Hardware Design Implementation of HEVC IDCT Algorithm with High-Level Synthesis

Author

Magoulianitis Vasileios

A thesis submitted for the Faculty of the Department of Electrical and Computer Enginnering in Partial Fulfillment of the Requirements for the Diploma of Science

Supervisors:

Dr. Christos Sotiriou

Dr. Gerasimos Potamianos

Department of Electrical and Computer Engineering

UNIVERSITY OF THESSALY

October 13, 2015

UNIVERSITY OF THESSALY

Department of Electrical and Computer Engineering

Hardware Design Implementation of HEVC IDCT Algorithm with High-Level Synthesis

by

Magoulianitis Vasileios

Graduate Thesis for the degree of Diploma of Science in Computer and Communication Engineering

Approved by the two member inquiry committee at 13th of October 2015

Dr. CHRISTOS SOTIRIOU Dr. GERASIMOS POTAMIANOS

Declaration of Authorship

I, Vasileios Magoulianitis, declare that this thesis titled, 'Hardware Implementation of HEVC Inverse Integer Transform with High-Level Synthesis Tool' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a bachelors degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"The roots of education are bitter, but the fruit is sweet."

Aristotle

Abstract

Video applications are used widely nowadays, supporting different aspects of our life, such as entertainment, security, medicine e.t.c.. However, the increasing number of video contents yields issues in its storage and transmission, since it conveys high volume of data. High Efficiency Video Coding (HEVC) is the new video compression standard, reducing bitrates nearly at half compared to its predecessor the H.264, supporting high demanding video contents. This reduction in bitrate is achieved by a series of computationally expensive algorithms, thus making imperative to implement some complex parts of HEVC into hardware, so as to meet the real-time constraint of video coding applications. Unfortunately, hardware design maintains large cycle for design and verification processes and therefore many man-months have to be spend for a hardware video codec implementation.

High-Level Synthesis (HLS) draws much attention from industry lately, because of the short time-to-market value that it offers. Describing an algorithm with C/C++, is much easier than writing Hardware Description Languages (HDLs), while lets to explore different architectures for the same algorithm. Also, HLS is used widely for Digital Signal Processing (DSP) applications, one of them being video coding. Therefore, the subject of this thesis is the design space exploration of the HEVC Inverse Integer Transform (IIT) using the Vivado HLS tool, for synthesis on FGPAs. Different code versions and directives, yields different RTLs in terms of latency and device utilization. All these RTLs, are extensively analyzed according to their throughput performance, identifying the different video contents that each of them can support. Finally, a throughput comparison is conducted among other levels of implementation, in order to find how efficient are the RTLs from a HLS tool. Results, show that HLS tools provide great flexibility for design space exploration and verification of RTL, but their efficiency ($performance/area$) is still far, when compared with RTLs written by human's hand.

Περιληψη

Οι διαφορες χρησεις του βιντεο στις ημερες μας, πληθαινουν ολοενα και περισσοτερο κυριως σε τομεις οπως η ψυχαγωγια, η ασφαλεια, η ιατρικη κ.τ.λ. Ο αυξανομενος αριθμος των εφαρμογων που χρησιμοποιουν βιντεο, δημιουργει προβληματα τοσο στην αποθηκευση του, οσο και στην μεταδοση του διαμεσω καναλιων επικοινωνιας, καθιστωντας την συμπιεση των δεδομενων βιντεο ιδιατερα σημαντικη. Το τελευταο βιντεο στανταρντ H.265 προσφερει σημαντικη συμπιεση στα δεδομενα, σχεδον διπλασια απο το προηγουμενο H.264 στανταρντ. Ωστοσο, το διπλασιο κερδος σε συμπιεση, επιτυγχανεται χρησιμοποιωντας μια σειρα πολυπλοκων αλγοριθμων, αυξανοντας την συνολικη πολυπλοκοτητα στον αποκωδικοποιητη περιπου στο διπλασιο, ο οποιος σημειωτεον, πρεπει οπωσδηποτε να τρεξει σε πραγματικο χρονο και να επεξεργαζεται δεδομενα με συγκεκριμενο ρυθμο. Ως εκ τουτου, γινεται σαφες οτι καποιοι περιπλοκοι αλγοριθμοι, θα πρεπει να υλοποιηθουν σε υλικο ετσι ωστε να επιταγχυνουμε την αποκωδικοποιηση. Οσων αφορα τωρα την αναπτυξη υλικου, τελευταια απο τη βιομηχανια εχουν αρχισει να εξερευνωνται εργαλεια συνθεσης απο υψηλο επιπεδο, λογω του μικρου κυκλου εργασιας που απαιτουν για την σχεδιαση και επαληθευση κυκλωματων. Ειδικοτερα, τετοια εργαλεια χρησιμοποιουνται ευρεως για αλγοριθμους ψηφιακης επεξεργασιας σηματων –οπως ειναι η συμπιεση του βιντεο– επιτρεποντας διαφορετικες αρχιτεκτονικες λυσεις για εναν αλγοριθμο σε μικρο χρονικο διαστημα. Ετσι, το αντικειμενο αυτης της εργασιας ειναι η εξερευνηση του χωρου λυσεων διαφορων κυκλωματων για τον αντιστροφο μετασχηματισμο του αποκωδικοποιητη H.265 με χρηση καταλληλων εργαλειων για συνθεση απο υψηλο επιπεδο. Οι διαφορετικοι κωδικες και οι διαφορετικες οδηγιες που δοθηκαν στο εργαλειο, μας εδωσαν διαφορετικα κυκλωματα, με βαση τον χωρο που καταλαμβανουν σε μια επαναπρογραμματιζομενη συσκευη με βαση τον ρυθμο των δεδομενων που επεξεργαζονται καθε δευτερολεπτο. Ολα ται διαφορετικα κυκλωματα που μας εδωσε το εργαλειο, διευρενωνται ως προς την αποδοση τους, ωστε να διαπιστωσουμε ποσο αποδοτικες ειναι οι λυσεις που βγαζουν αυτα τα εργαλεια, σε συγκριση με αλλα επιπεδα υλοποιησης. Τελικα, στοχος μας ειναι να δουμε ποια ειναι τα ορια αυτων των εργαλειων και ποιο ειναι το μεγαλυτερο αναλυσης βιντεο που μπορει να υποστηριχθει απο τετοια κυκλωματα. Τα αποτελεσματα δειχνουν, οτι τα εργαλεια συνθεσης απο υψηλο επιπεδο, προσφερουν μεγαλη ευελιξια, τοσο στην συνθεση οσο και στην επαληθευση των κυκλωματων, αλλα ειναι μακρια σε αποδοση, σε σχεση με κυκλωματα, στα οποια η αρχιτεκτονικη περιγραφεται απευθειας απο τον ανθρωπο.

Acknowledgements

At this point, I feel glad to express my sincere gratitude to my supervisor, Professor Christos Sotiriou, who trusted and supported me throughout this work. His technical background and in depth knowledge of the subject, provided valuable feedback to this thesis. Also, i would like to thank him for his precious advices from our office discussions.

Furthermore, i would like to thank my co-advisor, Professor Gerasimos Potamianos, for his feedback at the final stages of this thesis.

Once the accomplishment of this work required mixed knowledge from different divisions of computer engineering, i would like to express my acknowledgments to all the professors that i collaborated through my undergraduate studies.

I am also grateful to my closest friends and colleagues for helping and supporting me all those years.

Last but not least, I would like to thank the members of my family, for everything they offered me in all aspects of my life and their support during my academic experience.

Contents

List of Figures

List of Tables

Abbreviations

HEVC	**H**igh **E**ffeciency **V**ideo **C**oding
AVC	**A**dvanced **V**ideo **C**oding
IIT	**I**nverse **I**nteger **T**ransform
CTU	**C**oding **T**ree **U**nit
CB	**C**oding **B**lock
PU	**P**rediction **U**nit
TU	**T**ransform **U**nit
QP	**Q**uantization **P**arameter
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
RTL	**R**egister **T**ransfer **L**ayer
HLS	**H**igh **L**evel **S**ynthesis
MPEG	**M**otion **P**icture **E**xperts **G**roup
JPEG	**J**oint **P**hotographic **E**xperts **G**roup
SIMD	**S**ingle **I**nstruction **M**ultiply **D**ata
HD	**H**igh **D**efinition

To my family...

Chapter 1

Introduction

In 21st century, digital video applications are used widely in a huge variety of daily consumer products. Desktop PCs, laptops, cell phones, tablets, TVs and watches, are only a some small part of the high volume of applications that use video technology. However, video is not only used for entertainment purposes, but its objective span in many other fields. Video surveillance, video tracking and medicine are some of the aspects that video enhances our life, out of entertainment reasons. In previous decade, some of the previous applications were using analog video signal to perform their various processing tasks. In this thesis work, we are occupying only with digital video, since analog seems to be the predecessor of it, because most –if not all– of today video applications, are using digital technology.

Unfortunately, digital video has a huge volume of data that must be stored, processed and submitted, thus making imperative the compression of those big data. To realize the magnitude of video data, we quote that according to CISCO, "2/3 of the internet traffic, will belong in video by 2018" [3]. Video standards that are used in order to compress video data, remain an open field the last three decades, for both research and industry development and therefore they attract quite deliberation from research community. Progress in video compression technology, results in higher compression ratio, while retaining same perceptive quality –in terms of dB– from human eye. Every video compression standard is characterized as lossy compressor, because it cannot retrieves all primary information, although, this loss is not perceivable many times from human eye. Hence, each video standard incorporates itself many years of research, to attain better compression ratio while keeping video reconstruction quality in as much as possible standard levels. Although each new coming video standard has better performance in video coding, the complexity that is introduced in each of them constantly increases,

because more and more complex algorithms are used to achieve improved results in compression. High Efficiency Video Coding (HEVC) or H.265-ITU is the new video coding standard, was introduced in 2013 that reduces bitrates at half in video streaming, in comparison with its predecessor the H.264/AVC (Advanced Video Coding) standard. The module which we are implementing in this work, is a part of HEVC decoder.

Video coding, except for the large data that has to manage, has also another characteristic that video implementations have to take into account before their design. Video coding/decoding can be considered as critical tasks, because of the high complexity of their algorithms and also the requirement that they need to be performed in a real-time constraint. Let us assume for instance that a video decoding application has to decode 30 fps –which is a typical frame rate– in a dedicated resolution, but it has performance only for 20 fps. One can be easily deduced from this assumption, video playback shall stalls, something that it is an undesirable effect. Consequently, video coding and decoding applications have to be implemented under certain specifications, regarding video resolution and frame rate, so as to achieve a minimum performance requirement.

Several video coding applications that have been developed so far, are implemented entirely either in software or in hardware. Software solutions, running on several types of processors, provide very flexible solutions for video coding in terms of design cost and portability, but they have poor performance for high content videos. Optimization can be performed in software solutions, exploiting hardware resources that are called "hardware accelerators", accelerating critical algorithms from video codecs. On the other side, exclusively hardware implementations, are very efficient in performing video coding tasks, achieving high throughput performance, while running in relatively low operating frequencies. Having although high design life cycle (design, simulation, debugging, verification, fabrication and testing), the disadvantages comes from design cost considerations, as concerning time-to-market value.

High Level Synthesis (HLS) concept, last yeas have been introduced more aggressively in industry, in order to overcome the high cost disadvantage of custom-hardware implementations. Describing an algorithm with a software level language, such as C/C++ is much easier than writing Hardware Description Languages (HDLs) and leads to quicker exploration of design space. Moreover, the simulation of C/C++ code is faster, because writing a C/C++ testbench to verify module's functionality, is much quicker than writing it into Verilog or VHDL. As a consequence and as already implied, HLS takes as input an algorithm in C/C++ and exports RTL (Verilog or VHDL). The RTL that is exported can be easily verified, only if we assure that C algorithm has proper functionality. Hence, an HLS tool guarantees that if C/C++ code works properly, then the exported RTL shall have the same behavioral functionality. Therefore, one needs to verify our

design in RTL level, only write a piece of code in software that will verify the algorithm in software level. After that, a reference output has to be used for comparison and eventually if output results match, we do know that RTL will have the same behavioral functionality. Now, as concerning the performance of HLS implementations, it does not reach those of custom hardware RTLs, because RTL is outputted from a tool which follows standard templates and techniques. Nevertheless, the shorter time-to-market value that they provide, has special worth in industry, thus drawing inevitably as much attention from research community. Further details on HLS concept and Vivado HLS tool, are explained in Chapter 3 and in Chapter 6 we will ascertain where HLS performance stands among others.

Video codecs have several different modules that perform compression algorithms. In almost every video and image codec, there is a module that converts a block of pixels from its spatial representation to frequency domain, in order to evaluate a block of pixels according to its frequency components, so to reject those components that are not perceivable to human eye. The algorithm that is shouldered in performing this task, is called Integer Transform and is essentially the same with Discrete Cosine Transform (DCT) algorithm. One difference that exists, integer only numbers are now used replacing floating point arithmetic of DCT. DCT is based on Fourier's family transforms and its usage is not limited to video coding. Other applications such as video processing, computer vision, audio coding, speech recognition and communication, also use some kind of DCT algorithm. This thesis, conducts a HLS implementation, regarding the HEVC Inverse Integer Transform in particular, which is used in video decoding applications and converts frequency coefficients back to spatial domain. Further details are extensively discussed on this interesting algorithm in Chapter 3.

1.1 Motivation

The number of video applications increases day by day for a variety of reasons, in many different aspects of our life. Also, the big amount of video data that are transmitted worldwide has to be reduced, for bandwidth saving reasons and so as to store video using less storage space. This problem of huge data gets worse, as the video content increases. Today video applications have the trend to use more and more higher resolutions and frame rates, in order to provide better visual quality to users, requiring as much less inherited distortion from compression process. HEVC, achieves best results among prior standards, regarding compression ratio for a standard video quality and definitely will be used in future video applications.

Adopting future applications HEVC standard, have to deal with a variety of issues that will be presented. The high complexity that have been introduced in this more sophisticated codec, is the major concern about HEVC adoption. According to a survey [4], HEVC decoder is roughly twice more complex than AVC decoder and HEVC encoder is expected to be several times more complex than H.264/AVC encoder. For this reason, future researches should propose optimized implementations in different platforms, thus supporting different target video contents according to the specifications of the target device. Software implementations, have low granularity levels for optimization, in comparison with the hardware ones. Some complex modules of HEVC have to be implemented using hardware accelerators, in order to enhance software implementations and to achieve demanding performance, for supporting high video contents.

Another incentive of this work, HLS, as already mentioned, has attracted a lot of attention in recent years from industry, because it provides shorter design cycle and eventually smaller time-to-market when compared to traditional hardware implementations, though it doesn't achieves the performance of custom RTLs. In other words, it provides great flexibility to explore hardware design space of a specific algorithm, in comparison with custom RTLs. Hardware accelerators that can be created with HLS tools, may be used from embedded systems, in order to enhance some critical parts of HEVC decoder and encoder. If a HLS implementation meets a certain performance requirement and the specifications of the circuit (area, power, delay) are also met, HLS could be a quick and efficient solution, for creating a hardware accelerator. Afterwards, this accelerator can be placed onto FPGAs or in embedded systems, or to create an ASIC hardware accelerator, which is going to enhance parts from software codecs. Hence, future video implementations may use HLS, so to explore more efficiently and rapidly the design space of the HEVC video codec implementation, which requires as much speedup as only hardware can provides.

Finally, the HEVC Inverse Integer Transform module that we got hands-on, is among the most complex modules of a video decoder and in HEVC decoder its complexity has further increased at 9% according to [4], due to the higher number of transform sizes. So, the acceleration of inverse transform is valuable, in order to accelerate the HEVC video decoder.

1.2 Objective

- ◇ Design space exploration of the HEVC Inverse Integer Transform (IIT) algorithm using Vivado HLS tool, so to realize the pros and cons from different RTLs that

tool derives and how HLS tool reacts on different directives and sources, describing the same algorithm.

◇ Deciphering the Vivado HLS tool, on how it manipulates latency on different architectures and data paths and how RTL architecture changes, forcing design to meet as much lower delay constraints.

◇ Throughput exploration analysis, so as to identify the different architectural plans for the algorithm, what throughput performance will they have. The outer purpose of this exploration, how a HLS tool is compared with other implementations in terms of throughput performance, such as software (x86), SIMD-accelerated software and custom-hardware RTL implementations.

◇ Realize when each different RTL solution becomes a critical component in a video decoder at the IIT module, thus finding the limits of HLS for decoding demanding video contents.

1.3 Other Works

In this section we briefly discuss, what other works exist on video codec implementations, just to have an intuition about the different platforms and levels of implementation and also the results that other works provide.

Video coding is an open topic in research community and for this reason several papers have been published all those twenty five years that digital video had a great evolution. Research works can be distinguished into two major classes. The first category deals with proposals that induces on video coding field in terms of signal processing, thus determining algorithms and methods for improving compression. The second category deals with ways, to implement different video codecs, in different platforms and making different trade offs and optimizations. This thesis, is entirely related to the latter category, so we are going to focus on this, in literature review.

Several implementations have been proposed so far for every new standard. starting from pure software. up to custom hardware RTL. Software solutions mainly focus, either on supporting as much higher frame rates and resolutions, exploiting SIMD architectures on processors [5] and [6] or on performing complexity analysis [4], in order to give useful information in other researches that will use them.

Other implementations are based on software, but gain a lot of performance from hardware features. Configurable microprocessors are such a solution because the Instruction Set Architecture (ISA) of those low-power microprocessors, can be extended with new custom instructions that will reduce total cycle effort and eventually shall increase performance or reduce power consumption. Various works have been proposed on this level for different video codecs such as [18] for H.264/AVC and [19] for HEVC.

The lower level of implementation is hardware RTL that is going to be used, either as a hardware accelerator onto an embedded system or as a module in a hardware video codec on FPGA or ASIC. Hardware implementations, due to the high design's complexity often are focusing in a specific module of a video codec and they provide different optimization results about performance, area and power. In general, hardware RTLs before turn in logic synthesis flow, can be distinguished as "custom-made", where architecture is designed from engineers or it can be exported from a HLS tool. Some good hardware references regarding some complex modules of HEVC are: [12] and [13] that they implement motion compensation module, [8], [9], [10] and [11] for integer transform module and [14], [15] and [16] that they are touching the difficult CABAC–entropy coding module of HEVC.

Hardware implementation proposals are countless, because video codecs are so complex applications and therefore they have a strong requirement in hardware, that their examination cannot be limited in this small section. We have to say that most of them, deal with a specific type of optimization and finally provide results to prove what they achieved. For instance, most hardware implementations that deal with throughput performance, aiming to reach the limits of highly demanding videos with upper limit the 8K @ 120 fps, keeping area and power as much low as possible.

Finally, except for custom-made RTLs, also have been proposed papers for implementing video codecs with HLS and now except to performance, area and power, man-month work is also used, in order to show how HLS tools can short time-to-market, thus showing its comparative advantage against custom RTLs. One HLS implementation for ASIC have been proposed regarding H.264/AVC codec [17] and to our knowledge this is the first effort that implements a module of HEVC with a HLS tool for FPGA.

1.4 Thesis Structure

This thesis is organized in several different chapters, each of them analyzes a small division or a theoretic background of our work. The outline of the thesis is organized as follows:

- Chapter 2 provides some background theory about video coding and finally it briefly presents the more important video codecs that have been used so far, in video applications. The objective of this chapter, only show the general concept that video codecs inherent through years without give a lot of details about each video standard. The reader may understand some fundamentals about video coding theory, having read this chapter, so to be able to follow up some basic notions in rest parts from this work.

- Chapter 3 presents forward and inverse integer transform algorithms and all the mathematical background behind them, aiming primarily on how they work. Also, discusses how we get faster computational versions of the same algorithm and how they help in video coding process, which is a critical task.

- Chapter 4 initially clarifies the idea of High Level Synthesis for digital circuits and why it is so valuable in industry. Additionally, Vivado HLS tool is presented extensively on how it works and what options can be selected in order to explore hardware design space of an algorithm, meeting different latencies.

- Chapter 5 shows the way we set up the experiment and how we are using Vivado HLS tool to obtain results and to see as much aspects of total design space of algorithm.

- Chapter 6 contains all results and is structured with several different results in tables and diagrams, thus helping to understand better how tool reacts on different inputs. Finally, throughput performance is extensively explored for all the different configurations and is compared with other implementations.

- Finally, Conclusion further discusses on results, paying attention on the big picture of the problem and makes a total inference on this thesis. Alongside, future work discusses what other surveys may follow up this work.

Chapter 2

Video Coding Background

As briefly discussed in introduction, video data has a quite big volume that leads in two important problems. At first, a large storage space has to be reserved in order to store a video file and secondly when we want to transmit a video sequence, we require huge bandwidth to do so. To better understand this problem, we present a simple example. A typical video movie has length roughly 90 minutes. If assumed HD resolution and frame rate at 30 fps, then we have 1920x1080x30x3x90x60 bytes to store information for 3 color channels (e.g. RGB or YUV) with 8-bit color depth. Thus, we need about 900 GB (1 TB is a typical hard disk size) to store a typical Blu-Ray movie, without include audio data. Now, one needs to transmit this content in a live streaming application, send 1920*1080*3*30*8 bits per second, in order to see video without stall effects. This volume is translated into 1.5 Gb/sec, which requires huge bandwidth that is difficult to be found in daily consumer products. Finally, according to Cisco surveys, 2 of 3 data packets that are send every time over internet network, belong to video content. Consequently, we realize that video compression is a big deal in our digital epoch and how it directly affects our lives, because video is everywhere among us.

2.1 Video in Signal Processing

Initially, compression algorithms can be distinguished in two categories according the type of elaboration that they perform on camera data. The two categories are called lossy and lossless compressors. Lossless compressors are those that reconstructed data on decoders side, are exactly equal with those that inserted as input in encoder side. Lossy compressors are those that reconstructed data, are slightly different from input, in that way that human eye cannot perceive it. Lossy compressors attain high compression rates and provide different levels of trade offs, between compression and reconstruction

quality. Every almost video standard that is used in products, utilizes a lossy codec, thus attaining great compaction results. In next sections, we are showing the basic stages that modern video codecs utilize, in order to compress video content.

A still image, is represented as a 2-D signal –in terms of signal processing– with one dimension denoting color change in horizontal direction and the other dimension, color change in vertical. In this approach, video is a 3-D signal, with 3^{rd} dimension being the temporal factor, to wit the color change between different frames in time, because video is actually a sequence of frames or still images. Video and image compression standards, exploit spatial and temporal correlation from frames, in order to compress data. If we carefully pay attention in ordinary images, we will realize that some parts of the image, have about same intensities with others and therefore image signal has a spatial correlation between different regions in image. In video, except for the spatial correlation in one frame of it, different neighboring frames are very similar between them and so video has a temporal correlation as well. Realizing this correlation, prediction algorithms can be performed in video codecs, so to predict some parts of video signal, thus do not requiring to send all information in decoder's side. Even more, video and image codecs exploit one more attribute that is based on a property of human's eye. Human eye cannot perceive high frequency changes in color, similar to ear which has a restricted bandwidth in acoustic frequencies. So, rejecting some of the high frequency components, we reduce information, without eye realize this degradation. In next Section 3, all these notions about frequency components, will be clarified further, to see how they are translated into signal processing.

