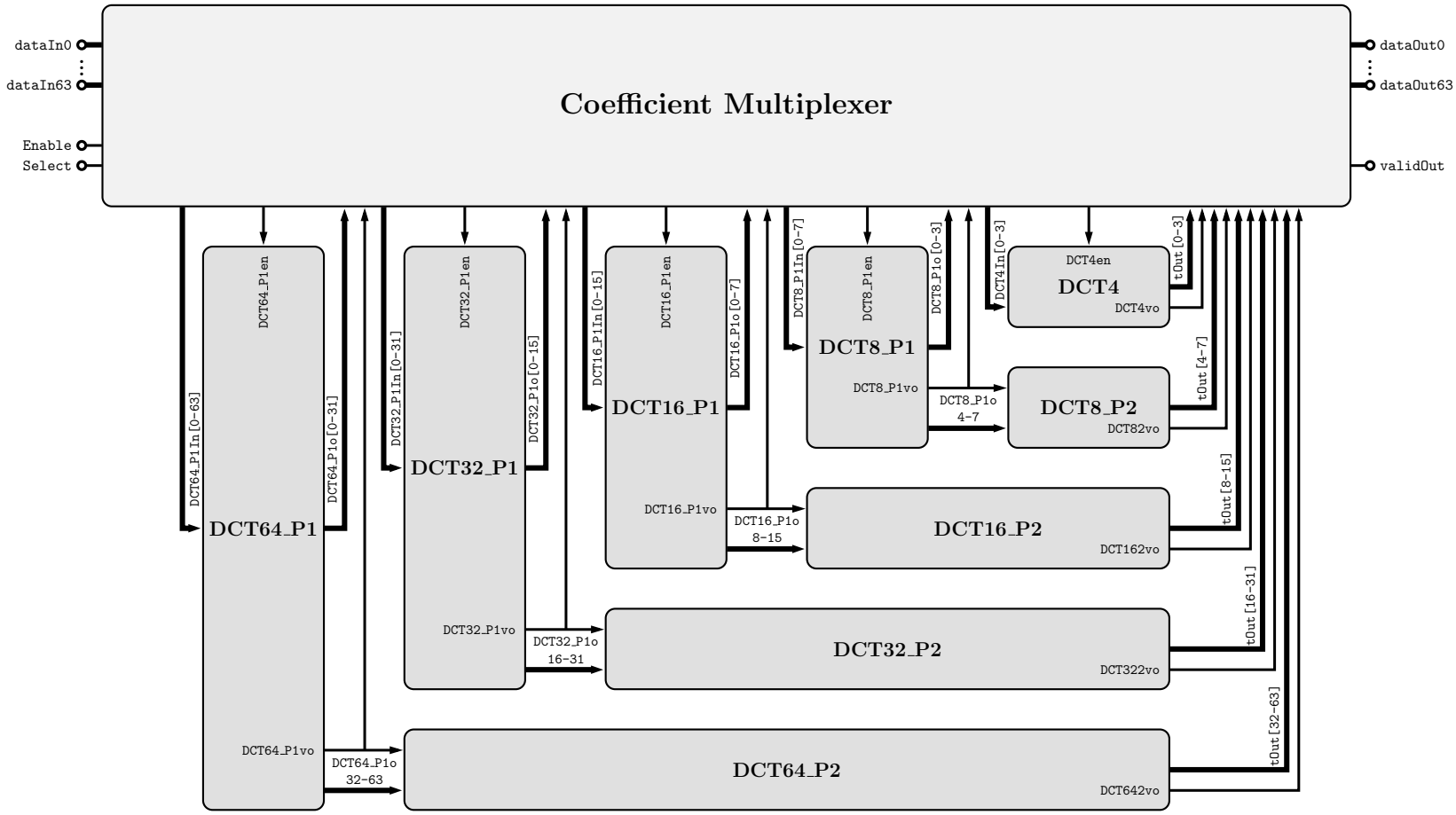


Figure 4.10: Simplified architecture of the second version of the full *DCT* wrapper.

This architecture was subjected to the same test bench as the previous version. In Figure 4.11 it is represented one of these tests, where 16 was the selected size. In this case, there are no internal signals represented, only being shown the most relevant of the ones accessible by the central multiplexer, for this case.

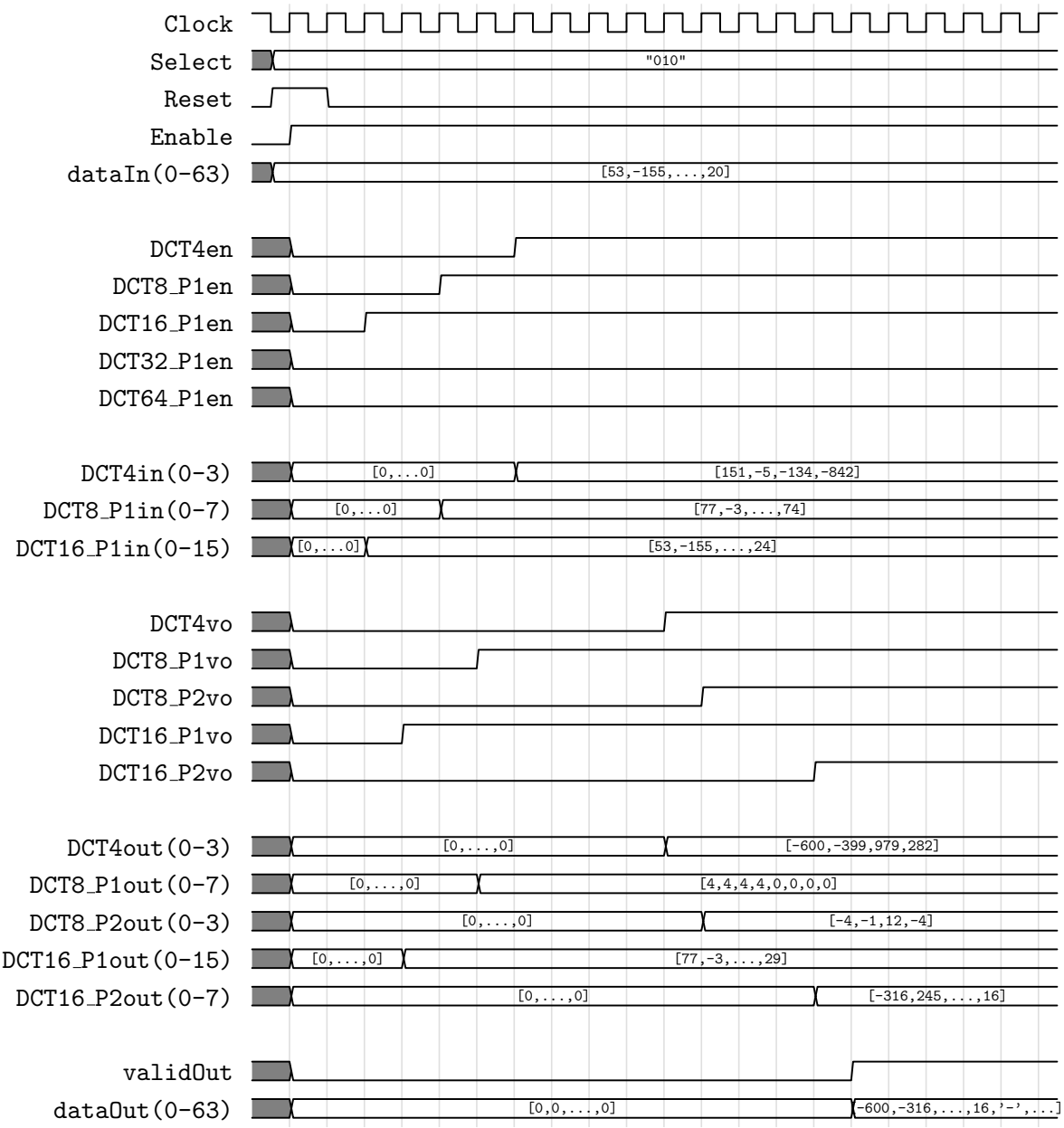


Figure 4.11: Timing diagram for a test run on the second *DCT* wrapper.

As it is observable, the central block handles the direction of intermediary coefficients and enable signals between the internal blocks, according to the selected size. The final stage taken by it is the re-organization of each of the **P2** stages' outputs.

Similarly to the previous architecture, the current was also synthesized considering an *Artix 7* FPGA, giving origin to the utilization results from Table 4.6.

Table 4.6: Second developed architecture's utilization in number of LUTs and Registers.

Block	Utilization	
	Slice LUTs	**Slice Registers**
DCT4	1077	507
DCT8_P1	709	257
DCT8_P2	1064	717
DCT16_P1	1285	513
DCT16_P2	3860	2150
DCT32_P1	3064	1025
DCT32_P2	9090	5624
DCT64_P1	6123	2049
DCT64_P2	22344	14000
Wrapper	50039	32352

As observable, this implementation occupies 66% of the area occupied by the former, approximately corresponding to the occupation taken by **DCT64**. This makes the current version more efficient from the utilization standpoint. However, it can only calculate one *DCT* size at a time, as all blocks are interdependent. This makes such implementation undesirable for an encoder, since no parallelization of transformation options is achieved and, therefore, to obtain the five *DCT* outputs for a given input vector, the architecture would need to calculate each size sequentially, bringing the total number of clock cycles to

$$N_{Total} = 6 + 10 + 14 + 18 + 22 = 70 \tag{4.14}$$

while to obtain the same results with the previous implementation there would only be needed 22.

If real time encoding is not necessary, such implementation would be most suitable for mobile applications, where size and consumed power are key requirements, as this version would perform better than the previous implementation on both of these aspects.

4.2.3 *Microblaze* Integration

Considering the system presented on Figure 2.6 (page 13, *Simplified Basic Encoder Model*), it would be safe to assume that, if it were to be implemented in hardware, the **Control Unit** would be implemented on a generic CPU, controlling the surrounding blocks, e.g. the developed *DCT* kernels. Such an architecture would allow for a complex software, such as *libaom*, to be massively simplified, as most of the complex calculations would be run on specialized co-processors, while leaving the system's controlling and decision-making processes to be run on a central unit.

In order to prove the applicability of the developed architectures in such system, the designs were tested on a *Digilent*'s Nexys 4 hardware kit, equipped with an *Artix 7* FPGA, XC7A100T-1CSG324C. This board is presented in Figure 4.12.

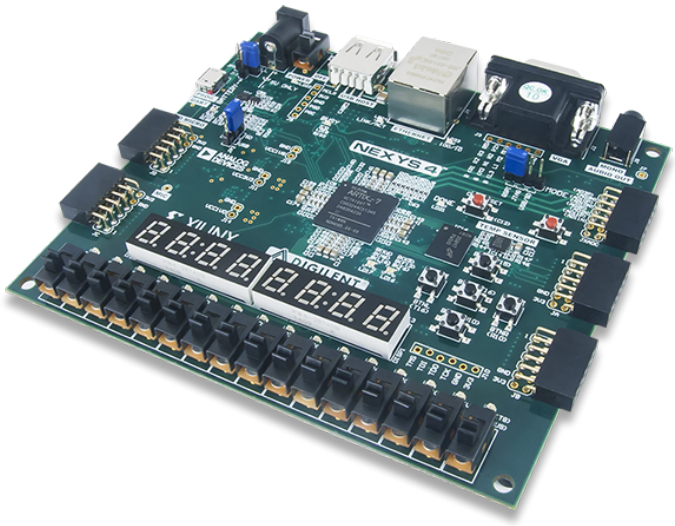


Figure 4.12: Nexys 4 hardware kit [6].