2.2 Typical Compression Diagram

All renowned video compression standards that have been introduced so far, are based on a certain structure with same stages as shown in Fig. 2.1. The general scenario is the following. At first, a frame is declared as an intra or inter frame. In the former case, only spatial correlation is utilized to remove content redundancy, while in latter case, both spatio-temporal may be used. In either cases, an arrived frame get stored in frame buffer and is divided into small blocks of pixels. Each of the following stages from now on, refers to block operations. The first frame of video, must be declared as an intra (I-frame), because there aren't previous frames to make predictions, so it is encoded without having references from other frames. In Subsection 2.2.1 we provide further details on intra prediction. Other frames except intra, can be declared as P or B frames. P frames, use temporal prediction from previous frames, in order to reduce temporal redundancy, while Bi-directional frames are capable of using both previous

and future frames as reference, thus exploiting correlation from both future and past frames. Of course, future reference frames –from which B-frames take prediction– have to be decoded beforehand, so current B-frame have in memory the reference block of pixels, so to perform prediction – Fig. 2.2. Refer to Subsection 2.2.2 for further details on inter prediction.

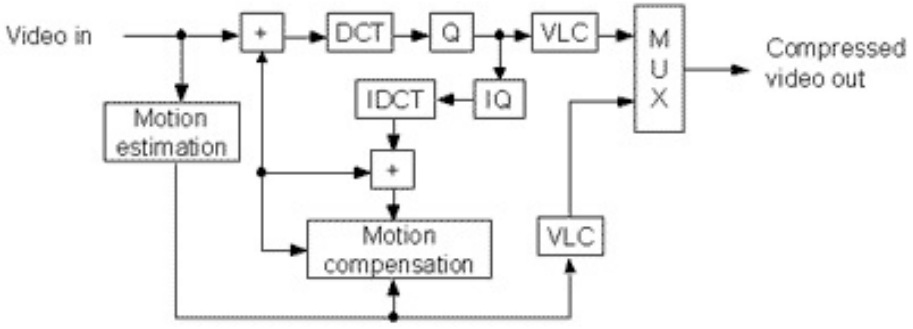

FIGURE 2.1: Architectural diagram of a typical video codec [44]

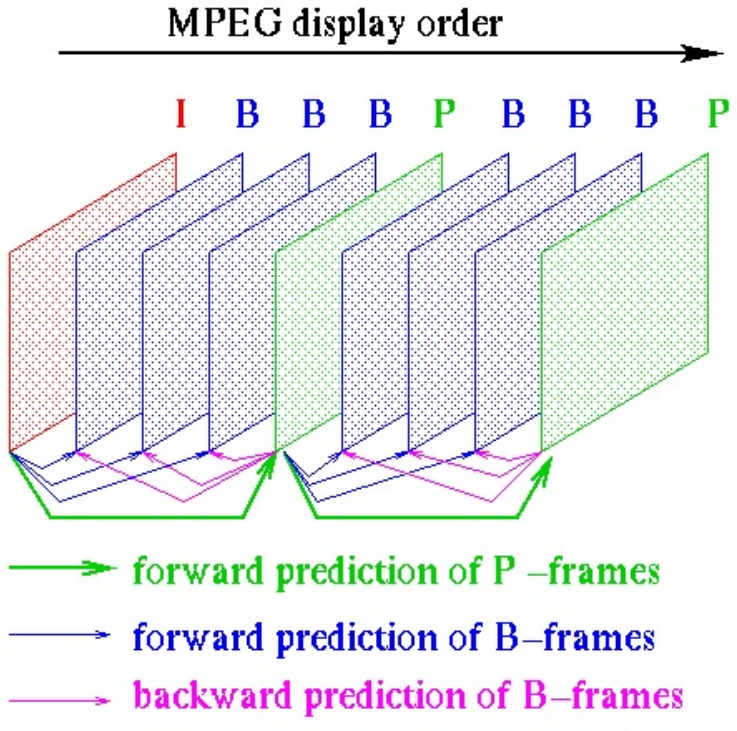

FIGURE 2.2: A group of pictures with I, P, B frames [39]

After removing redundancy for both intra and inter cases, we have the error that is

called residual of pixels or distance from prediction. The predictors from inter prediction are called motion vectors. Prediction error, will be send for transformation and quantization, in order to retain only low frequency components of error, thus requiring to send fewer information. Quantization process, introduces the lossy notion, because for those coefficients that have low energy in frequency domain, they will become zero. In this step, we have lost information, because decoder cannot retrieve zero coefficients in their primary value, before quantization step. First video standards, such as MPEG-1 and 2, that haven't exploited intra prediction, utilize transformation in a block of pixels –not in residual error– for I-frames. This concept that transforms a block of pixels without prediction and discard high frequency components, is used in image compression from JPEG standard (see Fig. 2.3). We have to say here that there are video codecs such as Motion-JPEG that do not exploit neither spatial nor temporal correlation. All frames are encoded as still images (JPEG coding is performed in each one) and only by rejecting high frequency components in block of pixels, we achieve some compression ratio. After all this procedure, the final stage of a video encoder in called entropy coding (Huffman, CAVLC, CABAC). The entropy module, undertakes the task to compress information according with the likelihood of each syntax element, which can be one of the following: motion vector, quantized coefficient, intra predictor, various indices and flags. Entropy encoder operates in bit level, using small length codewords for symbols with high likelihood and large codewords for more infrequent symbols.

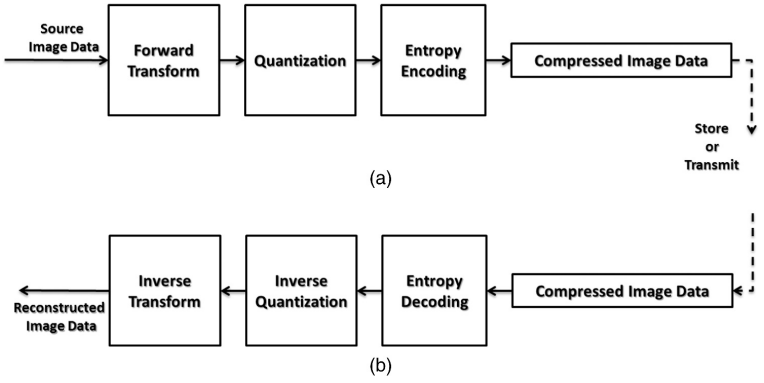

FIGURE 2.3: Basic stages of JPEG codec for still image compression [40]

Decoder, have to follow same steps in reverse order, starting with entropy decoding and so on. Encoder, have to decide about several parameters in order to encode a video, but decoder needs only to follow what encoder have decided on. This scenario and the "communication" between encoder and decoder, is indicated via encoded bitstream. Hence, video encoders are considerably more complex than decoders, due to many decisions that they have to try. Also, some high complex algorithms, such as motion estimation, performed in encoder's side, so they increase further the computational complexity of encoder.

A very strong feature that is met in later video standards, is some filters from image and video processing fields, that their task is to remove blocking artifacts that video codecs introduce, due to the block-based structure that they have. The in-loop filter or else de-blocking filter, applies a filtering in all vertical and horizontal edges, thus removing blocking artifacts. Other such filters that have been introduced in HEVC standard like Sample Adaptive Offset (SAO) filter [20], gives an offset in pixel values, after reconstruction process in decoder side. Visual results in order to compare differences are illustrated in Figures 2.4 and 2.5.

FIGURE 2.4: Deblocking Filter [41]

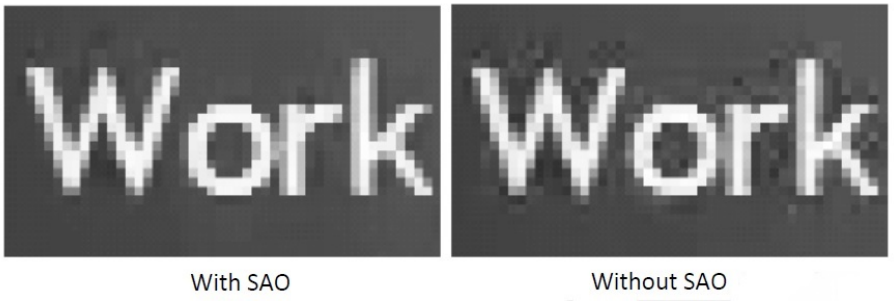

FIGURE 2.5: Sample Adaptive Offset (SAO) filter [20]

2.2.1 Spatial Correlation

Spatial correlation, exploited at first in H.264/AVC standard. Until then, only temporal was exploited by video codecs and so I-frames (intra frames or frames without temporal prediction) were using only transformation and quantization in block of pixels, in order to reduce input information. The basic idea is that in many frames, there are some regions that can be predicted from other already decoded. Therefore, some blocks can be predicted from other co-located blocks, according to a certain direction. Direction,

indicates the algorithm that we take pixels from up and left blocks and how we use them in order to best predict pixels in our current block. Let us assume that a quite spread area into one frame has about the same color information. Then, it is easy to predict some blocks from other neighboring already decoded blocks, just by coping pixel information either in horizontal or in vertical direction. Of course, there will be an error from prediction that is going to be transformed and quantized. Fig. 2.6, shows the nine different possible directions for intra prediction that is maintained in AVC codec. We can see vertical prediction (just copying information from upper adjacent block), horizontal (from left block), diagonal predictions with different angles and finally DC prediction that finds the mean, between the two rows of pixels. In real life, most of real objects have vertical correlation, so we can notice that respective mode has number zero, because this is the number that has the smallest entropy information in a video codec. Hence, modes with high likelihood are represented by numbers with small entropy in a video codec, as long as it is a rational practice, in order to achieve high compression.

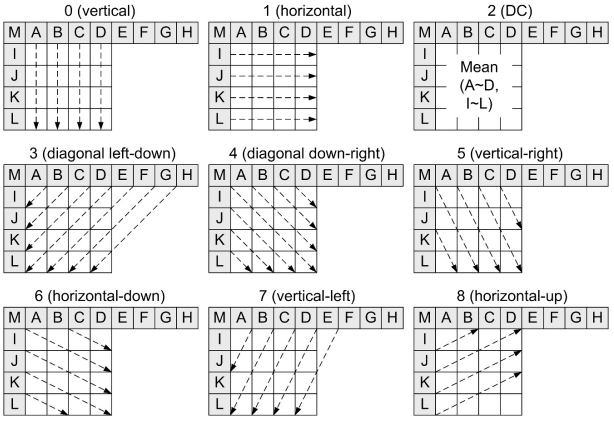

FIGURE 2.6: Possible directions for intra prediction in H.264/AVC standard [2]

As long as a frame is declared as an I-frame in the GOB, then its blocks proceed for finding best intra prediction mode. First intra block (upper-left in frame) that has no predictors is the only that in encoded just by transformation and quantization on pixel intensities, like to JPEG. A greedy approach, it could be this: try all possible directions and find what is this with the smallest prediction error. Although, some early termination algorithms can be utilized, so under a threshold condition according to Mean Square Error (MSE) value, prediction is over. Finally, intra prediction can be used also from inter frames, because if one block cannot be predicted well from other frames, then an intra block may be used to give better prediction. If so, this block in the inter frame, has to be declared as intra via some flag, so decoder knowing the predictors, in the reconstruction process of the block.

2.2.2 Temporal Correlation

Temporal correlation is a common attribute in video sequences and thus it is exploited largely from video codecs, for reducing the coding information. The third dimension of a video signal is the temporal factor and as already mentioned, different frames in time have a strong correlation between them. Temporal correlation is occurred in video signals, due to the short time that frames are captured from the camera. Capturing a video at 30 fps for instance, means that a new frame is captured every 33 millisecond. It is rather straightforward to see that those frames, shall have a strong relationship between them and temporal prediction can be used to predict each other.

A simple approach of such a procedure is this: a current block that is going to be predicted is searched in different frames in a certain search range and the frame with the smallest error, is declared as best. The motion vector that refers to the best block, is transmitted to decoder's side. Motion vector, is a vector in (X, Y) format that declares how far from current block we have to go, in order to find the best predicted block. Additionally, an index is encoded in bitstream, which declares from which frame we have used the block for prediction. In Bi-directional frames, there is also option to find two prediction blocks, from different temporal frames and find their average or weighted average with some pre-defined weights, thus constructing the prediction block that will be used to calculate the prediction error.

The module that elaborates the previous demanding task, is the popular motion estimation algorithm, which is the most complex algorithm in a video encoder, since it takes many cycles to find the best motion vector. The greedy algorithm or else the full search, takes all possible blocks in a specific search range and finds the best error between them in terms of MSE. As we can realize, operations are performed pixel by pixel for the entire block, thus making motion estimation a computational demanding algorithm. Some researchers such as [22], have proposed different schemes for early termination, making trade-offs, between time for prediction accuracy. After motion estimation is accomplished, a motion vector, an index and a residual block (prediction error), are yielded from this module. Fig. 2.7 illustrates motion estimation between two frames.

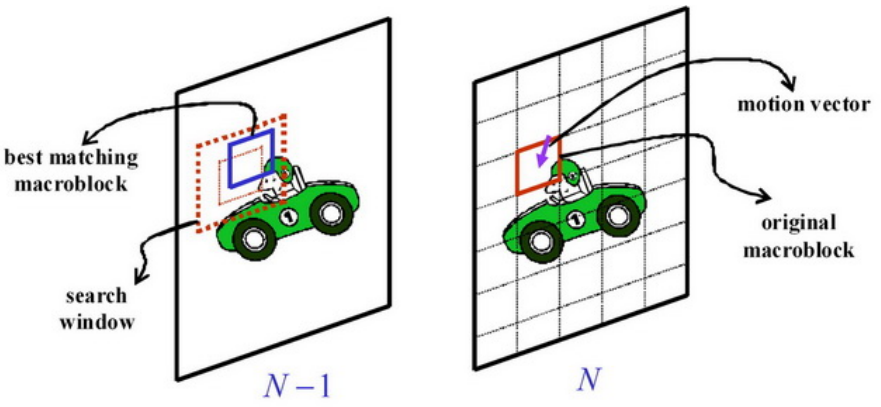

FIGURE 2.7: Motion estimation algorithm. Finds best match block in different temporal frames. Motion vector indicates how far from current position best block is located [42]

In typical videos, the very smooth motion that exists frame by frame, induces another attribute that video codecs take into account. In actual video sequences, there is a big probability that motion doesn't matches so well on integer pixel distance. What we are trying to say, many times block's motion, doesn't match with integer pixels, because motion goes in sub-pixel distances and therefore integer block options, don't give as much accurate prediction as it could, if sub-pixel values were exploited.

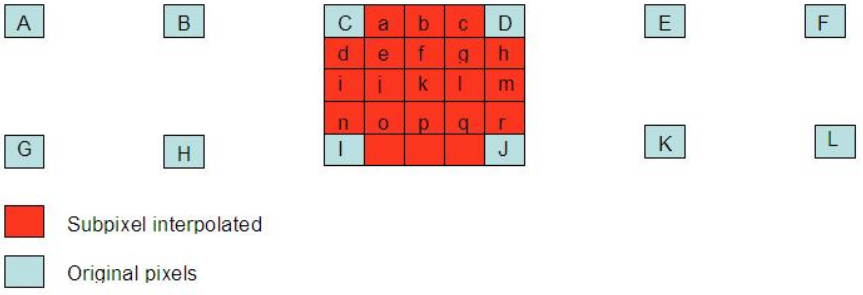

FIGURE 2.8: All possible sub-pixel values that can be found in quarter distance. Different filters are used to obtain values in each position [43]

Motion in sub-pixel values can be captured only by moving in distance lower than pixel. Of course, a prediction block in half pixel distance is not in memory, because only integer pixel values from already decoded frames are in there. So, in order to find new pixels in sub-pixel distance, interpolation process has to be performed, with input the integer pixel values from integer part of motion vector. Most of video codecs are using quarter pixel interpolation values, thus enabling accuracy at quarter pixel distance. Fig. 2.8, shows all quarter pixel positions that can be found from already decoded integer values. Each new video codec, adopts new techniques on interpolation method. As the accuracy of interpolation goes higher, it leads to better coding efficiency because energy of motion's prediction error gets decreased.

TABLE 2.1: Popular video standards that have been used in video applications

Year	Standard	Applications	Bitrate (Mbps) (720x480)
1993	MPEG-1	VCD	7
1995	MPEG-2	DVD	6
1999	MPEG-4	DivX, XVid	5
2003	H.264/AVC	BluRay, DVB-TS	4
2013	H.265/HEVC	next of H.264	2

2.3 Video Standards

Video standards through years, aim on better coding efficiency, so to delivering video at lower bitrate, trying to retain a good quality in video content in terms of PSNR. Peak Signal to Noise Ratio (PSNR) is a metric that evaluates how much faithful is the reconstructed video sequence after the video decoding process. PSNR, in terms of signal processing, is the amplitude of true video signal in respect to noise. Noise in video is declared as the MSE, between pre-encoded and post-decoded frames. So, each new video coding standard, aims on achieving better PSNR for same bitrate or reduced bitrate for same PSNR. In next subsections 2.3.1, 2.3.2, 2.3.3 and 2.3.4, we shortly present some of the most popular video standards that have been used in daily consumer products so far 2.1, via architectural diagrams and their key innovations.

2.3.1 MPEG-1/2

MPEG-1 is the first video codec that exploited temporal correlation using motion estimation techniques. I-frames, don't have spatial prediction and are coded like JPEG, only using transformation and quantization to reduce information before entropy coding. P and B frames, use motion estimation in order to find best prediction block and in these frames, only error of prediction is sending to decoder, along with motion vectors. MPEG-1, utilizes Huffman algorithm [21] for the entropy coding stage. MPEG-2, has small differences when compared to MPEG-1. Different scanning order of quantized coefficients, standard half-pel motion estimation and support for other color formats, are some of the small difference between the two standards. A typical diagram of an MPEG-1/2 video codec, is depicted on Fig. 2.9.

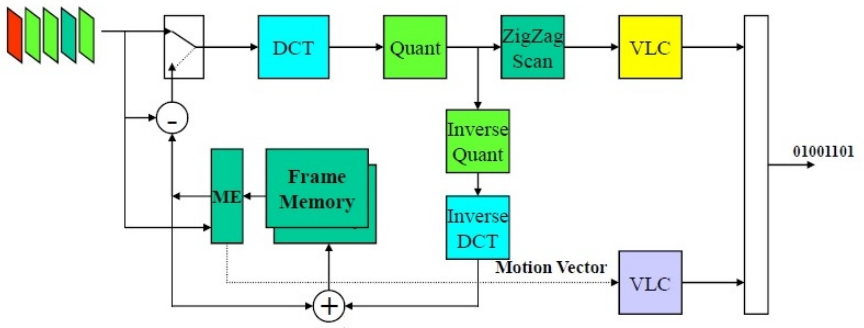

FIGURE 2.9: Typical diagram of an MPEG-1/2 encoder [44]

2.3.2 MPEG-4

MPEG-4 is the most interesting video standard from academic and research aspect. The video coding process, differs a lot in comparison with other standards, because everything is consisted from multimedia objects and background. True objects, faces and mesh can be considered as multimedia objects and also transparency of each object, can be used in coding process. Scalability of video content is used as well, either spatial or temporal, thus enabling video delivery in different bandwidths and qualities. MPEG-4 supports now color bit-depths from 4 up to 12 bits, in comparison with MPEG-1/2, where only 8-bit was permitted. Quarter pixel accuracy in interpolation, is now an option that leads to better coding efficiency. Besides, there are schemes for prediction on DC and AC coefficients, among adjacent transform blocks. Finally, a great advantage of MPEG-4, is the error resilience tools and techniques that utilizes, in order to be more robust in errors that are introduced in video streaming over networks.

2.3.3 H.264/AVC

H.264 or Advanced Video Coding (AVC) is also known as MPEG-4 part 10, because it was developed as an amendement of MPEG-4 standard. Here, coding methods return back into block-based structures, without having any more notions, such as multimedia object, background and transparency.

Spatial correlation is exploited for the first time and is called intra prediction, following what we described on Subsection 2.2.1, giving up to four times better compression in I-frames. Quarter pixel motion accuracy, is now a standard method for more accurate motion prediction, thus providing better coding efficiency. A 6-tap sync-based FIR filter, is now used for half pixel values; for quarter values, a bi-linear filter is used, taking half-

and integer-pixel values as inputs. Blocks of pixels, have also greater degree of freedom, for partitioning into smaller blocks, giving more accurate prediction. Additionally, DCT transform has altered in integer transform with the same properties, but using now only integer arithmetic, avoids rounding errors between encoder and decoder. Moreover, a deblocking filter is used for the first time in order to alleviate blocking artifacts as explained in Section 2.2. Finally, except for Huffman entropy encoder, now there is option for CABAC, which has about 15% better compression performance, since it is a superior entropy algorithm than Huffman, in terms of coding theory.

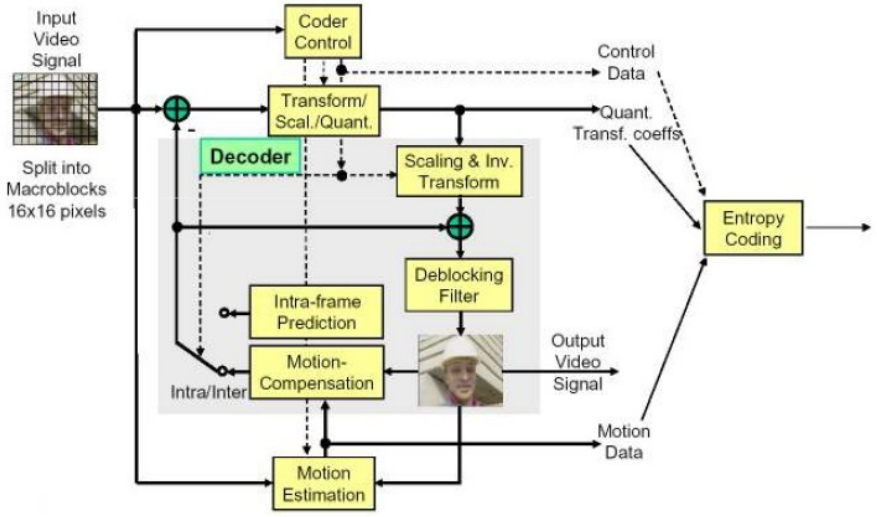

FIGURE 2.10: Typical diagram of an H.264/AVC codec [2]

2.3.4 H.265/HEVC

The latest video standard is the H.265 or High Efficiency Video Coding (HEVC), which is the AVC's predecessor and is expected to be adopted in future multimedia products, because it reduces bitrates at half compared to its predecessor. This directly implies that better video quality can be delivered over the same bitrate or for the same video quality, bitrate can be halved.

HEVC has further improved the block-based video coding structure that existed so far, adopting a quad-tree structure called Coding Tree Unit (CTU) that starts from the largest block of pixels (typically 64x64) and recursively splits into smaller blocks for prediction (Prediction Unit - PU) and transformation (Transform Unit - TU). Moreover, blocks not only allow symmetric partitions but asymmetric ones are also utilized, allowing for a better match with actual visual-object shapes, thus reducing motion residual energy [23]. Interpolation filters, has quarter-pel accuracy with longer tap FIR filters for improved prediction. AVC was using only 4x4 and 8x8 integer transform sizes, while

in HEVC, 16x16 and 32x32 have been introduced, enabling higher energy compaction in high resolution videos. Here, CABAC is standard algorithm for entropy coding module and also besides to deblocking filter, SAO [20] is also used, to ameliorate the quality of reconstructed frames. The diagram of HEVC is presented in Fig. 2.11.

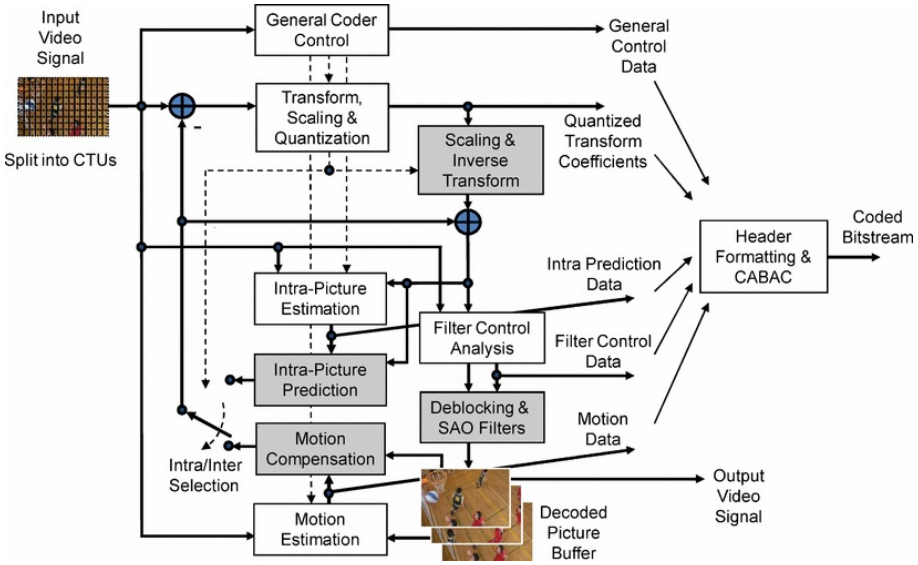

FIGURE 2.11: Typical diagram of an H.265/HEVC codec [1]

Chapter 3

HEVC Inverse Integer Transform

HEVC integer transform, is the module that undertakes the task to change a block of samples from spatial to frequency domain. Therefore, forward integer transform is used by encoder, in order to evaluate frequency components of a block of samples and how much energy has each of them. Inverse integer transform, is the inverse procedure that takes coefficients and converts them back to spatial domain, thus reconstructing pixel information at decoder's side. The integer transform module, uses essentially the same algorithm with Discrete Cosine Transform (DCT), but it manipulates only integer arithmetic instead of floating point, so to avoid rounding errors that leads on a slight mismatch between encoder and decoder. Video standards up to MPEG-4, use DCT instead of integer transform, but after that, only integer transform is used.

There are several transforms in general, out of DCT, that are used in order to decompose samples from spatial to frequency domain. Karhunen-Love Transform (KLT) [24] is a unitary and orthogonal transform that attains best energy compaction among all, but its high complexity, constraints the implementation for real-time applications. Discrete Fourier Transform (DFT) [25] is a separable transform for different dimensions. It is also a unitary and orthogonal transform that is used to decompose the original data into its sine and cosine components. DCT transform now, belongs to Fourier-family transforms, because it is essentially the even part of a DFT, so it is also a separable transform that we are going to analyze in this Chapter. Hadamard Transform [27] is a simple low complex algorithm, but it achieves moderate energy compaction and is used from video codecs in very special cases. Finally, the Discrete Wavelet Transform (DWT) [30] is a unitary, orthogonal and separable transform that is usually applied to the whole input data (or large parts of it, called tiles) but typically not to small data blocks like all the previous transforms.

This Chapter is organized as follows: Section 3.1, shows the basics about DFT algorithm, because DCT and therefore integer transform, is based on it and is going to help in understanding how DCT was created. Section 3.2 shows the DCT forward and inverse algorithm that is used in many different video and image compression standards. All algorithms are based on 2-D transforms, because video standards are using 2-D transforms for the block of pixels, thus capturing both horizontal and vertical signal change. Final Section 3.3 shows how a Fast Fourier Transform (FFT) is constructed from DFT and in respect to this method, DCT fast version is deployed as well, which is utilized in every real-time video application.

3.1 Discrete Fourier Transform (DFT)

As previously mentioned in this chapter, DFT is a separable orthogonal transform, that converts input data into its sine and cosine components. 2-D algorithm is the same as two 1-D transforms in row, with the first dimension taking into account horizontal frequencies and the second the vertical ones. Calculation of 2-D forward and inverse DFT is based on Equations 1 and 2 respectively.

$$y(k,l) = \frac{1}{N}\left(\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} x(m,n)e^{-\frac{2\pi i(km+ln)}{N}}\right) \tag{1}$$

$$x(m,n) = \frac{1}{N}\left(\sum_{k=0}^{N-1}\sum_{l=0}^{N-1} y(k,l)e^{\frac{2\pi i(km+ln)}{N}}\right) \tag{2}$$

In either equations, $x(m,n)$ represents a block of pixel data which is a 2-D signal and $y(k,l)$ the output coefficients, each of them representing the energy of a basis frequency function, according to its position. The $y(0,0)$ coefficient is called DC, because it represents the energy of zero frequency in both horizontal and vertical direction. So, if all pixels in a block have equal values, then all coefficients except for DC become zero and DC's energy maximized, according of course to the sample's intensities. Other coefficients than DC, are called AC. Figure 3.1, shows basis functions for each different frequency component. We can see DC component in upper left corner and also how horizontal signal frequency increases, while moving left-wise and how vertical increases for down-wise scanning.

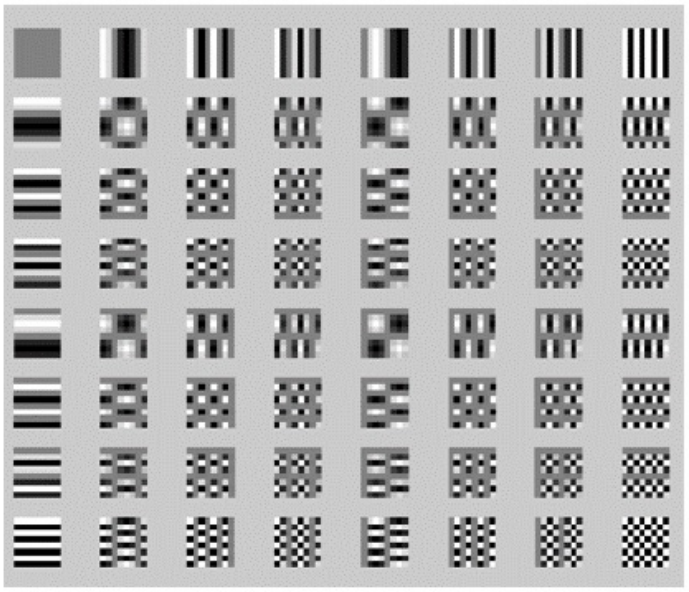

FIGURE 3.1: 8×8 DFT basis functions [28]

It is straightforward to see someone that DFT produces complex coefficients, with real and imaginary parts, to wit magnitude and phase, The storage and manipulation of these complex values it is a disadvantage when compared to other available transforms, e.g. the DCT which use real and not complex numbers. It is a much better solution than DFT for real implementations, achieving also better energy compaction for highly correlated signals, such as image. Higher energy compaction means that with fewer coefficients we reconstruct signal with less error than DFT. The main reason DCT is used in video codecs, is that a lot of coefficients will be discarded in quantization process and therefore we want to reconstruct as better as possible the signal with fewer coefficients. Figure 3.2, illustrates the main difference between DFT and DCT, as concerning energy compaction and reconstruction with fewer coefficients.

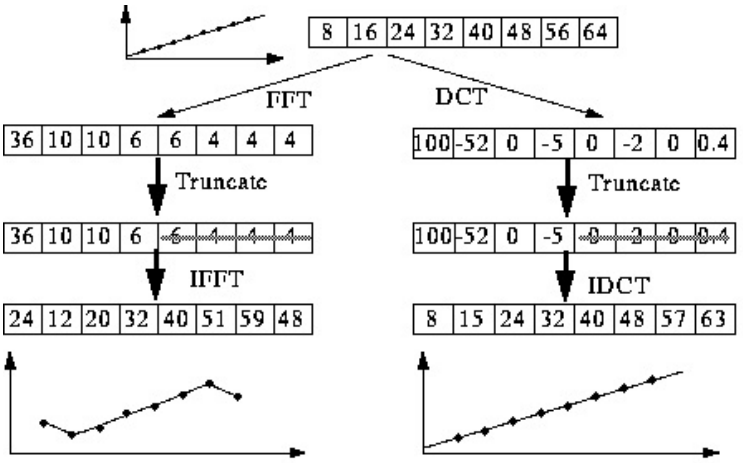

FIGURE 3.2: DFT vs DCT in terms of signal reconstruction [29]

3.2 Discrete Cosine Transform (DCT)

The Discrete Cosine Transform (DCT) is a unitary and orthogonal transform, conceptually rather similar to the DFT, but only using real numbers (and not complexes any more). For a NxN block of samples, the forward 2-D DCT is defined by Equation 3

$$y(k,l) = \frac{4C(k)C(l)}{N^2}\left(\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} x(m,n)\cos\frac{(2m+1)k\pi}{2N}\cos\frac{(2n+1)l\pi}{2N}\right) \tag{3}$$

and the inverse 2-D DCT is defined by 4.

$$x(m,n) = \left(\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} C(k)C(l)y(k,l)\cos\frac{(2m+1)k\pi}{2N}\cos\frac{(2n+1)l\pi}{2N}\right) \tag{4}$$

$$C(\omega) = \begin{cases} \frac{1}{\sqrt{2}} & \omega = 0 \\ 1 & \omega = 1,2,\ldots,n-1 \end{cases} \tag{5}$$

Like the DFT, since the DCT is also a separable transform, it can be represented as the product of two 1-D DCTs; the first for the 1-D horizontal and the second for the vertical. The 2-D basis functions of DCT are presented in Fig. 3.3. Since the cosine function is real and even, i.e., $\cos(x) = \cos(-x)$ and the input signal is also real, the inverse DCT generates a function that is even and periodic in 2N, considering N the length of the

original signal sequence. In contrast, the inverse DFT produces a reconstruction signal that is periodic in N.

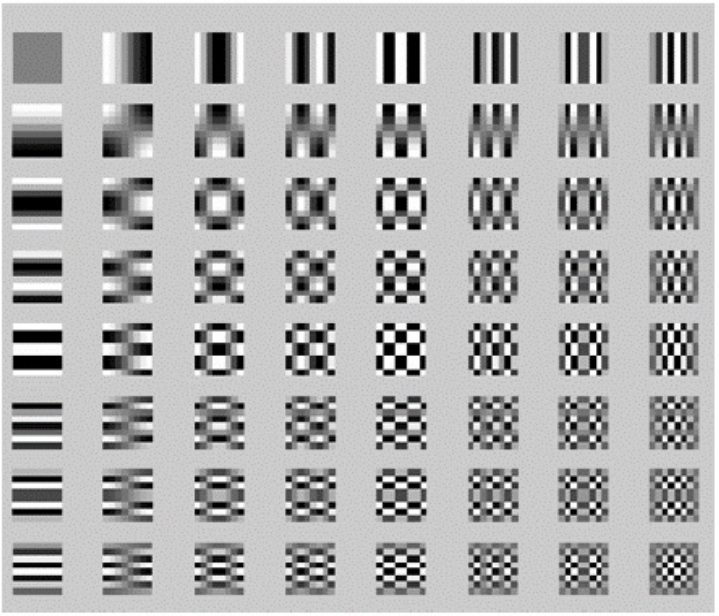

FIGURE 3.3: 8×8 DCT basis functions [28]

In other representation, DCT can be declared also as a multiply of two 2-D matrices, each one for 1-D stage of transform. The basic algorithm –not the fast version– that is incorporated in video codecs, is essentially the product of three 2-D matrices: two of them contain the basis of DCT and the third represents the input signal (block of pixels). Equation 6, shows the procedure of a 2-D DCT transform. B is the NxN matrix with transformed coefficients, A is the input NxN pixels or residuals and U the NxN basis components of DCT. We can see briefly that inverse transform procedure is this: a block of coefficients is arrived and 1-D transform is applied in each of its rows capturing horizontal frequency. After that, coefficients from first stage will be the input after transposition of the second step of transform. The output result from the second transform, is the 2-D transform of a NxN block of pixels.

$$B = UAU^T \tag{6}$$

Integer transform of HEVC is essentially the same algorithm with DCT, but U matrices contains only integer values –not real numbers– making an approximation to basis functions. As we said, DCT and therefore integer transform is an orthogonal transform and this is the reason that HEVC contains four such transforms the $4x4$, $8x8$, $16x16$ and

$32x23$. These four different transforms apply usually to residual of pixels, so to convert them in frequency domain and by quantization, to discard high frequency components of error. The bigger size of transform is used, the better energy compaction is achieved for large blocks of pixels. A typical 4x4 block of pixels-residuals, can be described by 2-3 coefficients if prediction is accurate and error has low energy. We can see that sending three coefficients, we can reconstruct sixteen pixels. Now, a typical 32x32 block can be described by ten about coefficients, thus letting to retrieve 1024 pixels sending only ten coefficients and that is why better energy compaction is achieved.

3.3 Fast Transform Implementation

Having seen the DCT algorithm through equations, it is easy to realize that in order to transform a NxN block, a computer have to perform N^2 operations (multiplications and additions) for the 1-D stage and one more time the same computations for the second stage (2-D). So, the complexity of DCT via matrix multiplications is $O(N^2)$, which is a prohibited complexity for real-time applications. Especially in HEVC standard, the complexity would be very high for the two large transforms ($16x16$ and $32x32$), thus making difficult the optimization of integer transform module.

Several algorithms have been proposed all those years which reduce DFT-family algorithm's complexity. The most famous technique-algorithm was carried out from Cooley-Tukey and the relevant paper was published in 1965 [36]. This is a divide and conquer algorithm that recursively breaks down a DFT of any composite size $N = N1N2$, into many smaller DFTs of sizes N1 and N2, along with $O(N)$ multiplications. The best known use of the CooleyTukey algorithm, is to divide the transform into two pieces of size $N/2$ at each step, (also known as radix-n, where n are the steps) and is therefore limited to a power-of-two sizes, but any factorization can be used in general. The two pieces of $N/2$ transforms, are consisted from the even entries for the first transform and the odd ones for the second divided transform.

Figure 3.4, shows an 8-point DFT with a radix-4 scheme, according to CooleyTukey's algorithm, that splits in smaller transforms up to 2-point DFT. All these diagrams are called butterfly schemes due to their shapes. The butterfly scheme of the 2-point DFT, is illustrated in Fig. 3.5. Now regarding the total complexity of the fast algorithm, it is easy to see that each N-point or each radix, requires N multiplications and additions. Having $\log N$ stages for each fast implementation, the total complexity for the 1-D transform becomes $N \log N$ and $2N \log N$ for the 2-D transform, since it is separable. By converting a DFT algorithm in FFT, one we achieve, reduce the order's complexity from $O(N^2)$ to $O(N \log N)$ which is a very good performance for an algorithm that

will be incorporated into a real-time application and also it enhances significantly the performance of larger transforms such as 16x16 and 32x32. The complete fast DCT diagram on which our implementation is based on is depicted on Fig. 3.6.

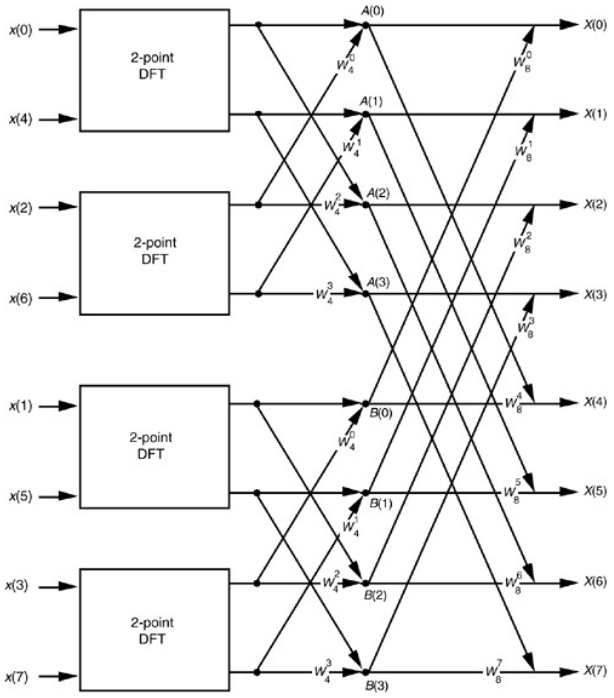

FIGURE 3.4: Cooley-Tukey algorithm with radix-4 [26]

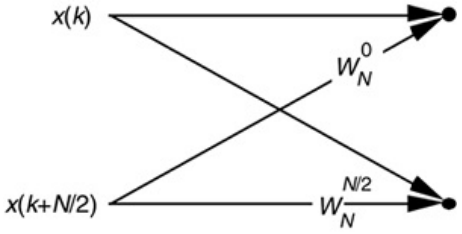

FIGURE 3.5: Radix–2 bytterfly [26]

We shown how the DFT algorithm can be modified into FFT, using some techniques in order to reduce its complexity. On exactly the same way, DCT transform is also optimized in order to obtain a version of algorithm that can be used from demanding applications. For DCT and so for integer transform, Chen's algorithm [38] is utilized in order to create a more efficient in terms of complexity algorithm that can be used from video encoding-decoding applications. In HM reference software, the standard

algorithm that is utilized for integer transform, is based on Chen's algorithm which contains the $4x4$, $8x8$, $16x16$ and $32x32$ transforms. The multiplicands are contained in separate arrays for each size of transform and as we have already said, they are integer approximations of DCT's ones.

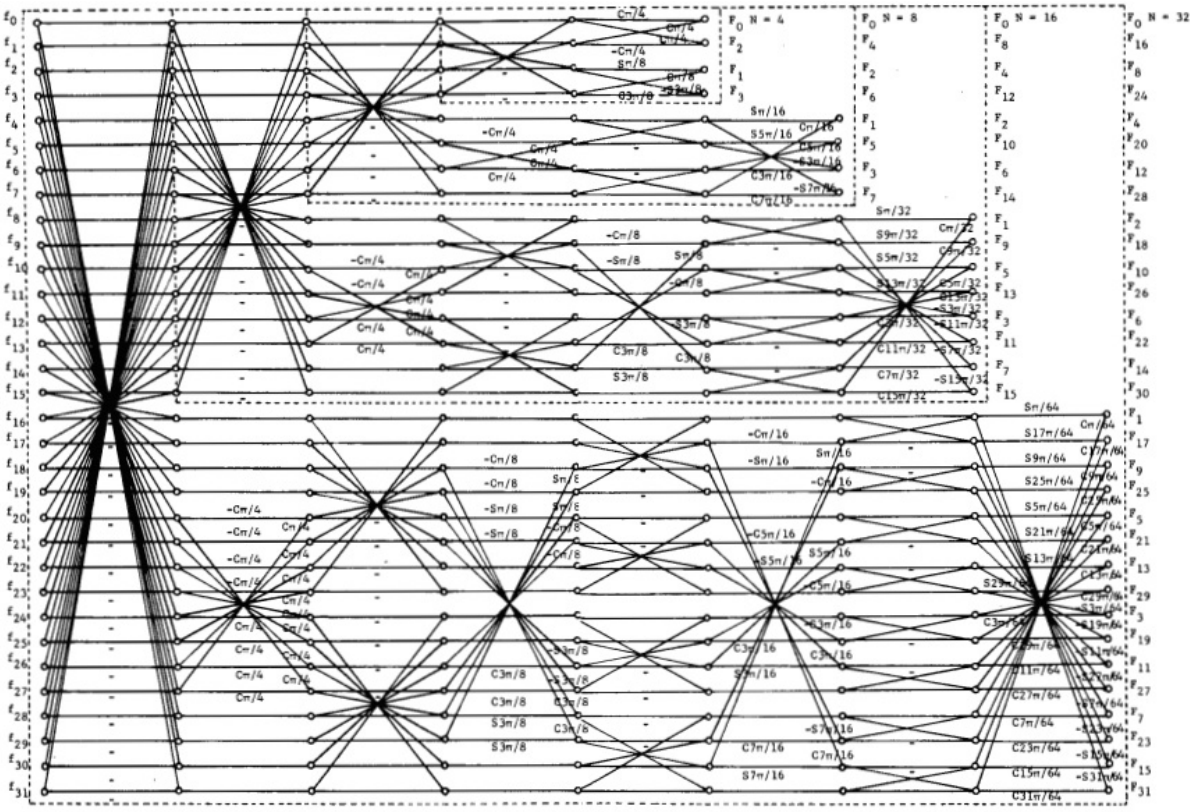

FIGURE 3.6: Signal flow graph of Chens fast factorization for 4x4, 8x8, 16x16 and 32x32 transforms [38]

Chapter 4

High Level Synthesis on FPGA

The growing capabilities of silicon technology and the increasing complexity of applications in recent decades, have forced design methodologies and tools to move in higher abstraction levels. Raising the abstraction level and accelerating automation of both the synthesis and the verification processes, has allowed designers to explore the design space more efficiently and rapidly (shorter time-to-market). Essentially, the most valuable feature of HLS and that is why industry have started to explore further, is the short time requirement for developing an algorithm into hardware including synthesis and verification processes. As it is already known, an algorithm can be mapped onto a hardware design with different architectural ways considering performance, area and power. This is called design space of a certain algorithm, changing several hardware architectural options, in order to make different Register Transfer Levels (RTLs) designs of a specific algorithm. HLS tools, are very efficient in this approach, because given an algorithm description and only changing directives, different RTL designs are produced. Custom RTL designs, are written by hand with Hardware Description Languages (HDL) code (Verilog, VHDL), so we have to write down a new code each time, if we want to explore a different RTL design. Hence, HLS maintains more efficiency for exploring design space, spending fewer time than classic logic synthesis approach.

In this chapter, we initially present an introduction Section 4.1 about HLS, in order to clarify what is the general concept of HLS and what means "raising the abstraction level". After that, in Section 4.2, we briefly introduce Vivado HLS tool, which is a tool for HLS on FPGAs and is available from Xilinx corporation. We explain a few things about tool's structure and how it manipulates designs, according to the directives that are inserted. The most useful directives of Vivado HLS, are presented in Subsection 4.2.1, since some of them were used in our experiment. Finally, in Subsection 4.2.2 we explain how Vivado controls latency, according to the directives that inserted.

4.1 Introduction

High Level Synthesis, implies a general concept that have been introduced so far, for both software and hardware developments techniques, despite the matter of fact that formally it refers to hardware implementations. For example, in software domain until 1950's engineers were writing directly machine code (bit-level). In 1950's assembly language was introduced and assembler had the task to translate into machine code. Furthermore, after 1960's first programming languages were utilized for programming a machine. Languages such as C/C++, Pascal, Lisp and many others, use commands more close to human cognition and other software tools (compilers), undertake the task to produce assembly and after that machine code. When someone writes source code into C for instance, in fact he does not know what exactly will be the machine code that will be executed. Compiler makes a lot of platform based optimizations, in order to produce more efficient assembly code. Actually, programmer is based on compiler program, thus giving him an efficient and functional correct binary code.