With the hardware chosen, the first step would be to synthesize the developed architectures to the desired FPGA. This operation provides numerous relevant data, such as the percentage of utilization, the estimated power drawn or some timing characteristics.

Referring to the first implementation of the *DCT* kernel, the Synthesis to the current hardware is not possible, since the design occupies 119.57% of the available slice LUTs. However, on a more capable kit, this design could be tested.

However, as the second implementation uses only 79.93%, and no other *floorplaning* errors occurred, the Synthesis and Implementation were successful, and therefore, *Vivado* provides the mentioned estimations. With an operating frequency of 100 MHz, the obtained power draw is 50 mW. With the Worst Negative Slack (WNS), it is also possible to calculate the system's maximum operating frequency. This measure represents the lowest delay to meet the design's requirements, i.e., taking the system's longest clock propagation path, if it were increased by WNS, the timing constraints would still be met. Therefore, considering T as the current operating clock's period, an estimation of the maximum operating frequency may be calculated by Equation 4.15.

$$f_{Max} = \frac{1}{T - WNS} = \frac{1}{10 \cdot 10^{-9} - 0.188 \cdot 10^{-9}} = 101.9\,MHz \tag{4.15}$$

To consider a full *libaom* integration in the focus of this work, with the target hardware, would be too demanding both in terms of complexity and achievability. Since a single block of the encoder occupies most of the available resources, the integration of the following processes would certainly need a highly capable hardware kit.

Nonetheless, a simpler architecture can still be implemented, as to test the applicability of the developed architecture on a full encoding system. To do this, the encoder's central unit would need to be instantiated on the hardware kit on the shape of a Microcontroller (MCU), alongside the developed kernels. The latter also needed to be adapted for the communications between it and the MCU.

To accomplish this, *Vivado* provides a wide set of tools to easily achieve integration of

a design described in VHDL with a generic processor, efficiently obtaining a hybrid design between the developed hardware and a software algorithm, easily developed in *C*, as an example.

Therefore, two separate tasks were conducted. First was the preparation of the *DCT Wrapper*, and finally the development of a complete system.

Taken the architecture from Figure 4.10, the first step was the creation of a custom block with an *AXI4* interface. This data-transfer protocol is heavily used in *Xilinx*'s tools and *ARM*'s processors.

It is based on a generic *Master-Slave* interaction with two separate channels. The *Address* channel includes the location of the register to read/write from, as well as the necessary control signals. The *Data* channel transports the information coming from the *Slave* (read cycle) or *Master* (write cycle).

Vivado allows for three different versions of *AXI4*:

- **AXI4** implements a highly customizable memory mapped interface, indicated for complex applications;
- **AXI4-Lite** is a simplified version of the former, keeping the memory mapped communications;
- **AXI4-Stream** implements a streaming protocol, allowing a high throughput.

For the development of this work, the second protocol was chosen, as it allowed to prove the efficiency of the developed design on a full system, while keeping a low complexity. However, if a complete encoder were to be constructed, the stream configuration would be more adequate. Since the *Wrapper* would need to process large amounts of coefficients each second, the interface from which it received the input vectors would need to be able to provide the necessary throughput.

The integration of the *Wrapper* with the interface constituted of connecting each of the `dataIn`s to one of the interface's 32 bit writable registers. This way, all the inputs are accessible by the controller when performing a writing operation. As to the outputs, the same addresses were attributed to the corresponding `dataOut`s, allowing to read the calculated coefficients. The `enable`, `reset` and `Select` signals were all connected to the same write register, as these signals occupy 5 bits in total.

Besides the *DCT* kernel, three counters were implemented. Each of these were tasked with counting the write, read and transformation clock cycles. The first two are controlled by activating/deactivating bits from the control register, while the other is controlled by the `enable` and `validOut` signals from the kernel. The output values are also accessible to the microcontroller, making possible the monitoring of the whole cycle. And as the timers are implemented in hardware, there are not as significant overheads as if the operations were timed with a software counter.

Figure 4.13 shows a simplified version of the kernel's integration with the *AXI4* Interface. **DCT Timer**'s output is concatenated with `validOut` since it does not need the full 32 bits. The internal blocks are all driven with the same system clock.

The additional structures raised the number of used Look Up Tables to 50461 (79.59% of *Artix 7*) and Registers to 34686 (27.35%).

With this block, a separate design was created. To it was added a *Microblaze* microcontroller, which instantiates a Reduced Instruction Set Computer (RISC) soft core processor, specialized for use with *Xilinx*'s products [7].