In hardware domain now, first Integrated Circuits (ICs) were designed, optimized and laid out by hand. After that, in 1970's first gate-level and cycle-accurate simulation tools enhanced circuit's design process making more easy the verification which is a vital factor in hardware design flow. After 1980's HDL languages were developed in order to automate the design of a hardware implementation. Engineers now describe through HDL a specific hardware design that they have decided on and a logic synthesis tool converts HDL into netlist, to wit gates and wires interconnected each other. Additionally, except for logic synthesis several other tools have been developed such as place-and-route, timing analysis and formal verification that facilitate and automate the hardware VLSI design flow.

Using a specific technology library, the Logic Synthesis tool performs the process of the mathematical transform of RTL description, into a technology-dependent netlist. This process, is analogous to a software compiler that converts a high-level C-program, into a processor-dependent assembly-language. Each logic synthesis tool makes some mathematical transformations in boolean function of circuit, according to area, power and delay constraints that have been inserted. If high frequency is required then tool shall trade area (area expands exponentially in order to reduce logic levels) or else logic levels will be increased in order to share hardware, thus saving area. In every case of a function transformation, the boolean function will be exactly the same in every different trade off. Hence, logic synthesis tools explore a very small area of whole design space of an algorithm, because having determined RTL, they only explore boolean transformations into a small portion of global design space.

HLS concept, moves one level higher than logic synthesis, because through HLS we describe algorithm not RTL (not specific architectural design), thus giving more space into tool to explore other RTL that satisfy a particular algorithm and meet some specification constraints better. After desired RTL –which meet some specification requirements– is exported, it can be inserted into a logic synthesis tool, for mapping design into a netlist, under some area–delay constraints. Fig. 4.1 illustrates abstraction layers and how HLS describes algorithm, –not design– thus enabling more efficiency in terms of time for design space exploration.

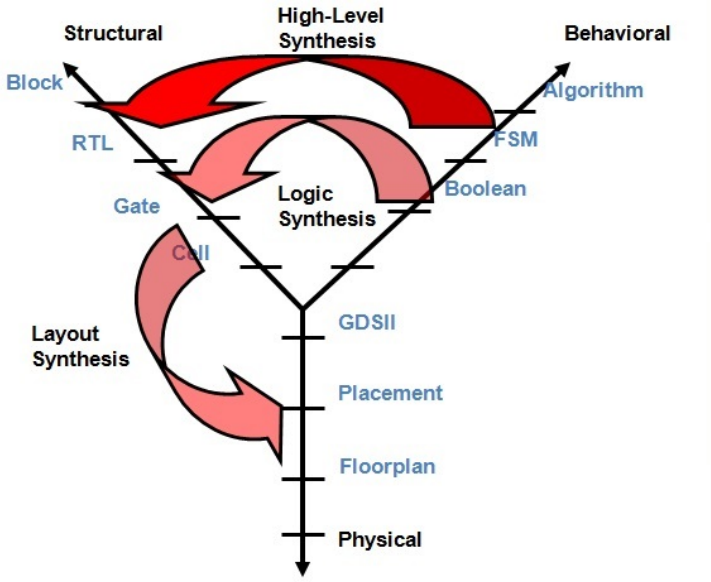

FIGURE 4.1: Abstraction layers on digital circuit design [33]

HLS tools in general take as input a source code in C/C++ or System C and output a specific RTL (Verilog and VHDL code), according to the directives that inserted. Directives are declared as some definitions and directions for HLS tool, thus helping it to output RTL under some specifications. For example, if we want a pipeline or a fully parallel RTL implementation of our input algorithm, we have to insert some specific directives into tool, so to know on what architectural plan we are aiming to, thus trying to best meet our requirements. Several different directives can be given describing the way that HLS tool will produce RTL, as we are going to see at next. Along with RTL, a HLS tool also produces a report with results, regarding latency, area or device utilization and delay that was achieved. Finally, we have to say that a typical HLS tool, aims either on ASIC designs or FPGA ones. Both are based on the same flow and same concept is utilized; having although different implementation objectives, hardware resources and

technology libraries, the outputted RTL may be different. Vivado HLS, aims to FPGA implementations and all the optimizations and reports are based on the target FPGA device.

4.1.1 HLS flow

Every almost HLS tool is based on a certain flow. Initially, given a C/C++ source code, tool makes a parsing in code and compiles the specification. In doing so, it represents source algorithm in a Control Data Flow Graph (CDFG) in a more formal model after parsing. Having this model, allocation of hardware resources takes place according to a standard input library that tool is based on. After that, scheduling is performed in order to assign different operations in clock cycles (see Fig. 4.2). Furthermore, binding process, binds different operations with already allocated functional units and also binds variables in storage elements (FIFOs memories) and transfers into buses. In the final stage, some architectural optimizations take place according to the directives that have been introduced in tool, thus creating final RTL architecture close to user specifications. Fig. 4.3 briefly illustrates a typical HLS flow.

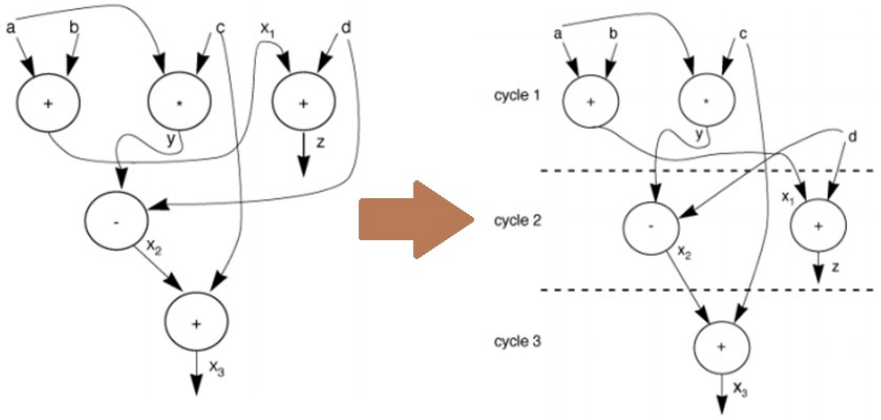

FIGURE 4.2: Different operations are scheduled in clock cycles [34]

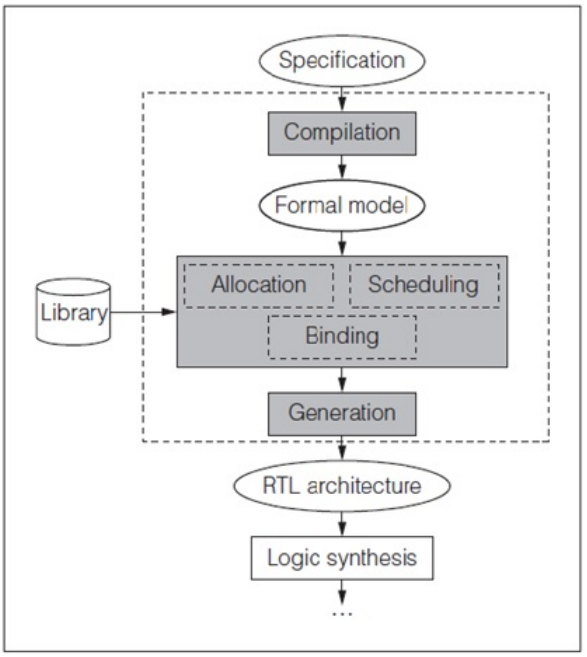

FIGURE 4.3: High Level Synthesis general flow diagram [35]

Every RTL that is produced from a HLS process, is consisted from two distinctive parts. The first one ,as already described, is the data path that produces calculation results and is consisted from classic hardware components such as: MUXs, ALUs, memories, arithmetic modules, buses e.t.c. The second basic element of a RTL from HLS, is the Finite State Machine (FSM) that controls the data path according to input signals. FSM also controls output signals, thus providing a complete interface on top-module, since it is considered as a separate IP hardware block – Fig. 4.4. This FSM is also manipulated some times as a counter, in order to count latency for different data paths in circuit, as we are going to see at next. According to these counters, FSM controls interface signals of separate modules.

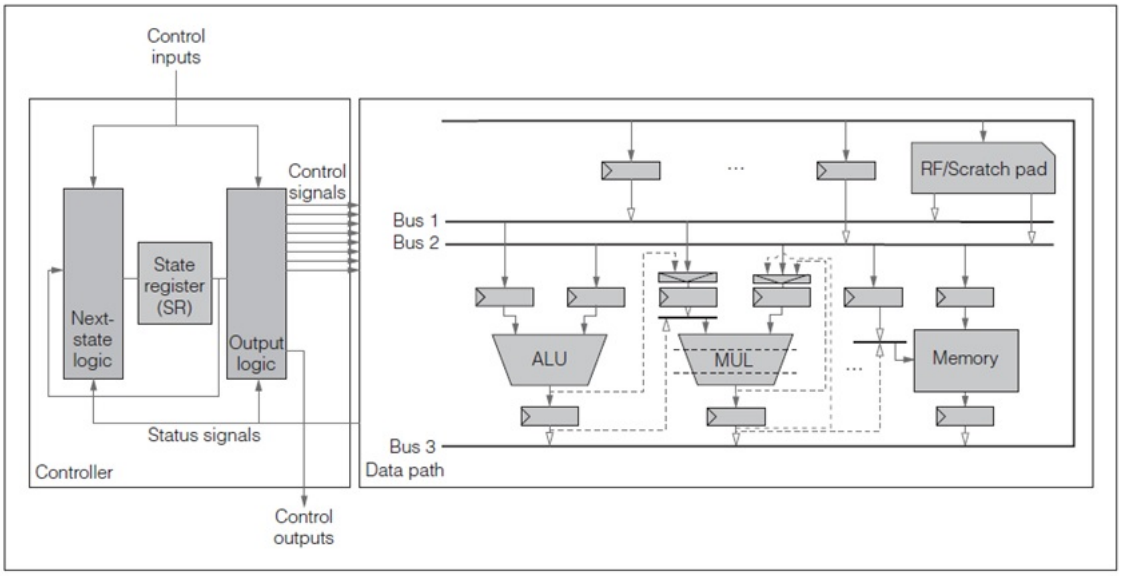

FIGURE 4.4: Typical structure of RTL, produced from a High Level Synthesis tool [35]

4.2 Vivado HLS – Tutorial

Vivado HLS tool from Xilinx, is a tool that produces RTL in three HDLs (Verilog, VHDL and System C) from a source input, described by C/C++ or System C. Except for C/C++ code that essentially describes the algorithm that we want to implement on hardware, directives and constraints are also inserted in order to help tool for outputting RTL close to user's specifications. These directives are inserted into a Tcl file or via GUI. Vivado always targets to a specific FPGA device which is given as constraint, so each RTL and therefore results, are closely related to device family and number. Different devices have different clock speeds and paths and also they have different hardware resources. Hence, HLS tool has to know about what and how many hardware resources are available and how fast ("fast" refers to signal delay) is the device, because different delays can be met and different hardware components may be allocated. Along with RTL, a report with synthesis results is also outputted, in order to realize how close is outcome to designer's specifications, in terms of area, power and performance. Report summarizes device utilization, min and max latency and interval and the clock delay was achieved. Interval, is the time that a new input can be inserted in a module and in pipeline implementations, it represents throughput of design, while latency gives the pipeline depth.

Typical input constraints can be the clock cycle in *ns*, the percentage of period's uncertainty, which is the portion of period that is going to be used afterwards in post-place and route results and finally the target device.

Directives now, can be large in number, because we have many choices on different types of directives and also they can be configured by different parameters. In Subsection 4.2.1, we present some of the most important directives that some of them are used in our experimental set up.

Vivado HLS, aims on creating hardware blocks in order to use them as separate IP blocks on FPGA hardware designs. For this reason when Vivado creates RTL, top-module and all sub-modules on different hierarchy level, have always some standard interface signals, in order to interact with other hardware modules. *Ap_start* signal, triggers top module, so to start performing its dedicated task. *Ap_idle* remains high, as long as the module doesn't perform any calculation and becomes low, when it starts any operation. This signal is used, just to know when our hardware block is elaborating a task. *Ap_done*, indicates when block finishes with its task, to know when output is valid for sampling. *Ap_return* essentially is the output of top module which of course can have more than one. *Ap_rst* is a standard reset signal, in order to set the circuit in a known state of FSM. *Ap_ready* indicates when a new input can be inserted in module. This is a very useful signal, especially in pipeline designs where a new input can be fed into, before *ap_done* is asserted. All previous signals are completely controlled from the FSM part (Control Unit) of RTL, which also undertakes the interaction between sub-modules. Fig. 4.5 shows a small piece of C code inserted into Vivado and all the input and output signals of top module, after high level synthesis process.

If final RTL result meet target application's specifications, then it can be easily extracted as an IP core from HLS tool and so opened as a self-contained design from Vivado Logic Synthesis tool. After that, design can follow next steps in logic synthesis (synthesis, map, place-route), till bitstream file will be created, thus transplanting our design onto the target FPGA device.

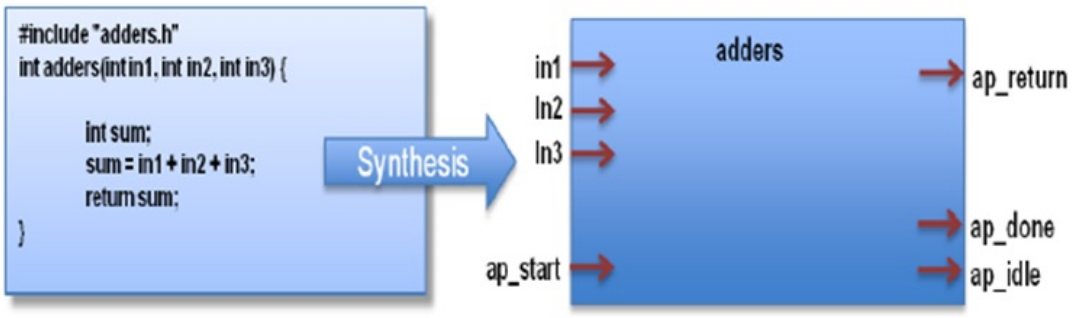

FIGURE 4.5: A small example of C code and what interface signals are produced at top module level after high level synthesis process [32]

One needs also to be mentioned in this section, the usage of DSP48E slices, which exist in some FPGA devices. DSP48E slices are essentially separate hardware modules on a FPGA and can be used from any design to perform usual DSP operation, thus saving LUTs reservation. The DSP48E slice supports many independent functions. These functions include multiply, multiply accumulate (MACC), multiply and add, three-input add, barrel shift, wide-bus multiplexing, magnitude comparator, bit-wise logic functions, pattern detect and wide counter. The architecture also supports cascading multiple DSP48E slices, to form wide math functions, DSP filters, and complex arithmetic, without using general FPGA logic. For instance, in our experiments DSP48E slices were used for multiply-accumulate operations (due to the origin of algorithm) and later on barrel shift operations when source code transformed from multiply to shift-add operations. For further details on DSP48E slices refer to [31].

4.2.1 Directives

Directives as mentioned earlier, are commands that are inserted in a HLS tool, so to help about what kind of RTL we want to output. We will present some of the most useful ones that utilized in our experimental set up, but are not limited here.

Except clock period, uncertainty and target device which are determined as input constraints , all the rest configurations are declared as directives. Reset style, FSM state encodings, interface signals, latency constraints for loops and functions, are some basic directives that designer may use to create RTL as close as possible to his preferences. In general, almost all directives can be summarized in three categories trying to optimize the created RTL: function, loop and array optimizations. Following Tables 4.1, 4.2 and 4.3, show some useful directives along with their description.

TABLE 4.1: Directives for function level optimizations

Directive	Description
Inline	Inlines a function, removing all function hierarchy. Helps latency and throughput by reducing function call overhead
Instantiate	Allows functions to be locally optimized.
Dataflow	Enables concurrency at the function level and used to improve throughput and latency
Pipeline	Improves throughput of the function by allowing the concurrent execution of operations within a function
Latency	Allows a minimum and maximum latency constraint to be specified on the function
Interface	Applies function level handshaking

TABLE 4.2: Directives for loop level optimizations

Directive	Description
Unrolling	Unroll for-loops to create multiple independent operations rather than a single collection of operations
Merging	Merge consecutive loops to reduce overall latency, increase sharing and optimization
Flattening	Allows nested loops to be collapsed into a single loop with improved latency
Dataflow	Allows sequential loops to operate concurrently
Pipelining	Used to increase throughput by performing concurrent operations
Dependence	Used to provide additional information which can be used to overcome loop-carry dependencies
Latency	Specify a cycle latency for the loop operation

In our experiment loop unrolling was used on two different levels of parallelism to realize how much hardware resources expand and what the gain in latency. We experimented

TABLE 4.3: Directives for array-storage level optimizations

Directive	Description
Resource	Specify which hardware resource (RAM component) an array maps to
Map	Reconfigures array dimensions by combining multiple smaller arrays into a single large array to help reduce RAM resources and area
Partition	Control how large arrays are partitioned into multiple smaller arrays to reduce RAM access bottleneck
Reshape	Can reshape an array from one with many elements to one with greater word-width
Stream	Specifies that an array should be implemented as a FIFO rather than RAM

also with pipeline solutions because our aim was the throughput performance, which is favored from dataflow RTLs.

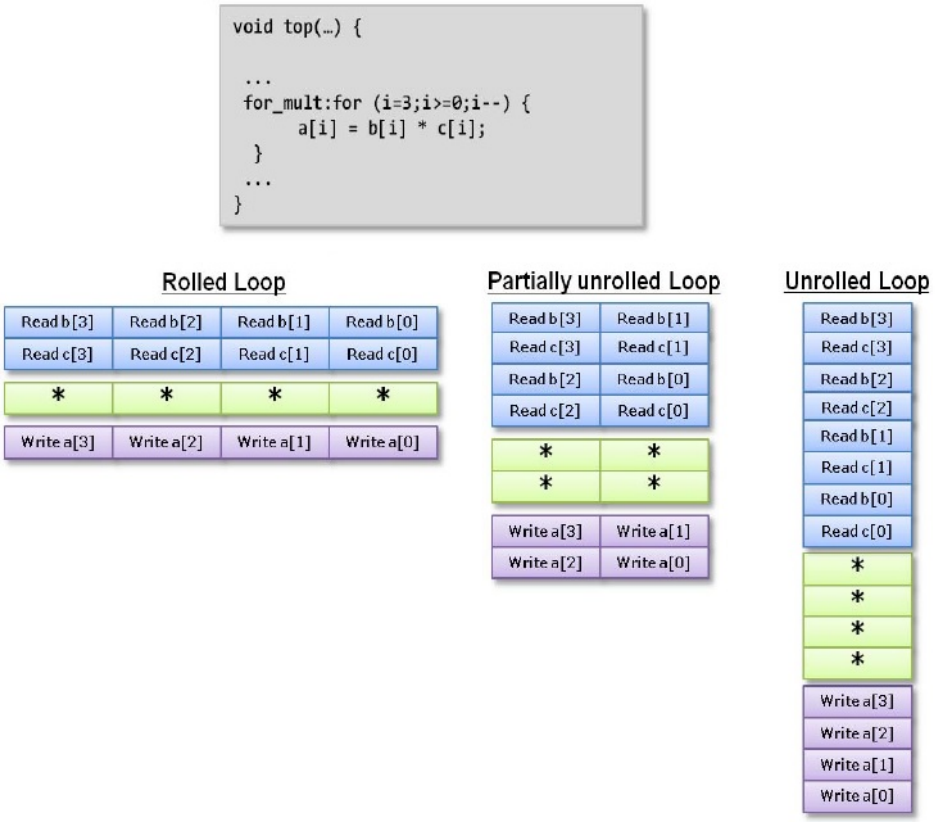

FIGURE 4.6: Partial and full loop unrolling example in a small loop. Latency is improved as level of unrolling increases [32]

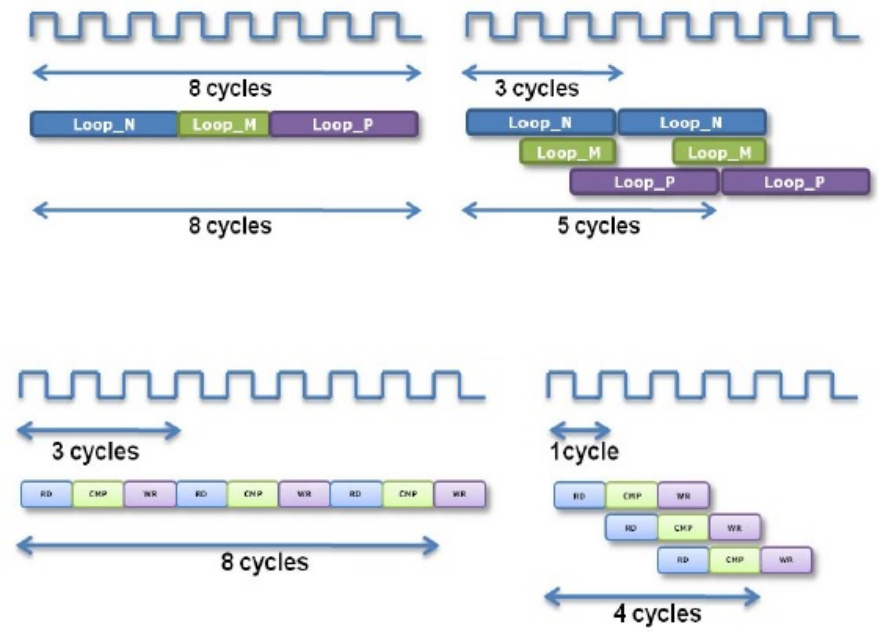

FIGURE 4.7: Pipelining betweern different operarations and loops examples. Interval and throughput are directly affected [32]

4.2.2 Latency-Based Control

In this subsection, we describe how HLS tool manipulates latency and interval, in pipeline and no-pipeline designs. We will realize how latency is computed, so to know in how many cycles different control paths may produce results. Inferences, are derived from experiments we conducted in small pieces of code, in order to deduce how Vivado HLS creates FSM, to control latency.

First, we analyze designs that are no-pipeline directed. If one circuit has only one control data path, a maximum latency is computed according to scheduling process and this is used to assert *ap_done* signal. If various different control paths are existed, HLS compute different latencies for every possible control path that may be used. FSM retains in its stages this information and according the value of control signals, the appropriate latency is used to assert *ap_done*. A MUX circuit is used to multiplex control signal's values and according its output, correct latency is chosen, according to which control path is going to be used. Figure 4.8, shows the following code example, if no-pipeline directive is given.

Regarding now pipeline designs, if one control path is inferred then a latency is computed witch implies the pipeline's depth. Also, an interval time is computed, indicating circuit's

throughput. At first, FSM counts maximum latency cycles in order to fill pipeline and after that every interval cycles it produces an output. If many control paths are existed, then it finds the maximum latency of all. Regardless which control path is used, at the first time it counts maximum latency to fill pipeline. After that, one input can be inserted every interval time, thus producing a result according to interval time as well. For operations with different latencies, tool align all latencies to worst by adding FFs until meet worst latency. The following code example illustrates what we explain here.

```
// TEST.cpp
int  EXAMPLE (short A, short B, short C)
{

   int tmp, res;

   if(C == 1){
      tmp = A + B; //latency 1 cycle
   }
   else{
      tmp = A*B;  //latency 3 cycles
   }

   if(C == 1){
      res = tmp / B; //latency 18 cycles
   }
   else{
      res = tmp - B; //latency 1 cycle
   }

   return res;

}
```

In the above code example, two possible control paths may be followed. In former case, a divider will be used after an adder, while in latter case a subtractor will be used after a multiplier. In quotes, we show every different latency for different operation. Having directed this code for pipeline design, all latencies in each sub-module align to the worst adding more FFs without any logic inside. So, latency of adder path will be 3 cycles and latency of subtractor 18 cycles. This happens because every time a module begins a new operation, maximum latency is counted in order to output the first result. Hence, in this example, we have to wait for 21 cycles before first outcome occurs, but after that

in every cycle we can give a new input and get a new output. Fig. 4.9 illustrates our previous analysis.

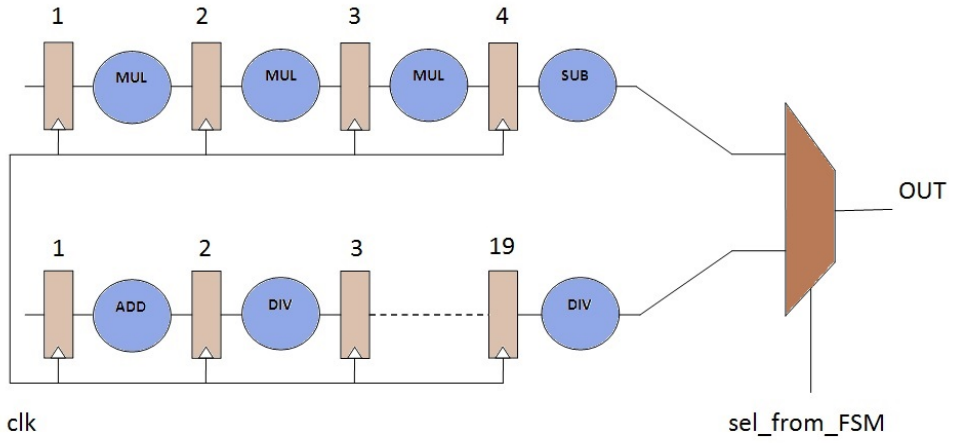

FIGURE 4.8: In unpipelined designs, different latency information is stored for different data paths. According to the input signals, FSM chooses each of them for output in specific latency cycles

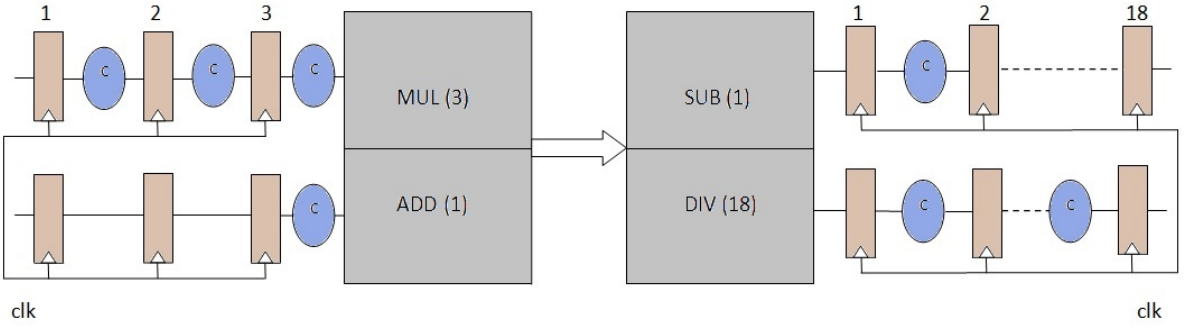

FIGURE 4.9: Two modules with different latencies are aligned in worst latency by adding FFs, when RTL is pipelined

Chapter 5

Experimental Methodology

This Chapter presents how we set up the experiment in order to obtain results and to compare with other different implementations. Vivado HLS tool that was utilized to explore design space of HEVC integer transform module, targets into FPGA mapping of RTL, hence, all hardware resources refer to FPGA hardware components.

The basic idea of the experimental set up, is to fulfil the aim of this work which is the design space exploration. So, we primarily experimented about how many different RTLs, HLS tool can yields for reference source code (refer to Subsection 5.1.1), so to capture their performance in terms of throughput. Observing output results from HLS tool for a certain source code, we can realize how tool reacts on different directives, input delays and several other characteristics that can be inferred from the HLS output. After that and having obtained this knowledge, we may know how HLS tool manipulates source codes and directives in order to output RTL, thus intuitively knowing how tool will react on future works with different algorithms and directives.

Having finished design exploration of reference source code, we tried to input source code, having replaced multiplications with shift-add operations. Shift-add operations are extensively used from custom RTLs, as a technique that replaces pre-defined multiplications with shift and add circuits, thus reducing area and critical paths. Thinking on the same way, we are trying to create RTLs that only use shifts and additions. Predefined shifts on hardware designs utilize only wires; shifts on FPGAs can be mapped using either LUTs or DPS48E modules. For that reason, we experimented on two different source codes. The former code, aims to map shifts on DSP48E slices (see Subsection 5.1.2) and the latter on LUTs (see Subsection 5.1.3).

5.1 General Flow

For all of the three source codes we experimented on, the same procedure was followed in order to implement, to verify and finally to take simulation results. The structure of Inverse Transform source code, is the same for the three different codes. A top function that performs the inverse transform, switches between the four different size transforms according to the size of current TU. Each of four functions that are called in order to perform a fast inverse transform, represent a sub-module of respective size transform. So, in each case–statement, a sub-module-function is called twice, one for the 1-D horizontal step and one for the 1-D vertical step (2-D transform). Vertical 1-D step takes as input the output of horizontal 1-D step as explained in HEVC Integer Transform chapter and yields a block of pixels after 2-D inverse transformation. Each such function, performs the fast inverse transform algorithm according to Chen's diagram [38].

Initially, each source code have to be incorporated into Vivado environment, so to compile it and verify that C++ code works properly, thus continuing in synthesis step. This is a very important step, because only if we have verified that algorithm works properly, we can proceed to synthesis; else the outputted RTL will have bugs into behavioural simulation. The verification of top module in Vivado as mentioned in previous Chapter 4, is carrying out using C-like testbenches, modelling top module for synthesis into a C function. Afterwards, one needs to verify module's functionality, give some known input data and observe output results. Therefore, the first thing we had to do, was to write a C-testbench in that way that it will be able to take a standard input, create output based on inverse integer transform and finally compare with a golden output that for sure has correct data.

Trying to create a model that will be self-checked, we had to fed C-testbench with some known input, so to compare the output. The HM-15.0 reference source code ported on Visual Studio environment and into the code segment concerning inverse integer transform, we added an extra piece of code that writes on a file input data of inverse transform (coefficients in TUs) and on another file it writes the output of transform. Input data are declared as standard input and output data as golden output that it must matches with the testbench one's. A reference video bitstream was used in HEVC decoder from JCT-VC database [45], so to obtain input and output data. After that, C-testbench reads data (to wit TUs) from input file, performs the inverse transform algorithm and retains its results in a buffer. Finally, it reads golden output data from file and eventually compares results with the golden output, printing an error in mismatch case. Having created this testbench and having also golden input and output files for validation, we can try every change we want from now on at source code, because we are able to check rightness, thus having correct behavioural RTL after synthesis step.

Having checked now source code validity, we can proceed to synthesis step. Except for source code, also directives must be inserted into tool in order to check out how tool induces. In each of three source codes we have given same directives. Each different directive leads to different RTL for the same source code; Vivado HLS indicates different RTLs for the same source as "solution N", so we are going to use similar terminology. For this work "Configuration M.N" indicates M source codes with N solutions each, so both declarations will be used at next chapters. Four solutions tried for each source code. First solution contains no directives. Second solution is directed to partially unroll all for loops by a factor of two, in respect to the higher number of iterations. Solution 3, is directed to fully unroll any loop that exists in source code. Finally, solution 4 is directed for a pipeline design in order to realize how throughput changes with pipeline implementations. Then, we have four solutions for each source code, to wit twelve different configurations and RTLs that each of them was tried on different input clock period constraints, to see what changes tool performs on RTL, trying to meet different critical paths and delays.

As explained in the previous Chapter 4, every synthesis that is performed from source code and directives, aims on a certain technology because high level synthesis produces RTL target to some device for optimized result. Also, the report from HLS shows the utilization of a specific device. The FPGA device we inserted as target device in HLS tool throughout the experiment, is from qkintex7 family and the device's code is xq7k410trf900-2l. Regarding the hardware resources this device has 1590 DSP48E slices, 508400 number of FFs, 254200 LUTs and 1590 BRAMs.

5.1.1 Reference Source

Reference software is the pure source code of inverse integer algorithm, as extracted from HM-15.0 reference software [45]. Small changes were carried out in some pointer variables, because HLS tool have to know what is the exact size that a variable is mapped on, so to create a buffer in hardware with the same size.

All four sub-modules with fast inverse transforms, use multiplications and additions in order to calculate the result. If target device has DSP48E modules, then all the multiplications and additions are by default forced to be mapped there, for better efficiency in those more specialized modules. Several for loops are used in functions, so to perform tasks that can be accomplished iteratively. The bigger size of transform the more for-loops we have, having also higher number of iterations. Observing whole dataflow diagram according to Chen's algorithm, we may understand how source code works and

how the complexity of each function–sub-module scales when compared to the other three.

5.1.2 Inline Shift–Add Source

Custom RTL designs for integer transform algorithm and general for convolution-like operations with defined multiplicands, usually utilize shift-add operations, because multiplication modules increase area cost, critical paths and latency. Trying to invoke HLS tool to deploy RTL using as much less multiplications, we changed source code, so to don't use them any more. They replaced from left shift operations in C level. For example, if we have to multiply a sample with three, one needs, only make a left shift by 1 on sample and add it once ($sample * 3 = (sample << 1) + sample$). In hardware, pre-defined amounts of shifts, are carried out only by exchanging wires without shift-registers. Besides, arrays with multiplicands of integer DCT, doesn't need any more because there aren't any to be used now. With this change, we expect that all multiplications will be replaced by shift-add operations, thus saving a lot of hardware resources and giving chances for smaller latency.

Observing results, we will see that DSP48E module's utilization is considerably decreased which is a vital factor for enabling a map onto target FPGA device. Although DSP48E utilization decreased, we would expect that DSP48E modules will not be used at all, because there is no multiplication operator in source code to invoke such a map. In next chapter, we are discussing why results show that DSP48E modules are still used.

5.1.3 Function Shift–Add Source

The third source code we tried, was created in order to eliminate DSP48E modules completely. The modification here is based on the source code with shift-add operations 5.1.2. The problem we tried to solve is the mapping of shift-add operations on DSP48E modules. So, we created separate function–modules, each of them takes as input a sample and makes a left shift by some defined quantity, according to which function is used. Ultimately, we change the hierarchy level for those functions that perform shift, expecting that tool will map all such functions in LUTs, in order to observe output results and to see how device utilization and latency get affected.

Indeed, results show that this modification completely eliminates DSP48E modules usage and LUTs number are increased because all shifts are mapped there now. Finally, as we are going to see in Chapter 6, this version of source code is the most efficient in terms of

device utilization (much less hardware resources are now used) and in some sub-modules latency is also decreased.

Chapter 6

Results

In this chapter we present and analyze the results obtained from the Vivado HLS tool. They illustrate the performance of the inverse integer transform hardware implementation, either in terms of an area–latency–delay trade off, or in throughput requirements for supporting a real time video decoding application. Results are illustrated for all the different configurations that Vivado HLS tool derived for the three different C++ sources and according to the inserted directives. By examining the raw results from tool's report, we can realize how the tool reacts on different directives and different C++ code in styles, describing the same algorithm. The sections of this chapter are organized as follows: Section 6.1, provides output results from HLS tool, giving information about different implementations on a target FPGA device. The results from the three different source codes we experimented on, are presented in Subsection 6.1.1 for the H.265 relevant reference code, in Subsection 6.1.2 for the inline shift–add version and in Subsection 6.1.3 for the function based shift-add version. Section 6.2, illustrates results in 2-D and 3-D figures. Final Section 6.3 provides throughput results for all the different implementations, comparing them in terms of performance. It is also useful, to identify when each module becomes a critical component into a hardware decoder, for different video resolutions and frame rates. An overview of the top module block diagram that HLS tool, yielded for the different RTLs is illustrated in Fig. 6.1. All the different architecture optimizations, are performed within each of the four sub-modules, without changing the architecture at top level's RTL.

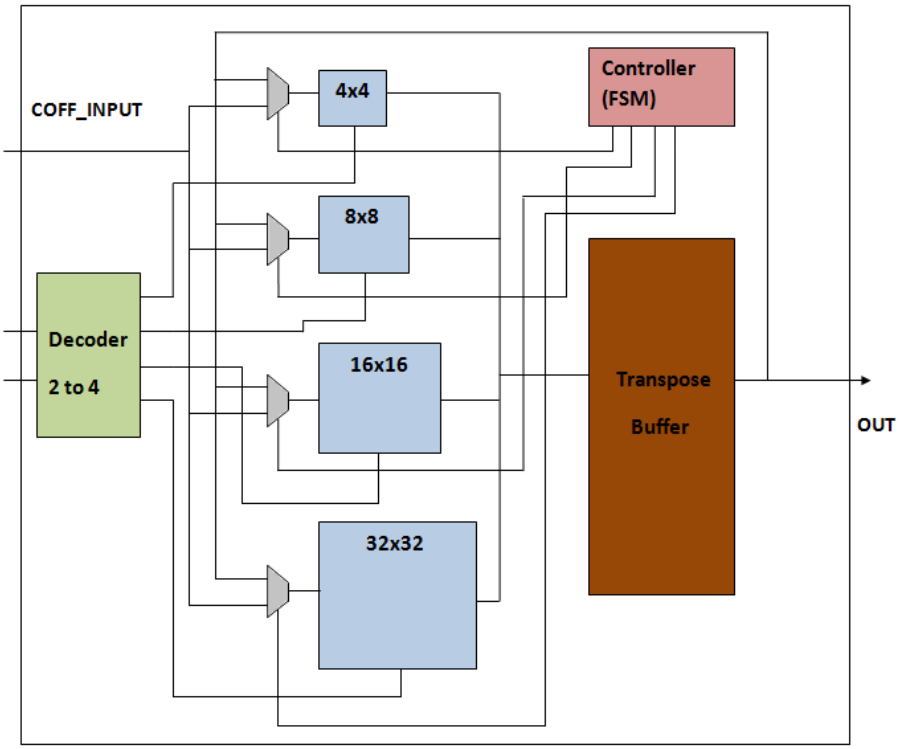

FIGURE 6.1: Block diagram at the top module hierarchy level that Vivado HLS tool yielded for all the different RTLs.

6.1 Vivado HLS Results

In following tables of this section, sub-module latency refers to the standard latency, required to accomplish the 1-D stage of each size transform. Latency at the top module level, refers to the maximum latency that is occurred from $32x32$ sub-module transform, which yields the worst latency among all other sub-modules. Minimum latency on top-module is zero, which occurs in case of error –if size of requested transform is invalid– because top module immediately terminates in this case. Other latencies for sub-modules $4x4$, $8x8$ and $16x16$, stand between minimum and maximum values. Because these four transform sub-modules are mutually exclusive, for each control path, the control FSM retains latency information and according to the selected path, different latency is used. Therefore, for the four transforms and the error case, five different latency information are stored in FSM, so to implement the control interface. Table 6.1, explains each different configuration that yields different RTL architecture. This terminology is going to be used throughout this chapter.

TABLE 6.1: All different configurations that experiments were conducted

Configuration	Source Code	Directives
1.1	Reference	No Directives inserted except for target period and device
1.2	Reference	Directives for partial unroll in all loops, by a factor of 2
1.3	Reference	Directives for fully unroll in all loops
1.4	Reference	Directives for pipeline design in all sub-modules
2.1	Inline Shift-Add	No Directives inserted except for target period and device
2.2	Inline Shift-Add	Directives for partial unroll in all loops, by a factor of 2
2.3	Inline Shift-Add	Directives for fully unroll in all loops
2.4	Inline Shift-Add	Directives for pipeline design in all sub-modules
3.1	Function Shift-Add	No Directives inserted except for target period and device
3.2	Function Shift-Add	Directives for partial unroll in all loops, by a factor of 2
3.3	Function Shift-Add	Directives for fully unroll in all loops
3.4	Function Shift-Add	Directives for pipeline design in all sub-modules

6.1.1 Reference Code

The reference software of inverse integer transform, includes only multiplications and additions as arithmetic operations in order to create results, for each stage of algorithm. Hence, DSP48E modules are used extensively because as already mentioned in Chapter 4, Vivado HLS tool maps arithmetic operations on DSP48Es modules wherever feasible. Therefore, it fuses multiplications with additions into a single arithmetic module (multi-cycle module) that exists on some devices for such arithmetic operations.

Configuration 1.1 is the most optimal implementation, in terms of device utilization (aka occupancy), because everything are performed in a serial fashion without exploiting any parallelism. In Table 6.2, we can see the same circuit over five different target periods and how tool trades area (FF and LUTs) for latency and delay.

Configuration 1.2 now, is directed to partial unroll all for loops by a factor of two, thus making a parallelism for identify a better latency result. However, unrolling loops requires more hardware resources, as some operations operate in parallel. Thus, as we can observe in Table 6.3, the number of FF, LUTs and of course DSP48E modules, increase significantly due to hardware expansion. The most loops and operations has a module, the more hardware resources are allocated further for it.

Configuration 1.3, we directed the tool in fully unroll any loop, trying to reach the minimum latency, while expecting area to be maximized. Indeed, this great level of parallelism, implies that latency falls significantly and area grows up extensively, as shown in Table 6.4. Area utilization in this configuration is very large, thus requiring high capable FPGAs, in order to map the circuit. Completely unrolling on large loops

TABLE 6.2: Configuration 1.1 – HLS Report for different delay constraints on the same configuration

Solution	**Period(ns)**	**Latency**	**Interval**	**BRAM**	**DSP48E**	**FF**	**LUT**
Top Module	20.67	9413	9414	1	52	3649	7258
4x4	17.82	17	17	0	2	161	1209
8x8	14.3	185	185	0	6	518	1453
16x16	17.97	1089	1089	0	14	1114	1952
32x32	20.67	4705	4705	0	30	1838	2244
Top Module	15.58	10949	10950	1	52	3770	7294
4x4	15.48	17	17	0	2	175	1209
8x8	14.3	185	185	0	6	518	1453
16x16	15.58	1217	1217	0	14	1141	1952
32x32	15.58	5473	5473	0	30	1918	2280
Top Module	10.49	13765	13766	1	52	4650	7305
4x4	10.49	21	21	0	2	290	1209
8x8	10.49	249	249	0	6	600	1455
16x16	10.18	1537	1537	0	14	1362	1955
32x32	10.49	6881	6881	0	30	2380	2286
Top Module	5.17	21829	21830	1	52	6546	7350
4x4	5.09	33	33	0	2	451	1212
8x8	5.17	361	361	0	6	877	1464
16x16	5.17	2257	2257	0	14	1829	1964
32x32	5.17	10913	10913	0	30	3371	2310
Top Module	2.39	39309	39310	1	52	8877	7674
4x4	2.39	63	63	0	2	964	1223
8x8	2.39	747	747	0	6	1333	1541
16x16	2.39	4115	4115	0	14	2484	2068
32x32	2.39	19651	19651	0	30	4063	2434

–as some of them exist in 16x16 and 32x32 transforms– leads to huge device utilization and impractical solutions, since they exceeds the capability of the largest FPGAs.

Architectural plan for configuration 1.4, is to pipeline the circuit of IIT, in order to achieve max throughput. Pipelining, introduced only in the four sub-modules, that exist in top-module. As mentioned in Chapter 4, the Vivado tool when pipelining a design, fully unrolls all loops, in order to create higher parallelism, which again leads to significant device utilization. Additionally, it utilizes more FF and LUTs –as we expected to do so– for creating pipeline stages and retain intermediate results (see Table 6.5). However, the main difference between this pipeline version comparing with previous configurations, time interval is considerably reduced that enables each module to take faster new inputs without having completed previous operations. Hence, each sub-module's throughput is significantly increased, affecting the entire performance of the top module.

TABLE 6.3: Configuration 1.2 – HLS Report

Solution	**Period(ns)**	**Latency**	**Interval**	**BRAM**	**DSP48E**	**FF**	**LUT**
Top Module	20.67	6665	6666	1	2908	31583	100559
4x4	17.82	13	13	0	4	198	2238
8x8	13.59	147	147	0	40	1983	7455
16x16	17.97	915	915	0	144	6727	22619
32x32	20.67	3331	3331	0	2720	22657	67847
Top Module	2.39	25041	25042	1	2908	164763	106611
4x4	2.39	37	37	0	4	1786	2254
8x8	2.39	437	437	0	40	7282	7975
16x16	2.39	3877	3877	0	144	23809	24608
32x32	2.39	12517	12517	0	2720	131853	71366

TABLE 6.4: Configuration 1.3 – HLS Report

Solution	**Period(ns)**	**Latency**	**Interval**	**BRAM**	**DSP48E**	**FF**	**LUT**
Top Module	21.87	1313	1314	1	10104	77434	386039
4x4	17.82	9	9	0	8	234	4246
8x8	21.87	61	61	0	112	2886	13444
16x16	21.87	230	230	0	992	10736	64034
32x32	21.87	655	655	0	8992	63560	303860
Top Module	2.39	2733	2734	1	10104	148084	384845
4x4	2.39	20	20	0	8	457	4248
8x8	2.39	135	135	0	112	4359	13214
16x16	2.39	428	428	0	992	18471	63282
32x32	2.39	1363	1363	0	8992	124779	303646

TABLE 6.5: Configuration 1.4 – HLS Report

Solution	**Period(ns)**	**Latency**	**Interval**	**BRAM**	**DSP48E**	**FF**	**LUT**
Top Module	20.96	1056	1057	1	10104	261020	386947
4x4	17.82	9	8	0	8	411	4248
8x8	15.22	53	32	0	112	5885	13068
16x16	18.26	135	128	0	992	32573	62776
32x32	20.96	527	512	0	8992	220686	301160
Top Module	2.39	2018	2019	1	10104	718501	536280
4x4	2.39	20	8	0	8	3411	4296
8x8	2.39	65	32	0	114	14359	14244
16x16	2.39	252	128	0	992	92263	79000
32x32	2.39	1006	512	0	8992	605753	427672

6.1.2 Inline Shift–Add Code

This subsection contains all the results, regarding the modified code, without invoke any multiply operation into source. All multiplications have been transformed into shift–add operations, a transformation exploited in various hardware designs, so as to reduce area and critical path or the latency.

Vivado HLS, is directed by default in mapping various usual arithmetic operations on DSP48E slices, for devices that have this option. We would expect that changing the code without any symbol for multiply operation, no DSP48E module would be utilized. However, DSP48Es are still utilized, despite that their number is considerably decreased. So, an obvious profit we have from this approach, DSP48E module's utilization is attenuated significantly and now our circuit can be mapped on a device requiring less DSP48E resources. Besides, reducing DSP48E modules, it is rather straightforward to see that LUT utilization is increased, because some arithmetic operations now, performed from LUTs.