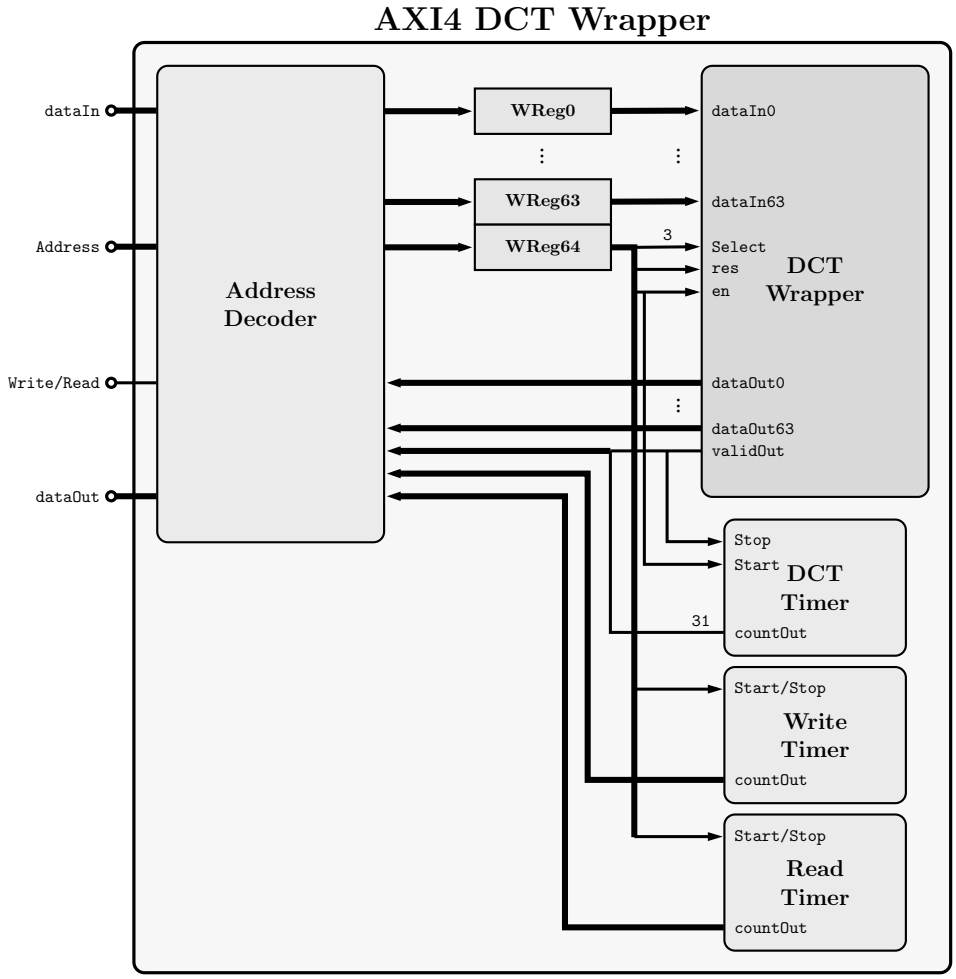


Figure 4.13: Simplified description of *DCT Wrapper* with AXI4-Lite interface.

Specifying the desired interfaces, *Vivado* automation routines handle most of the necessary connections. To the *Microblaze* and *DCT Wrapper*, was also added a Universal asynchronous receiver/transmitter (UART) connection for debugging purposes, giving origin to the block design from Figure 4.14. The addition of the microcontroller and peripherals added 1512 LUTs and 1387 registers, bringing to a final 81.98% and 28.45% utilization, respectively.

With the finalized design, some tests were developed in software, to be run on the *Microblaze*. The interaction between it and the *DCT Wrapper* is done with read and write routines provided by *Xilinx*'s Software Development Kit (SDK). These access the addresses attributed to the peripherals during *Vivado*'s Implementation process.

The first test revolved around the verification of the calculated of coefficients. The processor injected a sequence of input vectors into the co-processor, and compared the obtained results with the software's. This short test proved successful, as the hardware implementation gave the same result as the software version for all vector sizes.

Being proven the correct calculation of the transformed coefficients, the final test was to verify the timing performances of the system. For this, a small program was written, where a vector with the same length as the desired *DCT* would be loaded into the co-processor, this

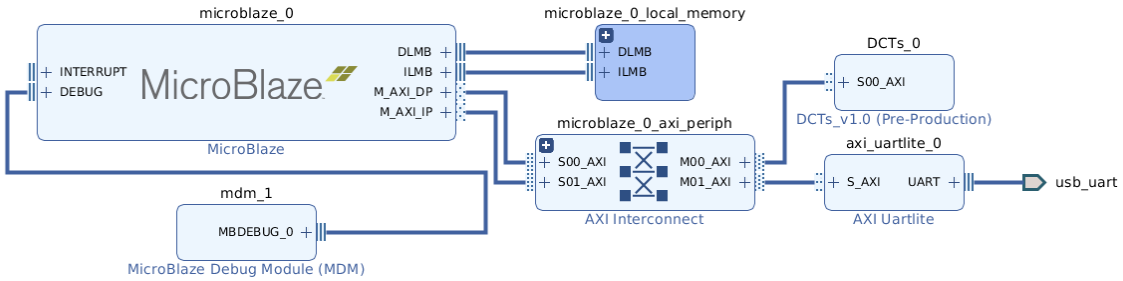


Figure 4.14: Block design generated by *Vivado* for integration of *DCT Wrapper* with *Microblaze*.

would be activated, and the same length transformed vector would be read back.

The most relevant result to measure was the number of clock cycles to calculate the transformed coefficients, and verify the accordance with the VHDL test bench. The read and write processes should only be seen as references, as they are highly dependent on the used interface. This way, in Table 4.7 there are presented the separate timing results for the three different processes measured, both in number of clock cycles, as well as the time duration at the calculated maximum frequency.

Table 4.7: Timing results for the *Microblaze* integration design.

Size	**Number of Clock Cycles($T_{@101.9MHz}$(ns))**		
	Write	**Transform**	**Read**
4	510 (5005)	6 (59)	462 (4534)
8	902 (8852)	10 (98)	806 (7910)
16	1686 (1646)	14 (137)	1494 (14661)
32	3254 (31933)	18 (177)	2870 (28165)
64	6390 (62709)	22 (216)	5622 (55172)

Taking the transformation times, it is possible to calculate the hypothetical throughput of the developed architecture on the current hardware. This is calculation is done with a similar process as the one of Table 4.4, but in reverse. Given the maximum estimated frequency, the maximum possible frame rate is estimated. In Table 4.8, these results are presented, considering all square block sizes.

As seen, the constructed architecture, on the current hardware, is not capable of processing high resolution video at usable frame rates. For HD video, it can maintain 30 fps for all block sizes. However, on higher resolutions, this throughput cannot be maintained, in order to obtain real time usability.

Other works capable of providing such performances have been published, mostly for the HEVC standard (see references [8, 9, 10]). These results should justify the choices made by the developers, since the proclaimed clock frequencies are obtained with Very Large Scale Integration (VLSI) circuits, and FPGA implementations, when present, are only used for validation purposes.

Table 4.8: Maximum frame rate for a given resolution, considering fixed square transformation blocks, on the Nexys 4 implementation.