Obtaining the output report from HLS, even though no multipliers are used in code, DSP48E modules are still used, to manipulate the various amounts of shifts and adds. We do know, that shift operations in ASIC hardware, cost only wires for fixed- length of shift operations. So, for circuits on FPGA, shifts cost in LUT utilization. Vivado HLS, for a certain number of different shifts decides to perform them on DSP48E modules and to combine with adders, thus saving LUTs and utilizing properly the existing device's hardware resources.

This mapping on DSP48E slices, yields a small increase in latency due to the greater number of FFs that utilizes, in order to control the DSP48E modules, and to perform shift-add operations. This increase in FFs becomes greater to lower target periods. This serial approach, with all the shifts going to DSP48E slices, produces higher latency to design, as a few modules have to perform many shift and add operations, which are FFs controlled. As we can see in Tables 6.6, 6.7, 6.4 and 6.9, having changed the input source code, despite latency has been increased, it drops down in bigger modules such as $16x16$ and $32x32$, in comparison with reference code implementation. Hence, using the RTL from the new source code, throughput of specific modules is increased, as long as their latency diminish.

TABLE 6.6: Configuration 2.1 – HLS Report

Solution	**Period(ns)**	**Latency**	**Interval**	**BRAM**	**DSP48E**	**FF**	**LUT**
Top Module	21.39	9413	9414	1	12	7582	19698
4x4	18.48	17	17	0	1	161	1324
8x8	19.28	113	113	0	1	416	2132
16x16	21.39	673	673	0	2	1167	5669
32x32	17.68	4705	4705	0	8	5838	10573
Top Module	2.39	47053	47053	1	12	66380	21714
4x4	2.39	79	79	0	1	2298	1360
8x8	2.39	667	667	0	1	5738	2387
16x16	2.39	3171	3171	0	2	18921	6511
32x32	2.39	15682	15682	0	4	39423	11456

TABLE 6.7: Configuration 2.2 – HLS Report

Solution	**Period(ns)**	**Latency**	**Interval**	**BRAM**	**DSP48E**	**FF**	**LUT**
Top Module	21.39	5769	5770	1	50	22786	157455
4x4	18.48	17	17	0	1	161	1324
8x8	19.28	113	113	0	1	461	2553
16x16	21.39	515	515	0	16	6617	49674
32x32	20.47	2883	2883	0	32	15529	103480
Top Module	2.39	21457	21458	1	50	524301	184476
4x4	2.39	79	79	0	1	2298	1360
8x8	2.39	443	443	0	1	6542	2849
16x16	2.39	2797	2797	0	16	164900	57309
32x32	2.39	7150	7150	0	32	350528	122526

TABLE 6.8: Configuration 2.3 – HLS Report

Solution	**Period(ns)**	**Latency**	**Interval**	**BRAM**	**DSP48E**	**FF**	**LUT**
Top Module	21.51	1061	1062	1	162	49510	356772
4x4	18.58	17	17	0	1	161	1324
8x8	19.28	65	65	0	1	419	3237
16x16	21.57	137	137	0	32	15930	111235
32x32	21.31	529	529	0	128	32982	240497
Top Module	2.39	3097	3098	1	162	1275214	253537
4x4	2.39	79	79	0	1	2298	1360
8x8	2.39	251	251	0	1	7596	3523
16x16	2.39	360	360	0	32	366376	127985
32x32	2.39	1545	1545	0	128	898944	120669

TABLE 6.9: Configuration 2.4 – HLS Report

Solution	Period(ns)	Latency	Interval	BRAM	DSP48E	FF	LUT
Top Module	21.51	1058	1059	1	162	89520	394576
4x4	17.39	9	8	0	4	471	4705
8x8	20.22	56	32	0	8	3350	19205
16x16	21.51	240	128	0	32	26110	109969
32x32	20.38	528	512	0	128	59589	260697
Top Module	2.39	2026	2027	1	162	1380302	618716
4x4	2.39	24	8	0	4	8636	4813
8x8	2.39	68	32	0	8	55205	20909
16x16	2.39	256	128	0	32	359532	123089
32x32	2.39	1010	512	0	128	956938	458705

6.1.3 Function Shift–Add Code

Our final modification to the source code, was based on the approach of the previous Subsection 6.1.2. If we recall previous results, we can see that though the usage of DSP48E modules is decreased, they are still mapped on FPGA device, thus conveying a higher latency especially to modules $4x4$ and $8x8$. Here we have modified the source code, trying to completely eliminate the usage of DSP48E modules and force tool, to make all shift and add operation using LUTs, instead of DSP48E slices. We try this, so as to find how latency is affected from different mapping on hardware resources, concerning shift and add operations.

Elaborating towards this direction, we created small functions, each of them performs a predetermined amount of left shift. Essentially, with this approach, we change the hierarchy level of shift functions as separate modules. In doing so, we evoke tool to map all shift and add operations in LUTs, without using any DSP48E module.

Results in Tables 6.10, 6.11, 6.12 and 6.13, depict that our approach (moving on different hierarchy layer the shift functions) worked, as we expected to do so. DSP48E slices utilization is removed, thus saving a lot of valuable hardware resources. Now, we leave more space to other video modules, to utilize DSP48E slices. Even more, another great benefit from this approach, the elimination of DSP48E slices, has decreased the number of FFs, thus reducing latency and eventually improving throughput performance, as we shall see in Section 6.3.

Comparing our latest effort, about the modification of source code, trying to optimize further the outcome RTL, we realize that the two great benefits are: the large diminish

Table 6.10: Configuration 3.1 – HLS Report

Solution	Period(ns)	Latency	Interval	BRAM	DSP48E	FF	LUT
Top Module	21.78	9413	9414	1	0	3491	12332
4x4	16.63	17	17	0	0	161	1361
8x8	15.28	113	113	0	0	416	2265
16x16	21.41	673	673	0	0	1069	6661
32x32	20.67	4705	4705	0	0	1838	2244
Top Module	2.39	39309	39309	1	0	15571	14274
4x4	2.39	63	63	0	0	1117	1421
8x8	2.39	435	435	0	0	2449	2510
16x16	2.39	2131	2131	0	0	7909	7501
32x32	2.39	9825	9825	0	0	4063	2434

Table 6.11: Configuration 3.2 – HLS Report

Solution	Period(ns)	Latency	Interval	BRAM	DSP48E	FF	LUT
Top Module	21.41	5769	5770	1	0	30299	164565
4x4	16.63	17	17	0	0	161	1361
8x8	15.25	113	113	0	0	461	2685
16x16	21.41	515	515	0	0	6130	57834
32x32	20.67	2883	2883	0	0	23529	102285
Top Module	2.39	17825	17826	1	0	230371	179374
4x4	2.39	63	63	0	0	1117	1421
8x8	2.39	291	291	0	0	2769	2965
16x16	2.39	2101	2101	0	0	66655	64863
32x32	2.39	6258	6258	0	0	159797	109717

Table 6.12: Configuration 3.3 – Solution 3 HLS Report

Solution	Period(ns)	Latency	Interval	BRAM	DSP48E	FF	LUT
Top Module	21.87	1313	1314	1	0	79850	436993
4x4	16.67	17	17	0	0	161	1361
8x8	17.42	65	65	0	0	373	3363
16x16	21.87	230	230	0	0	15738	127954
32x32	21.87	655	655	0	0	63560	303860
Top Module	2.39	2733	2734	1	0	696677	478978
4x4	2.39	63	63	0	0	1117	1421
8x8	2.39	171	171	0	0	3106	3624
16x16	2.39	399	399	0	0	141023	139744
32x32	2.39	851	851	0	0	551398	333726

of device utilization regarding DSP48E slices and the lower latency that achieved as well.

TABLE 6.13: Configuration 3.4 – HLS Report

Solution	Period(ns)	Latency	Interval	BRAM	DSP48E	FF	LUT
Top Module	20.96	1056	1057	1	0	250443	459496
4x4	16.63	9	8	0	0	319	4856
8x8	18.95	46	32	0	0	3112	20436
16x16	20.56	225	128	0	0	24695	126744
32x32	20.96	527	512	0	0	220686	301160
Top Module	2.39	2018	2019	1	0	774626	604844
4x4	2.39	20	8	0	0	3991	5072
8x8	2.39	61	32	0	0	20134	22260
16x16	2.39	250	128	0	0	142025	138808
32x32	2.39	1006	512	0	0	605753	427672

6.2 Area – Delay – Latency

In this section, we present previous results, in terms of an area-delay-latency trade off. The 2-D diagrams that presented in Subsection 6.2.1, provide results about how HLS tool trades area and latency for delay, in order to realize how it performs in target period changes. 3-D diagrams, show that area and latency change alongside, as period changes.

In general, in this subsection we shall make a quick discussion, based on the results we obtained later, about how tool reacts on different inputs of period, creating different RTLs. Thus, we will try to understand how tool is designed and decipher its behaviour about input delay changes.

6.2.1 2-D Diagrams

It is a well known rule, that every EDA tool which creates a netlist either from a high-level language or a HDL, is based on a pareto curve that makes a trade off between area and delay. According to results that are presented in this subsection, Vivado HLS tool is based on such a curve as well. Area utilization in FGPAs, is declared as what portion of existing resources is reserved from the current device, for a specific RTL. In order to normalize utilization from different data (DSP48Es, FF, LUTs, BRAMs), we calculated the final normalized utilization which is depicted on following figures as the average, of all secondary utilizations. In doing so, we can compare different RTLs from different configurations, only using the averaged utilization, which depicts the comparison between them. This number is quite close to real device's utilization and represents a median value that device is utilized. This number gives a more general view of device utilization, but it doesn't depict accurately if a RTL design can be mapped onto a device. Every secondary utilization has to be taken into account, in order to realize if a design doesn't exceed the available hardware resources.

Figures 6.2, 6.4 and 6.6, show the area-latency trade off, for all the configurations we experimented on. Essentially, in each figure we can see the design space for each different source code, in terms of area-delay trade off. Observing carefully these figures or tables from previous section, we can realize that reducing target period, tool introduces more FFs, in order to reduce the delay of critical paths and divide them in smaller pieces. As the number of FFs increases, –for the reason that already described– as we can see the number of LUTs increases as well, because FFs in fact correspond directly to structures created from LUTs. Considering all the tables from previous Section 6.1, we may safely support the argument that the more hardware resources are allocated, the more rapid is the rate on which FFs and LUTs increase, while moving on shorter target periods. This means that in configurations x.2 and x.3 where hardware resources are huge, due to the loop unrolling directive, the rate that number of FFs and LUTs is growing up is increased, when compared to solution 1 for example that have allocated lower resources. This happens because fewer FFs and LUTs are required to get increased while lessening input delay constraint in solution 1. Hence, we have to pay attention on the delay constraint that we will utilize on the design, because in circuits with high amount of hardware modules and resources, device utilization (FFs and LUTs) increase largely and may exceeds the area capabilities of the target device.

With respect to the latency-delay trade off, we can see how it directly connects to the number of FFs. When tool is directed to achieve shorter period, it actually tries as already mentioned, to cut critical paths in smaller ones, thus creating smaller delay between FFs. The more pieces we divide the logic, the more latency increases, as circuit

requires more cycles in order to complete the task. Although, this in not happens when we are considering latency for pipeline designs, because by the time the pipeline get filled, after that, in each cycle we can create a new result if it is possible. Configurations x.1, x.2 and x.3, are not directed to give a pipeline RTL, so they have to wait until one TU completely get finished, before take another input. In this particular case, designer has to take into account that reducing the clock period, except for device utilization, latency is affected as well, in a trade off scenario that may reduce throughput performance. In next Section 6.3, we shall see in further detail issues about throughput and we will dive deeper in its analysis. By the time, it is straightforward to see that if latency increases sub-linearly with delay, then throughput grows up, while in the opposite situation throughput is getting worse. So, we are expecting to find better throughput results, from RTLs with the lowest time delay.

Concluding this subsection, each designer has to carefully consider following diagrams, in order to have a knowledge about how tool take decisions on area and latency, when the input target period changes and even more, how throughput maybe get affected from this. So, before we insert directives in HLS tool, we have to know about the diagrams on which it is based on and the decisions about the area and latency that is going to be used, in order to meet a specific delay constraint.

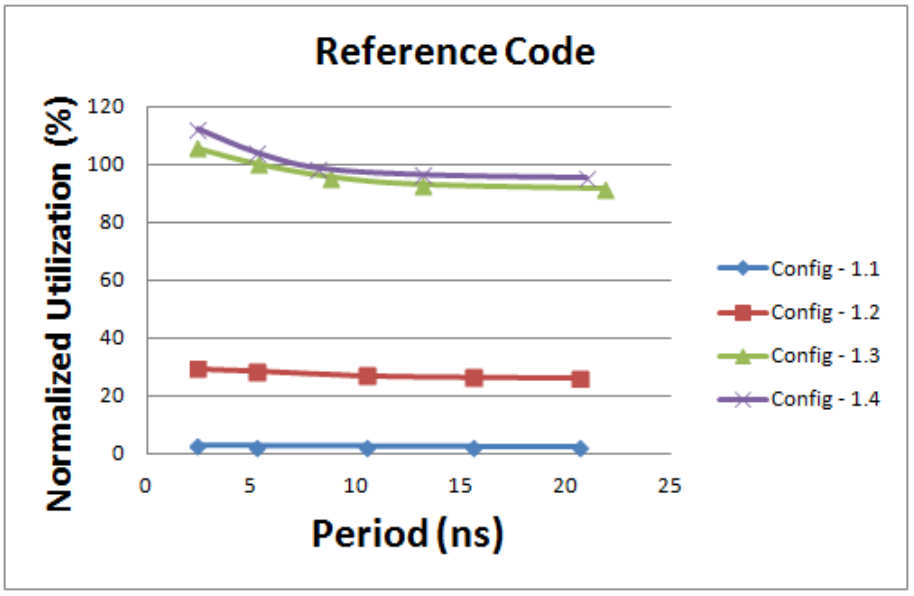

FIGURE 6.2: Normalized Utilization – Delay diagram for reference code experiment

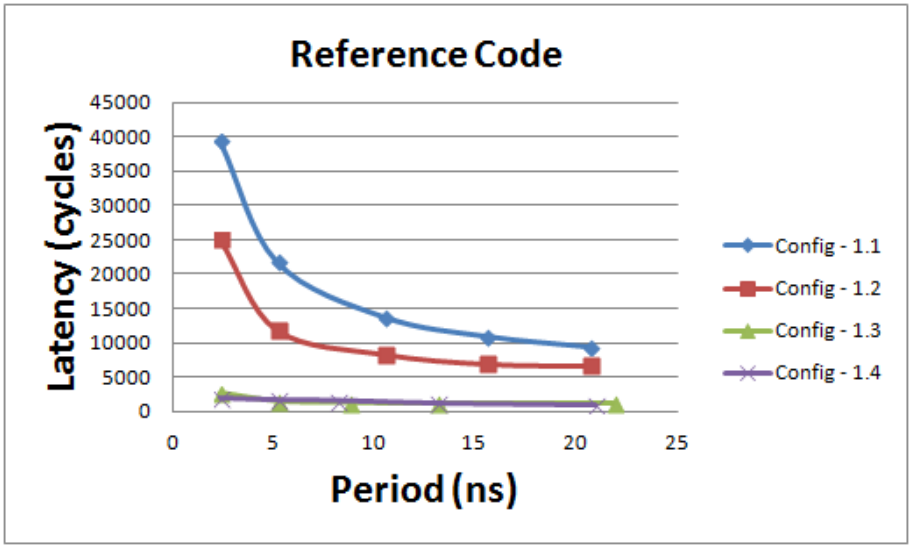

FIGURE 6.3: Latency – Delay diagram for reference code experiment

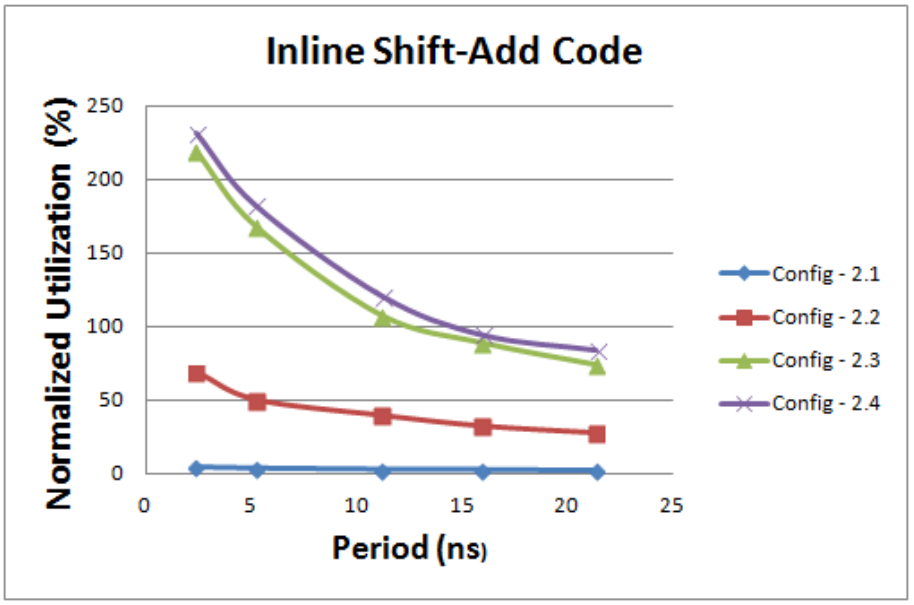

FIGURE 6.4: Normalized Utilization – Delay diagram for inline shift-add code experiment

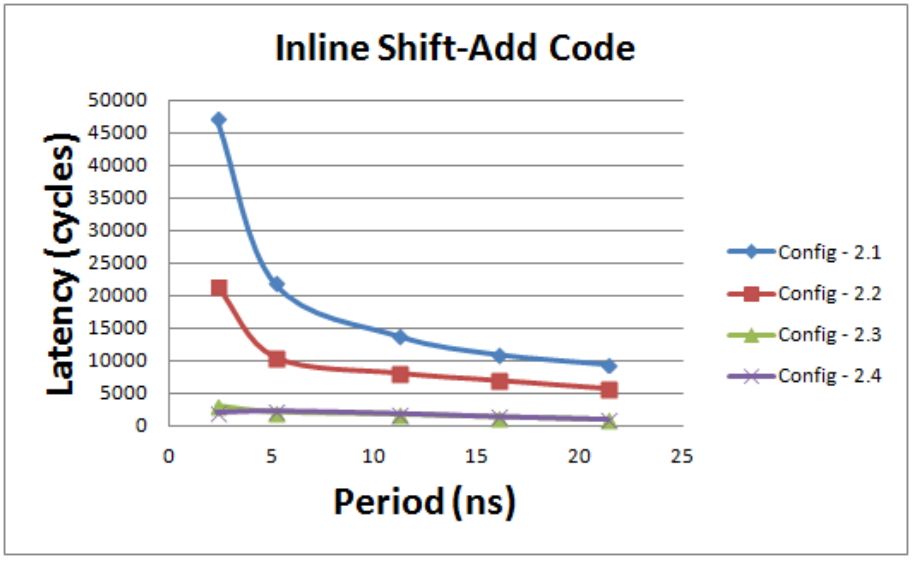

FIGURE 6.5: Latency – Delay diagram for inline shift-add code experiment

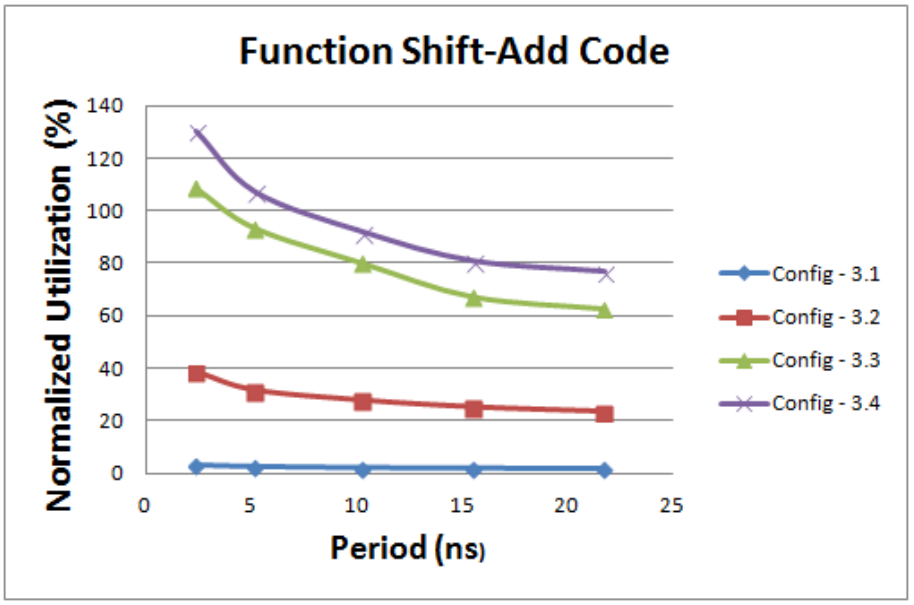

FIGURE 6.6: Normalized Utilization – Delay diagram for function based shift-add code experiment

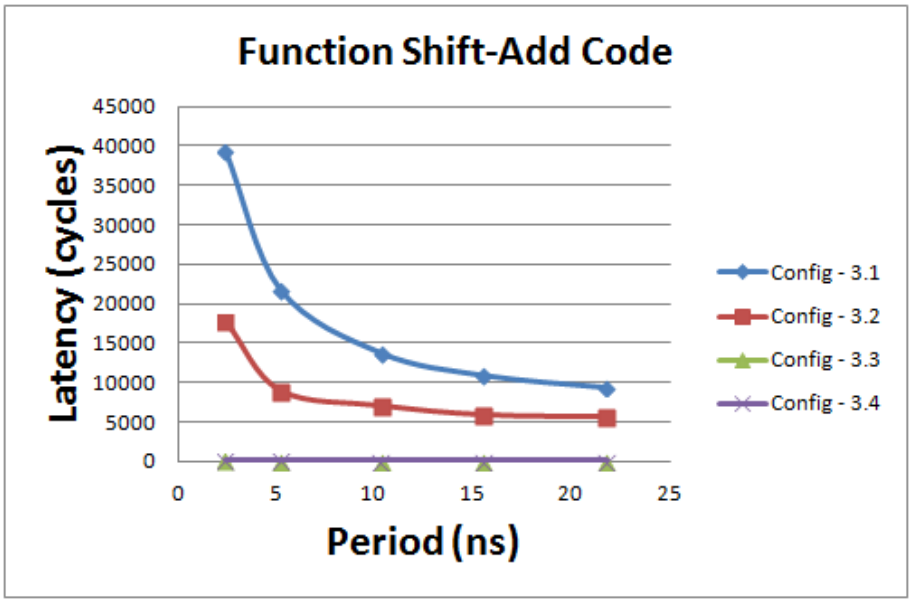

FIGURE 6.7: Latency – Delay diagram for function based shift-add code experiment

6.2.2 3-D Diagrams

Just in order to have a more complete picture of Pareto curves that we discussed on them earlier, we show here the 3-D diagrams with area and latency for delay trade-off.

The close relationship of area and latency when we are changing the target delay for our design, is depicted in following 3-D diagrams. In this subsection, area-utilization is presented as the sum of all secondary percentages –not the averaged as in previous subsection– only for visual reasons at the interpolation in 3-D plots. So, the averaged utilization can be obtained by dividing area's axis by three.

3-D diagrams in Figures 6.8, 6.9 and 6.10, show Pareto surface for some notional configurations. They depict that when design is evoked to work on higher operating frequency, to wit shorter period, number of FFs and LUTs are increasing in order to create shorter critical paths with fewer logic levels. Hence, this increases design's latency, thus negatively affecting throughput for unpipelined circuits. Each of five dots in each diagrams, indicate a specific RTL that exported from tool, with different target delay. As we moving on lower delays, dots move higher (latency increases) and have a direction at right (device utilization increases as well). So, following diagrams show up the cost in device utilization and latency that we pay, forcing our algorithm, to be implemented in higher frequency.

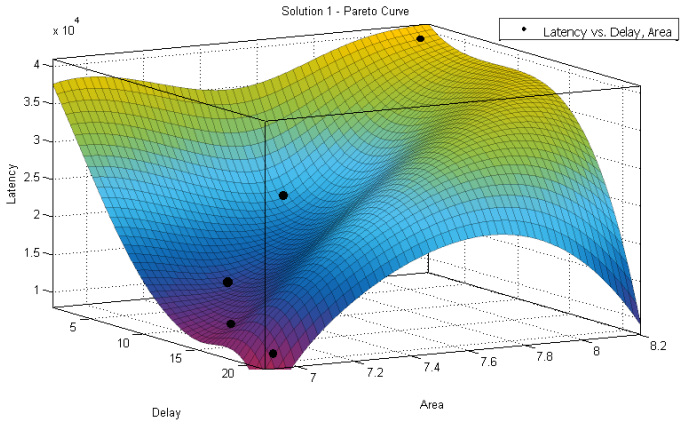

FIGURE 6.8: Configuration 1.1 – Trafe off Surface from Vivado HLS – Latency, Area, Delay

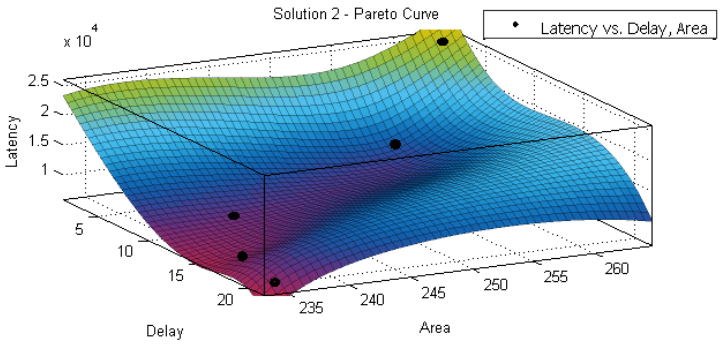

FIGURE 6.9: Configuration 1.2 – Trafe off Surface from Vivado HLS – Latency, Area, Delay

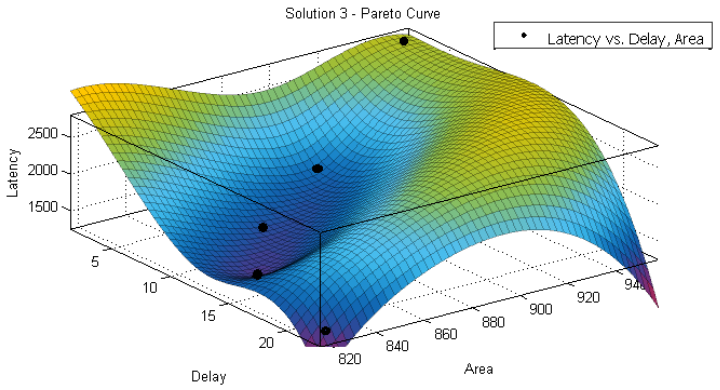

FIGURE 6.10: Configuration 1.3 – Surface from Vivado HLS – Latency, Area, Delay

6.3 Throughput Exploration

Having seen all previous results, obtained from the Vivado HLS, we can now proceed to throughput analysis, which is the outer purpose of this work.

Throughput calculation requires three variables: (i) latency of design, (ii) delay and (iii) the number of elements that are processed from every module. For pipelined implementations, interval must be used instead of latency, since we consider that pipeline is full, in order to calculate throughput. Interval, is number of cycles for a module to accept a new input. So in configurations x.4, we calculate throughput according to the interval, instead of latency.

In Subsections 6.3.1 and 6.3.2, we present two kind of tables regarding throughput. The first one, are tables with *pixel/cycle* metrics and the second kind on *samples/sec* results which are based on the former tables. When we refer to term "samples", we mean residuals, because as mentioned in prior chapters, the output in most cases of IIT module is the error of pixels in spatial domain. It is straightforward to see, that the entire performance of the IIT module, depends on the throughput performance of the 1-D transform sub-modules. Thus, we present their throughput results to identify how affect the performance of the top module.

Subsection 6.3.1, provide results about the worst and best case of throughput that is based on sub-modules $4x4$, $8x8$, $16x16$ and $32x32$. Additionally, in Subsection 6.3.2,

weighted results are presented in order to have a more realistic view of design's throughput. Actually, weighted throughput determines the real specifications of the design, since min and max throughput are unattainable performances.

In two final Subsections 6.3.3 and 6.3.4 we use throughput results, in order to compare them with other software and hardware implementations and also to realize for which video resolutions and fps, inverse transform module becomes a critical component for a hardware video decoder application and in which degree, it constraints the throughput performance of a video decoding application.

6.3.1 Min–Max Throughput

Here we provide results, regarding minimum and maximum throughput performance, in order to realize the limits of the implementation that HLS outputted, always having in mind the area cost that we pay for each solution, according to results from Subsection 6.2.

The calculation of these results are based on the best and worse in terms of $pixel/cycle$ modules, in order to find the maximum and minimum throughput respectively. One always needs to keep in mind, the delay that have been used for throughput calculation is 2.39 ns, where it is the smaller possible delay that HLS could achieve. In previous Section 6.2, we showed that while latency increases sub-linearly with delay, throughput increases as well. Hence, we considered the smallest delay ($2.39ns$), to maximize possible throughput of each sub-module. The calculation of $pixel/cycle$ results is based on equation 1. Pipelined RTL, provide for each sub-module, $2samples/cycle$ for both worst and best case (all sub-modules have the same performance), because the number of samples that each module process, increases linearly with the level of pipeline that HLS tool achieves. Thus, in each sub-module the throughput is the same and equal with $2samples/cycle$.

$$Samples/Cycle = \frac{Samples}{Latency_{cycles}} \tag{1}$$

$$Samples/Second = \frac{Samples}{Cycle} \times \frac{1}{Delay} \tag{2}$$

The above Equation 1, is used to calculate which of the sub-modules have the best and worst performance, so to be used in final throughput results for the limits of the design. Using previous results, we can now proceed to calculate the throughput results. One

needs, only multiply previous results with the frequency of the design, because having *samples/cycle* results, we have to multiply with the number of the cycles in one second, thus translating into *samples/second* domain. Calculation of $Msamples/sec$ is based on Equation 2.

In Tables 6.14, 6.16 and 6.18, we present results about throughput for both 1-D and 2-D transform modules. One can be easily inferred from those tables, 2-D modules have about the half throughput of 1-D. This is a rational outcome, because 2-D transform is essentially the same as using twice the 1-D and therefore the worst and best cases for 2-D and 1-D differ by a factor of two, in most configurations.

Considering all the configurations, we can see how throughput is affected from the architectural choice that have been made, according to directives we have given in HLS tool. As the level of parallelism get increased, throughput increases as well, because latency falls down. In configurations x.4, which are directed for a pipeline design, in each 1-D sub-module, throughput is maximized, because instead of latency, we have used interval time. Hence, while moving in higher solutions for the same source code, throughput is increased according to the level of parallelism and whether the design is pipelined or not. However, moving on higher throughput, we pay higher cost in terms of device utilization, as extensively discussed in Section 6.2.

Throughput in pipelined RTLs, is four up to ten times the throughput of other configurations, where HLS tool tries the same techniques, in order to decrease latency and therefore the number of hardware resources is roughly the same. Paying an additional small cost in averaged device utilization –about 5% when we move from configurations x.3 to x.4–, the pipeline version provides up to 10 times better performance, when compared with a non-pipeline version.

Comparing now the three different codes we tried on, we can see that in most cases the $4x4$ module gives the best throughput and $32x32$ the worst, because latency is increased according to sub-module's complexity, unlike to custom implementations that their latency increases sub-linearly with module complexity. This implies that in custom implementations, $32x32$ module gives best throughput and $4x4$ gives worst, while in implementations that are based on HLS RTL outputs, $4x4$ gives best and $32x32$ worst throughput performance. In some cases, for example in configurations 3.1 – 3.4, we can see that $32x32$ module becomes best in throughput, because latency in decreased compared to other input code's implementations.

At first, comparing the reference source code with the inline shift-add one, we realize that in best throughput case, reference code gives better results, because its $4x4$ module achieves lower latency than shift-add code in 2.39 ns delay. This happens, because inline

shift-add code, uses DSP48E modules in order to perform shift-add operations and this results in higher latency. Now, considering worst case results we again faced the high latency due to the way that Vivado tool manipulates shift-add operations. Comparing results from solution 2 at Tables 6.14 and 6.16 we can see that throughput in latter case is better. Concluding this comparison, we realize that reference code has better throughput results in most cases, when compared to inline shift-add code results. So, the only benefit we have from this second source code configuration, is the device utilization and especially the reduced number of DSP48E slices.

This paragraph, compares the third source code (function shift-add) with previous results. Here DSP48E slices are disappeared and then all shift and add operations are implemented on LUTs and FFs. One can be inferred by observing Table 6.18, throughput is increased because latency has considerably reduced. In all worst cases, throughput has been increased, when compared with the other two source codes. In best cases, it has been significantly improved, in comparison with inline shift-add code and in some cases it is better than reference code implementation. This improvement for the third code we have given, is noticed because $4x4$ is not always the best module in terms of throughput. Other modules such as $16x16$ and $32x32$ gives now better results due to the smaller latency that achieved. Hence, function shift-add code, is the most close approach to custom integer transforms implementations when throughput is a critical factor for design's compatibility. The most valuable result in this latest change of source code, DSP48E module utilization is completely eliminated, thus giving more available resources to other components of a video decoder. In order to realize what we have gained, if we consider throughput results in solution 3 for Tables 6.14 and 6.18, for reference and function shift-add codes respectively, we can see that throughput is doubled without using any DSP48E slices.

To conclude, we can see that pipeline RTLs has exactly the same throughput performance in all different source codes, because HLS tools seems to make same pipeline depth, for all the different source codes. Although, in function shift-add code that doesn't use DSP48E slices, performance is achieved using very few hardware resources and this makes implementation more power friendly and area non-demanding.

TABLE 6.14: Top Module Throughput in $Msamples/sec$ Min–Max results from reference code implementation

Msamples/sec	**2-D Worst**	**2-D Best**	**1-D Worst**	**1-D Best**
Config–1.1	10.89	53.13	21.8	106.26
Config–1.2	13.81	90.46	27.62	180.93
Config–1.3	99.17	167.36	192.35	334.72
Config–1.4	418.41	418.41	836.82	836.82

TABLE 6.15: Sub-Modules Throughput in *Samples/cycle* Min–Max results from reference code implementation

Samples/cycle	**Config–1.1**	**Config–1.2**	**Config–1.3**	**Config–2.4**
4x4	0.25	0.43	0.8	2
8x8	0.08	0.14	0.47	2
16x16	0.062	0.066	0.6	2
32x32	0.052	0.08	0.75	2

TABLE 6.16: Top Module Throughput in $Msamples/sec$ Min–Max results from inline shift-add code implementation

Msamples/sec	**2-D Worst**	**2-D Best**	**1-D Worst**	**1-D Best**
Config–2.1	9.1	42.37	27.32	84.74
Config–2.2	19.14	42.37	59.92	84.74
Config–2.3	42.37	148.76	84.74	297.53
Config–2.4	418.41	418.41	836.82	836.82

TABLE 6.17: Sub-Modules Throughput in *Samples/cycle* Min–Max results from inline shift-add code implementation

Samples/cycle	**Config–2.1**	**Config–2.2**	**Config–2.3**	**Config–2.4**
4x4	0.2	0.2	0.2	2
8x8	0.09	0.14	0.25	2
16x16	0.08	0.09	0.71	2
32x32	0.06	0.14	0.66	2

TABLE 6.18: Top Module Throughput in $Msamples/sec$ Min–Max results from function based shift-add code implementation

Msamples/sec	**2-D Worst**	**2-D Best**	**1-D Worst**	**1-D Best**
Config–3.1	21.8	53.13	43.6	106.26
Config–3.2	34.23	53.13	68.46	106.26
Config–3.3	53.13	225.5	106.26	451.1
Config–3.4	418.41	418.41	836.82	836.82

TABLE 6.19: Sub-Modules Throughput in $Samples/cycle$ Min–Max results from function based shift-add code implementation

Samples/cycle	**Config–3.1**	**Config–3.2**	**Config–3.3**	**Config–3.4**
4x4	0.25	0.25	0.25	2
8x8	0.14	0.21	0.37	2
16x16	0.12	0.12	0.64	2
32x32	0.14	0.16	1.07	2

6.3.2 Weighted Throughput

Weighted results are provided in order to see a more realistic throughput performance of the Inverse Transform top module. As already mentioned in the introduction of Section 6.3, min and max throughput performance is determined from best and worst sub-module respectively. A video bitstream cannot contains entirely one kind of TU and so using only one sub-module from top module. What we try to clarify, in a video sequence all the sub-modules are used in some percentage that fluctuates according to video's resolution and the Quantization Parameter (QP) which is used in encoding process. Therefore, min and max throughput are practically unattainable performances, and are calculated only to see the limits of different RTLs.

The calculation of weighted results, is based on the occurrence's likelihood for every type of TU. In order to make this state crystal clear to reader, let us consider the following scenario. Let us assume that we design a circuit that is going to be integrated in a video decoder top module. If we take best throughput for the specifications of design –that is based on the best sub-module's throughput– is an erroneous practice. In a true video sequence, definitely will be used in some percentage all other modules, thus reducing top module's throughput performance, because the rest sub-modules will have poorer performance. Hence, in order to calculate a more truthful and realistic throughput performance, which is going to be used for the specifications of top module, we have to obtain statistics from several videos. In doing so, we can find the average percentage of TUs that occur in different video resolutions and upon these results, to make an estimation about the average throughput that must have the module of Inverse Transform. As we can see in Tables 6.15, 6.17 and 6.19, the throughput performance of sub-modules $4x4$, $8x8$, $16x16$ and $32x32$ differs and since top-module's throughput depends on them, the average throughput of top module is depend on the percentage of sub-module's utilization, or else the percentage of TUs that exist in the encoded bitstream.

For this purpose, we trained a lot of different reference video bitstreams in resolution and QP that were obtained from JCT-VC database [45], in order to acquire statistics about the percent of usage for each TU. Statistic results are shown in Table 6.20. Results that are cited in Tables 6.21, 6.22 and 6.23, show the weighted performance according to likelihood of each size of TU that is based on throughput ($samples/cycle$) of each of four sub-modules. Results on these tables are calculated according to Equation 3, where i variable chooses current TU, W is the corresponded weight and $samples/cycle$ refers to throughput of each TU.

$$Weighted\_Throughput = \left(\frac{1}{delay}\sum_{i=0}^{3} W_i * (Samples/cycle)_i\right) \quad (3)$$

Obtaining statistics from several videos, we have found some interesting clues regarding video coding that it is worth to mentioned here. In low resolution videos, most common TUs are the two smallest $4x4$ and $8x8$ and as moving in higher video resolution, the percentage of $16x16$ and $32x32$ TUs get increased. This is a rational outcome, because videos in higher resolutions have wider areas that can be predicted, thus making larger TUs more efficient, to transform the residual error. Another interesting result we observed, as QP increases the percentage of larger TUs such as $16x16$ and $32x32$ also increases because encoder tries to achieve better compression result using larger TUs, in order to make greater energy compaction. Hence, we do know now that the more resolution and QP increases, the more large sub-modules are utilized and their weight to averaged throughput of top-module becomes greater.

Observing tables in this subsection, it is easy to realize that throughput increases when best sub-module in terms of throughput gets more weight. Comparing these results with Tables 6.15, 6.17 and 6.19, where we show the throughput of sub-modules, we can see that the greater percentage it takes the best module, the more weighted throughput is increased in top-module. In following tables, we illustrate throughput results, for different configurations of HLS design space exploration and for different video resolutions that are commonly used in video technology. Different video resolutions, use in different percentage the TUs and therefore weights change, resulting in different throughput performance of top-module. A general observation that can be made from these tables, as video resolution increases, the percentage of $4x4$ TU which has the best performance, falls down resulting in lower throughput at top module. Of course, in configuration 3.3 for example, where $32x32$ sub-module has the best performance, moving in higher video resolutions, throughput increases because more $32x32$ TUs are used, when compared to lower video resolutions. Hence, designer has to take into account, the video resolution of the application and the variation of QP, to decide among the various configurations, which of them best fits to application's throughput requirements.

TABLE 6.20: Results from several reference video bitstreams, regarding TU utilization for different resolutions and QPs

Video / TU-size	**4x4**	**8x8**	**16x16**	**32x32**
240p-QP=22	70.5%	24%	4.5%	1%
240p-QP=37	45.5%	32.5%	17%	5%
480p-QP=22	56.3%	30.3%	10.6%	2.8%
480p-QP=37	20%	40%	27.6%	12.4%
720p-QP=22	36%	41%	19%	4%
720p-QP=37	4%	33%	39%	24%
1080p-QP=22	18%	38%	28%	16%
1080p-QP=37	8%	27%	37%	28%

TABLE 6.21: Reference code weighted throughput for different video resolutions

Msamples/sec	**Config–1.1**	**Config–1.2**	**Config–1.3**	**Config–1.4**
240p	75.21	126.24	286.51	836.82
480p	59.82	98.52	269.57	836.82
720p	45.11	71.66	256.92	836.82
1080p	38.72	59.94	258.47	836.82
Average	54.71	89.09	267.87	836.82

TABLE 6.22: Inline Shift–add code weighted throughput for different video resolutions

Msamples/sec	**Config–2.1**	**Config–2.2**	**Config–2.3**	**Config–2.4**
240p	64.66	71.54	119.59	836.82
480p	54.33	64.23	148.14	836.82
720p	44.14	56.01	181.53	836.82
1080p	40.63	54.08	228.18	836.82
Average	50.94	61.41	169.36	836.82

TABLE 6.23: Function Shift–add code, weighted throughput for different video resolutions

Msamples/sec	**Config–3.1**	**Config–3.2**	**Config–3.3**	**Config–3.4**
240p	85.73	95.16	153.62	836.82
480p	75.18	87.93	195.37	836.82
720p	64.71	79.67	245.19	836.82
1080p	59.75	75.35	290.46	836.82
Average	71.34	84.53	221.12	836.82

6.3.3 Comparing Other implementations

In this subsection, we carry out a small survey, in order to realize how our implementation is compared to other implementations from research community. In doing so, we are trying to find out, where Vivado HLS tool stands, among other levels of implementation such as pure software, SIMD and custom hardware RTLs, in terms of throughput performance.

6.3.3.1 Reference Software Implementation (x86)

In order to find the throughput of a software implementation that is ported on an AMD processor, we had to carry out a profiling on HEVC video decoder, thus obtaining results about how much time takes each of four sub-modules to finish with its task. Finding this execution time, from a classic profiling process, it is easy to find the latency in terms of cycles for each sub-module, just by multiplying with the operating frequency of the processor that was used Eq. 4. The operating frequency of the processor that utilized is 1.8 GHz.

$$Latency = (TimeExec) \times (Frequency) \tag{4}$$

Having now latency, we can proceed to throughput results in order to compare with our HLS implementation. Table 6.27 illustrates all the results regarding the performance of four sub-modules. We compare here only results of 1-D sub-modules, just to compare the two different implementations without calculating min, max and weighted results, because it is an easy task, having these results to find everything. The best metric we have to use in order to compare two different implementations is $Msamples/sec$ which directly depicts the throughput performance. For this reason, we present here three Tables 6.24, 6.25 and 6.26 which are the same with Tables 6.15, 6.17 and 6.19 respectively, but now they have results in $Msamples/sec$ domain instead of $Msamples/cycle$.

Comparing our HLS implementation with a software running on a general purpose processor, we can see that each of them has some pros and cons. Processors running in high GHz frequencies, executing much more cycles in time unit, comparing with an FPGA that is impossible to reach that frequencies. On the other hand, FPGA implementations using HLS tool, may create better architecture for each sub-module, thus requiring less cycles to perform their task and despite running in relatively lower frequency, to provide better throughput results.

TABLE 6.24: Reference code sub-module's throughput

Msamples/sec	**Config–1.1**	**Config–1.2**	**Config–1.3**	**Config–1.4**
4x4	106.26	180.93	334.72	836.82
8x8	35.84	61.27	198.35	836.82
16x16	26.02	27.62	250.26	836.82
32x32	21.8	34.23	314.34	836.82

TABLE 6.25: Inline shift-add code sub-module's throughput

Msamples/sec	**Config–2.1**	**Config–2.2**	**Config–2.3**	**Config–2.4**
4x4	84.74	84.74	84.74	836.82
8x8	40.14	60.44	106.68	836.82
16x16	33.77	38.29	297.53	836.82
32x32	18.21	39.94	277.31	836.82

Throughput of $4x4$ module-function running on AMD processor is 81.63 $Msamples/sec$, while HLS implementation provide solutions that their performance are slightly or up to ten times higher than software. Module $8x8$, has also better performance in FPGA expect for Solution 1 which in reference and inline shift-add implementation has worse performance. Module $16x16$, seems that in solution 1 and solution 2 have lower throughput than software, but in solution 3 and 4, it is considerably higher. Finishing, module $32x32$ seems that in software cannot be executed efficiently, because it has lower performance than every solution, so in higher resolution videos it will go back the entire performance of xITrMxN which is the function that performs the inverse transformation. Software implementations, when compared with an HLS implementation, seems that are scaled on the same way, moving in more complex modules such as $16x16$ and $32x32$. Only function shift-add configuration retains latency in low levels for these sub-modules, thus achieving satisfactory throughput results. Of course, we do expect that custom RTL architectures shall have much more performance –especially for sub-modules $16x16$ and $32x32$– because latency doesn't scales linearly with sub-module's complexity. This is going to be further discussed in Subsection 6.3.3.3.

TABLE 6.26: Function shift-add code sub-module's throughput

Msamples/sec	Config–3.1	Config–3.2	Config–3.3	Config–3.4
4x4	106.26	106.26	106.26	836.82
8x8	61.55	92.02	156.59	836.82
16x16	50.26	50.98	268.45	836.82
32x32	43.60	68.46	451.1	836.82

TABLE 6.27: Throughput results from reference code running on an AMD processor

TU-Size	Time (μsec)	Latency	Samples/cycle	Msamples/sec
4x4	0.196	353	0.045	81.63
8x8	1.15	2070	0.03	55.65
16x16	6.17	11106	0.023	41.4
32x32	60.07	108000	0.0024	17.04

6.3.3.2 SIMD–Reference Software

Single Instruction Multiply Data (SIMD) has been extensively used in general processors such as Intel-AMD, in order to speedup an application that its algorithms have a vectorized/parallel nature. This technique can be considered as a hardware accelerated technique because it utilizes wider registers that can perform arithmetic operations in parallel. In this subsection, we are going to see how much optimization we can gain from SIMD techniques and how this directly affects into throughput performance.