Block Size	**Resolution**			
	1280 × 720	**1920 × 1080**	**3840 × 2160**	**7680 × 4320**
4 × 4	37	16	4	1
8 × 8	44	20	5	1
16 × 16	63	28	7	2
32 × 32	98	44	11	3
64 × 64	161	71	18	4

In addition, it should be kept in mind that the presented results represent a *best case* scenario, since they do not take into consideration any other delays from the encoding process.

Nonetheless, the integration of the developed architecture with the *Microblaze* should prove the applicability of the first on a full encoding system, even though the maximum performance of the system is not known, as it could be achieved through synthesis to ASIC.

References

[1] Wen-Hsiung Chen, C. Smith, and S. Fralick. "A Fast Computational Algorithm for the Discrete Cosine Transform". *IEEE Transactions on Communications* 25.9 (Sept. 1977), 1004–1009. ISSN: 0090-6778. DOI: 10.1109/TCOM.1977.1093941.

[2] Yun Qing Shi and Huifang Sun. *Image and Video Compression for Multimedia Engineering: Fundamentals, Algorithms, and Standards*. 2. ed. Image Processing Series. Boca Raton, Fla.: CRC Press, 2008. ISBN: 978-0-8493-7364-0.

[3] *FPGA vs ASIC, What to Choose?* https://anysilicon.com/fpga-vs-asic-choose/. Jan. 2016.

[4] *Genus Synthesis Solution*. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html.

[5] *Design Compiler Graphical*. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html.

[6] *Nexys 4 Artix-7 FPGA Trainer Board (LIMITED TIME)*. https://store.digilentinc.com/nexys-4-artix-7-fpga-trainer-board-limited-time-see-nexys4-ddr/.

[7] *MicroBlaze Processor Reference Guide*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug984-vivado-microblaze-ref.pdf. 2019.

[8] N. C. Vayalil, J. Haddrill, and Y. Kong. "An Efficient ASIC Design of Variable-Length Discrete Cosine Transform for HEVC". *2016 European Modelling Symposium (EMS)*. Nov. 2016, 229–233. DOI: 10.1109/EMS.2016.047.

[9] Pramod Kumar Meher et al. "Efficient Integer DCT Architectures for HEVC". *IEEE Transactions on Circuits and Systems for Video Technology* 24.1 (Jan. 2014), 168–178. ISSN: 1558-2205. DOI: 10.1109/TCSVT.2013.2276862.

[10] Mohamed Asan Basiri M. and Noor Mahammad Sk. "High Performance Integer DCT Architectures for HEVC". *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*. Jan. 2017, 121–126. DOI: 10.1109/VLSID.2017.68.

CHAPTER 5

Conclusions and Future Work

This work started with the objective of improving the performance of the recently released video coding standard, *AV1*, by optimization of the reference software, *libaom*, and through the development of hardware architectures for the *Transform* stage.

Thus, the reference software was analyzed on various aspects. Firstly, the internal functioning of that stage was studied, being described the most relevant features, such as the sequence of operations, internal data structures and implemented kernels.

In addition to these, statistical results were obtained, referring to the encoding choices accomplished in the encoder. The analysis of this data showed relevant opportunities to improve the performance of the reference encoder. From the number of bits used in the cosine approximations, to the verified number of occurrences of symmetric kernel blocks, there were found many starting points for the development of efficient software and hardware architectures for *AV1*'s *Transform* stage.

In this work, the tackled measure was the reduction of the number of bits used by the cosine approximations, as it was verified that these have a low impact on the obtained video quality, while influencing the overall encoding performance. With the changes performed in the reference software there was obtained a 3% reduction in the encoding time, as well as 81% lower memory usage in storing cosine approximations.

The algorithm was then described in hardware, achieving two different architectures. Although both perform the transformation as desired, they provide different levels of flexibility, as only the first provides some parallelization of encoding options. It was also shown that the proposed hardware architecture could be fitted into a real-time encoder for FHD@30fps, if a minimum operating frequency of 187 MHz were to be achieved with an ASIC implementation.

The second architecture was then implemented and tested on a *Nexys 4* board, with *Microblaze* integration. This test served as proof of concept for an eventual full encoder implementation, as the connectivity between the developed co-processor and a control unit was easily achieved and tested. However, the performance obtained with the tested kit is not adequate for implementation on a real time encoder, leading to the aforementioned necessity of ASIC implementations.

Taking these factors into consideration, it can be concluded that the original objectives were partially achieved. *Libaom*'s *Transform* stage was improved with the software changes, and two different hardware implementations were constructed for the *DCT* kernel.

Thus, the work started in this dissertation can be profoundly extended, in order to obtain an efficient hardware architecture for *AV1*.

AV1 supports two other transformation kernels, being these the *ADST* and *Identity*. In order to achieve a complete *Transform* co-processor, hardware implementations for both these kernels should be implemented.

As to the developed architectures, the most immediate measure would be the shortening of the internal signals, as well as the input and output coefficients. Although this measure would bring some complexity to the interconnection of the internal blocks, it would reduce the necessary footprint.

Other considerations would be the addition of smaller *DCT* blocks, such as **DCT4**, into the final *Wrapper*. This would go according to two aspects. Foremost, as shown in the encoding data, the most commonly used vector is the smallest, and lastly it is the quickest to be calculated, meaning that while a bigger vector is being calculated, many iterations of the smaller vectors could be run, sequentially or in parallel.

As to test the full impact of the developed hardware architectures on *libaom*, the hardware architectures should be implemented on a fitting FPGA kit, with capabilities for high-bandwidth communications between it and a CPU running the reference software, similarly to the hybrid *Microblaze* architecture.

Finally, an ASIC implementation should be considered, since, in most cases, FPGA designs tend to perform worse than specialized integrated circuits. Therefore, this design, as well as others aiming to achieve real-time, 8K video encoding performances, should be considered as ASIC implementations.

Annexes

A *aomenc* Configuration Options

```
Usage: ./aomenc <options> -o dst_filename src_filename

Options:
           --help Show usage options and exit
-c <arg>, --cfg=<arg> Config file to use
-D, --debug Debug mode (makes output deterministic)
-o <arg>, --output=<arg> Output filename
           --codec=<arg> Codec to use
-p <arg>, --passes=<arg> Number of passes (1/2)
           --pass=<arg> Pass to execute (1/2)
           --fpf=<arg> First pass statistics file name
           --limit=<arg> Stop encoding after n input frames
           --skip=<arg> Skip the first n input frames
           --good Use Good Quality Deadline
-q, --quiet Do not print encode progress
-v, --verbose Show encoder parameters
           --psnr Show PSNR in status line
           --webm Output WebM (default when WebM IO is enabled)
           --ivf Output IVF
           --obu Output OBU
-P, --output-partitions Makes encoder output partitions. Requires IVF output!
           --q-hist=<arg> Show quantizer histogram (n-buckets)
           --rate-hist=<arg> Show rate histogram (n-buckets)
           --disable-warnings Disable warnings about potentially incorrect encode
               settings.
-y, --disable-warning-prompt Display warnings, but do not prompt user to continue.
           --test-decode=<arg> Test encode/decode mismatch
                               off, fatal, warn

Encoder Global Options:
           --yv12 Input file is YV12
           --i420 Input file is I420 (default)
           --i422 Input file is I422
           --i444 Input file is I444
-u <arg>, --usage=<arg> Usage profile number to use
-t <arg>, --threads=<arg> Max number of threads to use
           --profile=<arg> Bitstream profile number to use
-w <arg>, --width=<arg> Frame width
-h <arg>, --height=<arg> Frame height
           --forced_max_frame_width Maximum frame width value to force
           --forced_max_frame_height Maximum frame height value to force
           --stereo-mode=<arg> Stereo 3D video format
                               mono, left-right, bottom-top, top-bottom, right-
                                   left
           --timebase=<arg> Output timestamp precision (fractional seconds)
           --fps=<arg> Stream frame rate (rate/scale)
           --global-error-resilient=< Enable global error resiliency features
-b <arg>, --bit-depth=<arg> Bit depth for codec (8 for version <=1, 10 or 12 for
```

```
    version 2)
                                8, 10, 12
          --lag-in-frames=<arg> Max number of frames to lag
          --large-scale-tile=<arg> Large scale tile coding (0: off (default), 1: on
              )
          --monochrome Monochrome video (no chroma planes)
          --full-still-picture-hdr Use full header for still picture

Rate Control Options:
          --drop-frame=<arg> Temporal resampling threshold (buf %)
          --resize-mode=<arg> Frame resize mode
          --resize-denominator=<arg> Frame resize denominator
          --resize-kf-denominator=<a Frame resize keyframe denominator
          --superres-mode=<arg> Frame super-resolution mode
          --superres-denominator=<ar Frame super-resolution denominator
          --superres-kf-denominator= Frame super-resolution keyframe denominator
          --superres-qthresh=<arg> Frame super-resolution qindex threshold
          --superres-kf-qthresh=<arg Frame super-resolution keyframe qindex
              threshold
          --end-usage=<arg> Rate control mode
                              vbr, cbr, cq, q
          --target-bitrate=<arg> Bitrate (kbps)
          --min-q=<arg> Minimum (best) quantizer
          --max-q=<arg> Maximum (worst) quantizer
          --undershoot-pct=<arg> Datarate undershoot (min) target (%)
          --overshoot-pct=<arg> Datarate overshoot (max) target (%)
          --buf-sz=<arg> Client buffer size (ms)
          --buf-initial-sz=<arg> Client initial buffer size (ms)
          --buf-optimal-sz=<arg> Client optimal buffer size (ms)

Twopass Rate Control Options:
          --bias-pct=<arg> CBR/VBR bias (0=CBR, 100=VBR)
          --minsection-pct=<arg> GOP min bitrate (% of target)
          --maxsection-pct=<arg> GOP max bitrate (% of target)

Keyframe Placement Options:
          --enable-fwd-kf=<arg> Enable forward reference keyframes
          --kf-min-dist=<arg> Minimum keyframe interval (frames)
          --kf-max-dist=<arg> Maximum keyframe interval (frames)
          --disable-kf Disable keyframe placement

AV1 Specific Options:
          --cpu-used=<arg> CPU Used (0..8)
          --dev-sf=<arg> Dev Speed (0..255)
          --auto-alt-ref=<arg> Enable automatic alt reference frames
          --sharpness=<arg> Loop filter sharpness (0..7)
          --static-thresh=<arg> Motion detection threshold
          --single-tile-decoding=<ar Single tile decoding (0: off (default), 1: on)
          --tile-columns=<arg> Number of tile columns to use, log2
          --tile-rows=<arg> Number of tile rows to use, log2 (set to 0 while
              threads > 1)
          --arnr-maxframes=<arg> AltRef max frames (0..15)
          --arnr-strength=<arg> AltRef filter strength (0..6)
```

```
--tune=<arg> Distortion metric tuned with
                          psnr, ssim, cdef-dist, daala-dist
--cq-level=<arg> Constant/Constrained Quality level
--max-intra-rate=<arg> Max I-frame bitrate (pct)
--max-inter-rate=<arg> Max P-frame bitrate (pct)
--gf-cbr-boost=<arg> Boost for Golden Frame in CBR mode (pct)
--lossless=<arg> Lossless mode (0: false (default), 1: true)
--enable-cdef=<arg> Enable the constrained directional enhancement filter
     (0: false, 1: true (default))
--enable-restoration=<arg> Enable the loop restoration filter (0: false,
    1: true (default))
--disable-trellis-quant=<a Disable trellis optimization of quantized
    coefficients (0: false (default) 1: true)
--enable-qm=<arg> Enable quantisation matrices (0: false (default), 1:
    true)
--qm-min=<arg> Min quant matrix flatness (0..15), default is 8
--qm-max=<arg> Max quant matrix flatness (0..15), default is 15
--enable-dist-8x8=<arg> Enable dist-8x8 (0: false (default), 1: true)
--frame-parallel=<arg> Enable frame parallel decodability features (0:
    false (default), 1: true)
--error-resilient=<arg> Enable error resilient features (0: false (
    default), 1: true)
--aq-mode=<arg> Adaptive quantization mode (0: off (default), 1: variance
     2: complexity, 3: cyclic refresh)
--deltaq-mode=<arg> Delta qindex mode (0: off (default), 1: deltaq 2:
    deltaq + deltalf)
--frame-boost=<arg> Enable frame periodic boost (0: off (default), 1: on)
--noise-sensitivity=<arg> Noise sensitivity (frames to blur)
--tune-content=<arg> Tune content type
                          default, screen
--cdf-update-mode=<arg> CDF update mode for entropy coding (0: no CDF
    update; 1: update CDF on all frames(default); 2: selectively update
    CDF on some frames
--color-primaries=<arg> Color primaries (CICP) of input content:
                          bt709, unspecified, bt601, bt470m, bt470bg,
                              smpte240, film, bt2020, xyz, smpte431,
                              smpte432, ebu3213
--transfer-characteristics Transfer characteristics (CICP) of input
    content:
                          unspecified, bt709, bt470m, bt470bg, bt601,
                              smpte240, lin, log100, log100sq10, iec61966,
                               bt1361, srgb, bt2020-10bit, bt2020-12bit,
                              smpte2084, hlg, smpte428
--matrix-coefficients=<arg Matrix coefficients (CICP) of input content:
                          identity, bt709, unspecified, fcc73, bt470bg,
                              bt601, smpte240, ycgco, bt2020ncl, bt2020cl,
                               smpte2085, chromncl, chromcl, ictcp
--chroma-sample-position=< The chroma sample position when chroma 4:2:0
    is signaled:
                          unknown, vertical, colocated
--min-gf-interval=<arg> min gf/arf frame interval (default 0, indicating
    in-built behavior)
--max-gf-interval=<arg> max gf/arf frame interval (default 0, indicating
```

```
            in-built behavior)
          --sb-size=<arg> Superblock size to use
                                  dynamic, 64, 128
          --num-tile-groups=<arg> Maximum number of tile groups, default is 1
          --mtu-size=<arg> MTU size for a tile group, default is 0 (no MTU
            targeting), overrides maximum number of tile groups
          --timing-info=<arg> Signal timing info in the bitstream (model unly works
             for no hidden frames, no super-res yet):
                                  unspecified, constant, model
          --film-grain-test=<arg> Film grain test vectors (0: none (default), 1:
            test-1 2: test-2, ... 16: test-16)
          --film-grain-table=<arg> Path to file containing film grain parameters
          --enable-ref-frame-mvs=<ar Enable temporal mv prediction (default is 1)
-b <arg>, --bit-depth=<arg> Bit depth for codec (8 for version <=1, 10 or 12 for
    version 2)
                                  8, 10, 12
          --input-bit-depth=<arg> Bit depth of input
          --sframe-dist=<arg> S-Frame interval (frames)
          --sframe-mode=<arg> S-Frame insertion mode (1..2)
          --annexb=<arg> Save as Annex-B

Stream timebase (--timebase):
The desired precision of timestamps in the output, expressed
in fractional seconds. Default is 1/1000.

Included encoders:

av1 - AOMedia Project AV1 Encoder v0.1.0 (default)

    Use --codec to switch to a non-default encoder.
```

B DCT8_1 VHDL Description

```
-- DCT8 Implementation for inetgration with aggregated architecture

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

entity DCT8_1_I is
port( -- Data Inputs
    dataIn0 : in integer;
    dataIn1 : in integer;
    dataIn2 : in integer;
    dataIn3 : in integer;
    dataIn4 : in integer;
    dataIn5 : in integer;
    dataIn6 : in integer;
```

```
    dataIn7 : in integer;
    -- Control Inputs
    res : in std_logic;
    en : in std_logic;
    clk : in std_logic;
    -- Data Outputs
    dataOut0 : out integer;
    dataOut1 : out integer;
    dataOut2 : out integer;
    dataOut3 : out integer;
    dataOut4 : out integer;
    dataOut5 : out integer;
    dataOut6 : out integer;
    dataOut7 : out integer;
    -- Control Outputs
    validOut : out std_logic
);
end DCT8_1_I;

architecture Behavioral of DCT8_1_I is
begin

stage1: process(clk, res, en)
        begin
            if(rising_edge(clk)) then
                if(res = '1') then
                    dataOut0 <= 0;
                    dataOut1 <= 0;
                    dataOut2 <= 0;
                    dataOut3 <= 0;
                    dataOut4 <= 0;
                    dataOut5 <= 0;
                    dataOut6 <= 0;
                    dataOut7 <= 0;
                    validOut <= '0';
                elsif(en = '1') then
                    dataOut0 <= dataIn0 + dataIn7;
                    dataOut1 <= dataIn1 + dataIn6;
                    dataOut2 <= dataIn2 + dataIn5;
                    dataOut3 <= dataIn3 + dataIn4;
                    dataOut4 <= dataIn3 - dataIn4;
                    dataOut5 <= dataIn2 - dataIn5;
                    dataOut6 <= dataIn1 - dataIn6;
                    dataOut7 <= dataIn0 - dataIn7;
                    validOut <= '1';
                end if;
            end if;
```

```
        end process;
end Behavioral;
```

C DCT8_2 VHDL Description

```
-- DCT8 Stage 2 Implementation for integration with aggregated architecture

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

entity DCT8_2_I is
port( -- Data Inputs
    dataIn4 : in integer;
    dataIn5 : in integer;
    dataIn6 : in integer;
    dataIn7 : in integer;
    -- Control Inputs
    res : in std_logic;
    en : in std_logic;
    clk : in std_logic;
    -- Data Outputs
    dataOut4 : out integer;
    dataOut5 : out integer;
    dataOut6 : out integer;
    dataOut7 : out integer;
    -- Control Outputs
    validOut : out std_logic
);
end DCT8_2_I;

architecture Behavioral of DCT8_2_I is
signal s_stg2M5, s_stg2M6 : integer := 0;
signal s_stg2A5, s_stg2A6 : integer := 0;
signal s_stg2D5, s_stg2D6 : integer := 0;
signal s_stg34, s_stg35, s_stg36, s_stg37 : integer := 0;
signal s_stg4M41, s_stg4M42, s_stg4M51, s_stg4M52, s_stg4M61, s_stg4M62,
    s_stg4M71, s_stg4M72 : integer := 0;
signal s_stg4A4, s_stg4A5, s_stg4A6, s_stg4A7 : integer := 0;
signal s_stage2MEn, s_stage2AEn, s_stage2DEn, s_stage3En, s_stage4MEn,
    s_stage4AEn, s_valOut : std_logic := '0';
begin

stage2M: process(clk, res, en)
        begin
```

```
            if(rising_edge(clk)) then
                if(res = '1') then
                    s_stg2M5 <= 0;
                    s_stg2M6 <= 0;
                    s_stage2AEn <= '0';
                elsif(en = '1') then
                    s_stg2M5 <= dataIn5*185;
                    s_stg2M6 <= dataIn6*185;
                    s_stage2AEn <= '1';
                end if;
            end if;
        end process;

stage2A: process(clk, res, s_stage2AEn)
        begin
            if(rising_edge(clk)) then
                if(res = '1') then
                    s_stg2A5 <= 0;
                    s_stg2A6 <= 0;
                    s_stage2DEn <= '0';
                elsif(s_stage2AEn = '1') then
                    s_stg2A5 <= s_stg2M6 - s_stg2M5;
                    s_stg2A6 <= s_stg2M6 + s_stg2M5;
                    s_stage2DEn <= '1';
                end if;
            end if;
        end process;

stage2D: process(clk, res, s_stage2DEn)
        begin
            if(rising_edge(clk)) then
                if(res = '1') then
                    s_stg2D5 <= 0;
                    s_stg2D6 <= 0;
                    s_stage3En <= '0';
                elsif(s_stage2DEn = '1') then
                    s_stg2D5 <= to_integer(shift_right(to_signed(s_stg2A5,32)
                        ,8));
                    s_stg2D6 <= to_integer(shift_right(to_signed(s_stg2A6,32)
                        ,8));
                    s_stage3En <= '1';
                end if;
            end if;
        end process;

stage3: process(clk, res, s_stage3En)
        begin
```

```
            if(rising_edge(clk)) then
                if(res = '1') then
                    s_stg34 <= 0;
                    s_stg35 <= 0;
                    s_stg36 <= 0;
                    s_stg37 <= 0;
                    s_stage4MEn <= '0';
                elsif(s_stage3En = '1') then
                    s_stg34 <= dataIn4 + s_stg2D5;
                    s_stg35 <= dataIn4 - s_stg2D5;
                    s_stg36 <= dataIn7 - s_stg2D6;
                    s_stg37 <= dataIn7 + s_stg2D6;
                    s_stage4MEn <= '1';
                end if;
            end if;
        end process;

stage4M: process(clk, res, s_stage4MEn)
        begin
            if(rising_edge(clk)) then
                if(res = '1') then
                    s_stg4M41 <= 0;
                    s_stg4M42 <= 0;
                    s_stg4M51 <= 0;
                    s_stg4M52 <= 0;
                    s_stg4M61 <= 0;
                    s_stg4M62 <= 0;
                    s_stg4M71 <= 0;
                    s_stg4M72 <= 0;
                    s_stage4AEn <= '0';
                elsif(s_stage4MEn = '1') then
                    s_stg4M41 <= s_stg34*56;
                    s_stg4M42 <= s_stg34*252;
                    s_stg4M51 <= s_stg35*147;
                    s_stg4M52 <= s_stg35*216;
                    s_stg4M61 <= s_stg36*147;
                    s_stg4M62 <= s_stg36*216;
                    s_stg4M71 <= s_stg37*56;
                    s_stg4M72 <= s_stg37*252;
                    s_stage4AEn <= '1';
                end if;
            end if;
        end process;

stage4A: process(clk, res, s_stage4AEn)
        begin
            if(rising_edge(clk)) then
```

```
                if(res = '1') then
                    s_stg4A4 <= 0;
                    s_stg4A5 <= 0;
                    s_stg4A6 <= 0;
                    s_stg4A7 <= 0;
                    s_valOut <= '0';
                elsif(s_stage4AEn = '1') then
                    s_stg4A4 <= s_stg4M41 + s_stg4M72;
                    s_stg4A5 <= s_stg4M52 + s_stg4M61;
                    s_stg4A6 <= s_stg4M62 - s_stg4M51;
                    s_stg4A7 <= s_stg4M71 - s_stg4M42;
                    s_valOut <= '1';
                end if;
            end if;
        end process;

outReg: process(clk, res, s_valOut)
        begin
            if(rising_edge(clk)) then
                if(res = '1') then
                    dataOut5 <= 0;
                    dataOut6 <= 0;
                    dataOut7 <= 0;
                    validOut <= '0';
                elsif(s_valOut = '1') then
                    dataOut4 <= to_integer(shift_right(to_signed(s_stg4A4,32)
                        ,8));
                    dataOut5 <= to_integer(shift_right(to_signed(s_stg4A5,32)
                        ,8));
                    dataOut6 <= to_integer(shift_right(to_signed(s_stg4A6,32)
                        ,8));
                    dataOut7 <= to_integer(shift_right(to_signed(s_stg4A7,32)
                        ,8));
                    validOut <= '1';
                end if;
            end if;
        end process;
end Behavioral;
```