Taking into account a survey [5], regarding HEVC decoder optimization with SIMD in several platforms, we focus on results about Inverse Transform module. According to authors, SIMD implementation of the inverse transform can be either performed inside one column or row transform, or using SIMD on multiple columns/rows, or a combination of the two. They have experimented with these approaches and found that the differences are marginal. The fastest SSE2 implementation uses SIMD over columns, followed by a transpose for both passes of the inverse transform.

Using previous techniques they are achieving a speedup 3.6× and 4.8× for SSE2 and AVX2 respectively for the inverse transform kernel. Hence, let us consider an averaged speedup of 4.2× for both SSE2 and AVX2 to see how it affects throughput. According to Eq. 4, lessening time execution by 4.2×, results in this amount of latency's reduction.

TABLE 6.28: Throughput results from reference code after SIMD optimization, on general purpose microprocessor

TU-Size	**Time (μsec)**	**Latency**	**Pixels/cycle**	**Msamples/sec**
4x4	0.046	84	0.189	342.8
8x8	0.274	492	0.126	233.73
16x16	1.47	2644	0.096	173.8
32x32	14.3	25714	0.01	71.7

Therefore, reducing latency, throughput is increased by the same amount of speedup. Following Table 6.28 is the same with Table 6.27, from previous subsection and shows the results after SIMD optimization, in order to illustrate how they differ from HLS performance.

As we can see, $4x4$ module of software has better performance now from all the configurations. Only solutions 4 - pipeline RTLs, provide considerably better throughput results than the optimized software with SIMD. The only comparable throughput from the other configurations, comes from solution 3 in reference software, because the other two codes cannot achieve sufficient results in $4x4$ sub-module. Module $8x8$, in HLS implementation has also poorer performance than software, except of course the solution 4 results. Module $16x16$, provide better results only for solution 3 and 4, for all the configurations, while at solutions 1 and 2 it has lower performance. Module $32x32$, is compared on the same way as $16x16$, but solution 2 is more close to software results. So, we are realizing that increasing the size of module, software yields more inefficient functions, while HLS makes more efficient –in terms of throughput– modules. As we are going to see in next subsection, custom RTLs provide even more greater results, as the size of the sub-module increases.

6.3.3.3 Custom Hardware RTL

In our latest effort to compare our HLS implementation using Vivado, we have to compare with existing custom RTL cores from the research community. In fact, these two implementations are identical because in either cases, an RTL model will be used either on ASIC or on FPGA to implement IIT. Software implementations running on processors have different characteristics from hardware, so we had to convert results in same metric ($Msamples/sec$), in order to compare them. Here, results can equally be compared

TABLE 6.29: Latency and Throughput results for 4, 8 and 16 core transform for FPGA @ 251 MHz

TU-Size	**Latency**	**Pixels/cycle**	**Msamples/sec**
4x4	14	1.14	286
8x8	16	4	1004
16x16	20	12.8	3212

because RTL level is tested in either cases with one difference; HLS, derives RTL from standard templates that have been exported from a software tool, while custom RTL, is designed from engineers, thus being more efficient in terms of performance, power and area. The only comparison that will be conducted in this subsection is about throughput performance because we desire to understand how far Vivado HLS tool stands, from custom RTL implementations with HDL.

We have to declare here that all researches have been conducted mainly for the direct transform. However, direct and inverse transform have the same complexity and structure, as discussed in prior Chapter and therefore their performance can be compared like being the same module. All surveys that have been carried out, essentially provide throughput results for 1-D core architecture, so we are comparing performance of 1-D sub-module instead of 2-D, since all implementations utilize two 1-D transform cores interleaved from a transpose buffer. Also, most of the implementations are converting multipliers into shift-add operations for area reduction, smaller critical paths and latency, as we did in software level as well.

At first we are taking into account first results in chronological order regarding Integer Transform that Ricardo J. et. al. have conducted [7]. A 314.6 $Msamples/sec$ is achieved in reference implementation and a 1401.6 $Msamples/sec$ in their optimized implementation, where they are have eliminated multipliers and they share hardware for 16x16 transform. In [8], researchers are conducting an efficient architectural approach for 4, 8 and 16 transform modules, using some piece of logic to control how often a new input can be inserted in a sub-module. $32x32$ transform wasn't implemented due to its complexity and work load and they left it as future work. Their results regarding latency and throughput are presented in Table 6.29 for the three sub-modules they tried on.

Other surveys such as [9], [10] and [11], use either folded structures with one 1-D module, for the 2-D transform, in order to reduce area or use fully parallel implementations using

two such modules. Also, some of them utilize pruned schemes of integer transform in order to reduce area and latency. Prune method is used in many papers as a proposed architecture; to reduce the computational complexity, some of the least significant bits (LSB) can be pruned before they insert in functional units for calculation. This is applicable since after inverse transform, a left shift will be performed, so some of LS bits do not required. In this way, a small error is inserted on calculations but the impact of the pruning on the final output is not significant. According to those papers, the most we prune bits, the more big error is introduced into final output of integer transform. Previous papers achieve sufficient results, regarding throughput for the 1-D core module, which fluctuates from 0.63 up to 2.99 Gsamples per second (Gsps) as they illustrate in their results section.

To conclude this subsection, we immediately realize that results from custom-made RTL architectures, provide better results than our implementation that is based on HLS synthesis results. The only comparable throughput from our implementations is Solution 4 (pipeline architecture) for all source codes, which provides throughput of 0.836 Gsps for 1-D transform modules and $Pixels/cycle$ equal to 2. Above custom hardware implementations, can support video requirements up to 7680x4320 @ 60 fps. Generally, we can say that in custom RTLs, throughput from $16x16$ and $32x32$ transforms is better than 4x4 for example, while in HLS RTL designs, occurs the opposite situation. One immediately inferred, human can make much more efficient optimizations in complex algorithms, in comparison with software tools, that scale module's latency, according to algorithm's complexity.

6.3.4 Supporting Different Videos

In this final subsection of this Chapter, we evaluate how our HLS implementation of HEVC inverse integer transform becomes critical module in a hardware video decoding application and what is the highest throughput performance that it requires, so to support a standard video resolution and fps.

A hardware video decoding application has several modules inside, thus performing several different tasks of video decoding algorithms. Such modules for instance are: inverse transform, motion compensation, in-loop filtering, Context Adaptive Binary Arithmetic Encoding (CABAC) e.t.c. Each of them, can be fully characterized from its performance in regards to throughput. According to the type of each module, throughput can be measured with different metrics. For those modules which have pixel related operations, throughput is measured in $Mpixels/sec$ or in general with $Msamples/sec$

TABLE 6.30: Throughput requirement in (Msamples/sec) for different resolutions of video and frame rate for YUV 4:2:0

Resolution	**FPS=30**	**FPS=50**	**FPS=60**	**FPS=120**
416x240	4.49	7.48	8.98	17.97
832x480	17.97	29.95	35.94	71.88
1280x720	41.47	69.12	82.94	165.88
1920x1080	93.31	155.52	186.62	373.24
3840x2160	373.24	622.08	746.49	1492.99
7680x4320	1492.99	2488.32	2985.98	5971.96

which describes pixels, residuals and coefficients. In bit related modules such CABAC, throughput is measured in $Mbin/sec$ because its input, is on bit level.

For every module which is component of a hardware video decoder, can be calculated a throughput requirement for different video resolutions and fps, just in order to realize whether each module satisfies this requirement or not. Hence, the total throughput performance of the video decoding application is constraint by the poorest module in terms of throughput. For example, if we have an HEVC video decoder in hardware and all modules are supporting UHD @ 60 fps, except for one module which supporting throughput for HD @ 30 fps, then total throughput performance of HEVC decoder is constraint at HD @ 30 fps, because if we move on higher video resolutions, video shall stalls, because this module outputting results in lower rate that it is required. In Equation 5, we show how we are calculating throughput requirement for each video class and it is the method on which all other surveys base their results. Table 6.30 shows the throughput requirement for different typical video resolutions and frame rates, considering YUV 3 channels with format of 4:2:0 (it is represented by 1.5× factor), which is the most prevalent in video sequences.

$$Throughput = Width \times Height \times FPS \times 1.5 \tag{5}$$

Having analyzed the method on which we are calculating throughput requirements, let us see each different RTL from HLS, what videos from table 6.30 can support. In order to carry out the comparison, we are going to use Tables 6.21, 6.22 and 6.23 because we consider that it is better to compare with more realistic results than minimum or

maximum ones and also each video resolution has its own characteristics as concerning TU-weights; so in each video we are exploiting different statistics.

Videos in resolution of 240p are supported from every different RTL as we can see from those tables. 480p videos are supported from every solution in function shift-add code and only at 120 fps there is no module to support it, for the other two source codes. Videos with resolution 720p are supported from every solution @ 30 fps, but as frame rate gets increased, higher solutions such as 3 and 4 are the only which support this demanding performance. Regarding now 1080p video, solution 3 can support all frame rates except for the highest –to wit 120 fps– and solution 4 supports all the frame rates of full HD videos. For resolutions higher than Full HD, such as Ultra High Definition (UHD-video) the upper limit that HLS can reach, is 2160p @ 60 fps as can easily implied by respective tables. For supporting more demanding videos, higher than UHD or in higher frame rates, custom architectures should be definitely used, in order to support those extremely demanding processing rates.

Fig. 6.11, shows the cheapest RTL – configuration –in terms of device utilization– that can support some of the more usual video contents that can be met in a video application. For each video, three configurations from three different codes are illustrated, because each code has different utilization percentages for different hardware resources, For example, if a configuration with as much less LUTs is required, one needs, only to choose configurations from reference code. If on the other side, DSP48Es slice's utilization is required to be eliminated (other video modules may want to utilize this resource as well), function shift–add code's RTL would be a good match, while intermediate requirements are covered by designs from inline shift–add code.

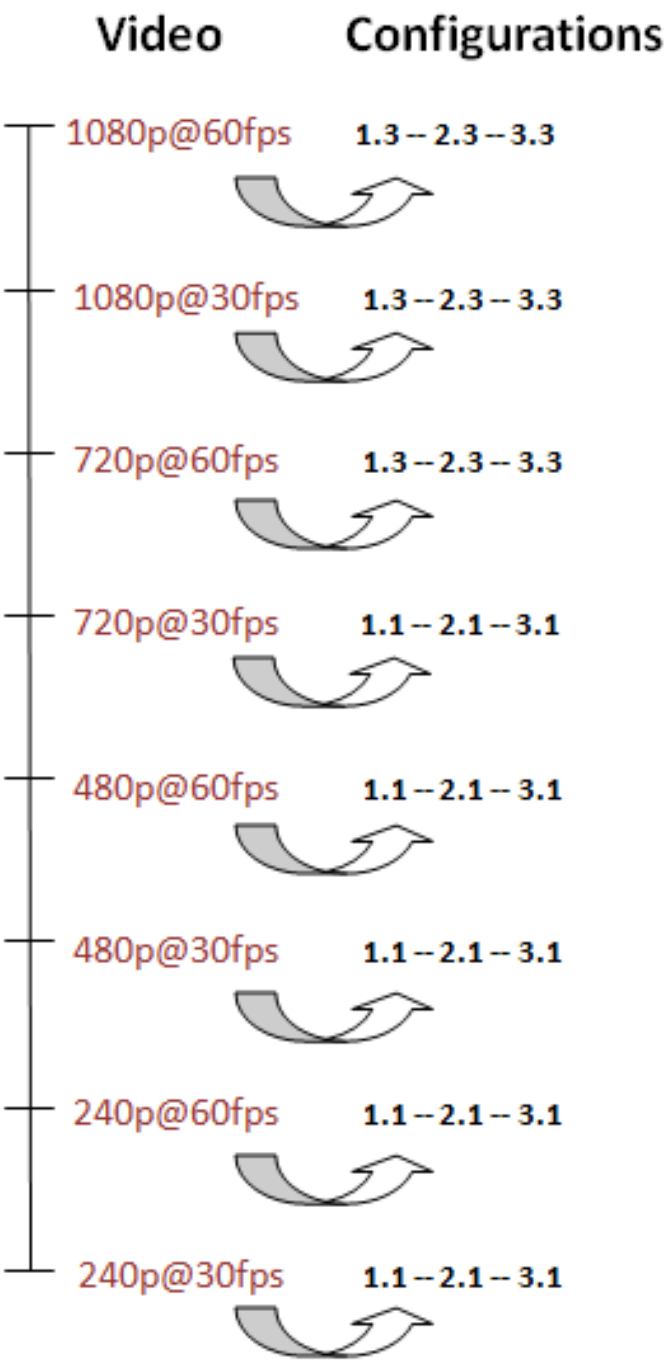

FIGURE 6.11: Usual video contens and which congigurations can support them with minimum device utilization. Solutions from three codes are presented, because each of them may support a video, reserving different percentages for FPGA resources.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

To conclude this thesis, we are conducting a brief review about the main fulfillments of this work that were presented mainly in Chapter 6. We accomplished the main objective of this work, the design space exploration of the HEVC IIT module, exploiting the high flexibility and automation that Vivado HLS tool enables. This work, provides out-of-the-box RTLs for FPGA, with different characteristics each of them, implementing the HEVC Inverse Transform module. Throughput performance, was further explored for different configurations, at the top- and sub-module level. We showed that in order to calculate a more realistic throughput performance for the HEVC IIT module, weighted throughput must be used, because min and max throughputs are unattainable performances, since all types of TUs and so the respective sub-modules, are used in some percentage. For this reason, we conducted a profiling process using different reference video bitstreams, so as to identify how the utilization of TUs change in different videos and to create weights for weighted throughput calculation. Besides, a small survey about different levels of implementation carried out. Software performance was measured in a x86 architecture via a simple profiling process, to realize the throughput of the four sub-modules. Moreover, other work's results from SIMD accelerated software were utilized, to figure out how far –in terms of throughput– can reach software implementations. Other works that deal with custom hardware RTLs were illustrated and discussed and their results were used to compare our HLS configurations. Finally, we showed which are the cheapest HLS configurations that can be used for supporting different video contents and which is the highest video content that our HLS designs can support (2160p@60FPS). Alongside with the throughput exploration, we identified how

Vivado HLS tool reacts on different target clock periods and how it trades device utilization and latency for delay. The latency control that HLS tool maintains for pipelined and no-pipelined circuits, have been largely examined (see Chapter 4).

Summarizing, we have to state here some deductions about our experiment, the results that obtained and an overall review. As concerning the High-Level Synthesis tools for synthesis and verification of an algorithm, the great flexibility that they provide for exploring different RTLs quickly, show how time-to-market can be shortened for the development of a hardware design. The different RTLs from Vivado HLS synthesised and verified in short time (about 3 man months); if they had been created with an HDL description, the required time would be much more times higher than that of HLS. Essentially, the higher amount of this work was spend to learn the HLS tool about how it reacts in different code versions and different input constraints and directives. After then, the process was simplified and the high level automation of HLS, let for quick navigation through different RTL architectures. However, as we can see from results, HLS provide poor throughput performance, in comparison with custom RTLs, since it stands close to software performance. In particular, to achieve comparable throughput with custom RTLs, huge hardware resources have to be allocated, thus requiring big devices only for this module of HEVC decoder. So, we claim here that HLS RTLs are inefficient when compared to custom ones, since they require much more area or device utilization, to achieve a certain throughput requirement. HLS tools, have a long way to go, in order to derive RTLs comparable with custom architectures, in terms of efficiency (performance/area). The HLS concept has special worth in industry (shorter time translates into less expenditures), so in future is expected to be a prominent method for hardware design, thus overcoming the high design cycle of custom RTLs.

7.2 Future Work

As mentioned in prior chapters, HEVC provides the best coding efficiency, when compared to other standards, but is the most demanding video codec, since it maintains higher complexity in its modules. Hence, in order to meet a real-time constraint in video coding, some parts of it should be implemented in hardware. Therefore, this work could be extended in other HEVC modules such as: Motion Estimation-Compensation, In-loop filters, CABAC, e.t.c.. Utilizing a HLS tool, we can quickly explore different architecture solutions for a specific algorithm and to compare them with other level of implementations, in terms of throughput or power. Finally, it would be interesting to identify, all these different HEVC modules yielded from a HLS process, how they can be integrated into a real video coding pipeline that its control unit has been created from a

HLS tool as well. The total performance of an HEVC video encoder-decoder, exclusively created from a HLS process, could be compared with other existing works in terms of total throughput, area, power and design time.

Bibliography

[1] G. J. Sullivan, Ohm Jens-Rainer, Woo-Jin Han, Wiegand Thomas, Overview of the High Efficiency Video Coding (HEVC) Standard, IEEE Trans. Circuits Syst. Video Technol., vol.22, no.12, pp.1649-1668, Dec. 2012

[2] T. Wiegand, G. J. Sullivan and A. Luthra, Draft ITU-T Rec. H.264/ISO/IEC 14496-10 AVC, JVT of ISO/IEC MPEG and ITU-T VCEG, Doc. JVT-G050r1, 2003.

[3] Cisco Systems, Inc., "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 20102015," 2011.

[4] M. Viitanen, J. Vanne, T.D. Hamalainen, M. Gabbouj, J. Lainema, "Complexity analysis of next-generation HEVC decoder", In Circuits and Systems (ISCAS), 2012 IEEE International Symposium on (May 2012), pp. 882-885

[5] Chi Ching Chi,, Mauricio Alvarez-Mesa, and Ben Juurlink, "SIMD Optimization of HEVC Decoding for General Purpose Processors", August 26, 2014.

[6] Hao Lv, Ronggang Wang, Jie Wan, Huizhu Jia, Xiaodong Xie, Wen Gao, "An Efficient NEON-based Quarter-pel Interpolation Method for HEVC", Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific. pp. 1-4, Dec. 2012.

[7] Ricardo Jeske, Jos Cludio de Souza Jr., Gustavo Wrege, Ruhan Conceio, Mateus Grellert, Jlio Mattos and Luciano Agostini "Low Cost and High Throughput Multiplierless Design of a 16 Point 1-D DCT of the New HEVC Video Coding Standard", VIII Southern Programmable Logic Conference (SPL), March 2012.

[8] Fatma Belghith, Hassen Loukil and Nouri Masmoudi, "Efficient Hardware Architecture of the Direct 2-D Transform for the HEVC Standard", International Scholarly and Scientific Research & Innovation 7(6) August 2013.

[9] Pramod Kumar Meher, Sang Yoon Park, Basant Kumar Mohanty, Khoon Seong Lim and Chuohao Yeo, "Efficient Integer DCT Architectures for HEVC", IEEE Transactions on Video Technology, August 2013, pp. 168-178.

[10] S. Shen, W. Shen, Y. Fan, and X. Zeng, "A unified 4/8/16/32-point integer IDCT architecture for multiple video coding standards in Proc. IEEE International Conference on Multimedia and Expo, Jul. 2012, pp. 788793.

[11] J.-S. Park, W.-J. Nam, S.-M. Han, and S. Lee, "2-D large inverse transform (16x16, 32x32) for HEVC (high efficiency video coding)" Journal of Semiconductor Technology and Science, vol. 12, no. 2, pp. 203211, Jun. 2012.

[12] Shihao Wang, Dajiang Zhou, and Satoshi Goto, "Motion compensation architecture for 8K UHDTV HEVC decoder". Multimedia and Expo (ICME), 2014 IEEE International Conference, pp. 1-6, July 2014.

[13] Zhengyan Guo, Dajiang Zhou, Goto S, "An optimized MC interpolation architecture for HEVC", Acoustics, Speech and Signal Processing (ICASSP), March 2012 IEEE International Conference, pp. 1117-1120.

[14] V. Sze and A. P. Chandrakasan, "A highly parallel and scalable CABAC decoder for next generation video coding, IEEE Journal of Solid-State Circuits (JSSC), vol. 47, no. 1, pp. 822, January 2012.

[15] Y.-H. Liao, G.-L. Li, and T.-S. Chang, "A Highly Efficient VLSI Architecture for H.264/AVC Level 5.1 CABAC Decoder, IEEE Transactions on Circuits and Systems for Video Technology (TCSVT), vol. 22, no. 2, pp. 272281, February 2012.

[16] Y.-H. Chen and V. Sze, "A 2014 MBin/s Deeply Pipelined CABAC Decoder for HEVC, to appear in IEEE International Conference on Image Processing (ICIP), 2014.

[17] Sangchul Kim, Hyunjin Kim, Jin-Gyeong Kim, "Design of H.264 Video Encoder with C to RTL Design Tool", SoC Design Conference (ISOCC), 2012 International, pp. 171-174, Nov. 2012.

[18] Jin Ho Han, Mi Young Lee, Younghwan Bae, and H. Cho, "Application Specific Processor Design for H.264 Decoder with a Configurable Embedded Processor". ETRI Journal, vol. 27, no. 5, Oct. 2005, pp. 491-496.

[19] V. Magoulianitis, I. Katsavounidis, "HEVC Decoder Optimization in Low Power Configurable Architecture for Wireless Devices", IEEE 16th International Symposium on World of Wireless, Mobile and Multimedia Networks, Boston MA, June 2015.

[20] C.-M.Fu et. al., "Sample Adaptive Offset nin the HEVC Standard , IEEE Transactions on Circuits and Systems for Video Technology, 2012

[21] Huffman, D. (1952). "A Method for the Construction of Minimum-Redundancy Codes" (PDF). Proceedings of the IRE 40 (9): 10981101. doi:10.1109/JRPROC.1952.273898.

[22] AM Tourapis, OCL Au, ML Liou, "Predictive motion vector field adaptive search technique (PMVFAST): enhancing block-based motion estimation", Photonics West 2001-Electronic Imaging, pp. 883-892.

[23] Il-Koo Kim, Woo-Jin Han, JeongHoon Park, Xiaozhen Zheng, "CE2: Test results of asymmetric motion partition (AMP), Torino, IT, Jul. 2011.

[24] https://en.wikipedia.org/wiki/Karhunen%E2%80%93Lo%C3%A8ve_theorem

[25] http://en.wikipedia.org/wiki/Fast_Fourier_transform

[26] http://en.wikipedia.org/wiki/CooleyTukey_FFT_algorithm

[27] http://en.wikipedia.org/wiki/Fast_Hadamard_transform

[28] Temics: Aurlie Martin: http://www.irisa.fr/temics/staff/martin/

[29] http://www.cs.cf.ac.uk/Dave/Multimedia/node231.html#DCTcomp

[30] http://en.wikipedia.org/wiki/Discrete_wavelet_transform

[31] http://www.xilinx.com/support/documentation/user_guides/ug369.pdf

[32] http://www.xilinx.com/support/documentation/user_guides/ug902.pdf

[33] Youn-Long Lin, "High-Level Synthesis of VLSI", Department of Computer Science National Tsing Hua University, Tsing Hua Electronic Design Automation.

[34] J. A. Abraham, "High Level Synthesis", University of Texas at Austin, EE 382V–SoC Design.

[35] Philippe Coussy, Michael Meredith, Daniel D. Gajski, Andres Takach, "An Introduction to High-Level Synthesis", Published by the IEEE CS and the IEEE CASS IEEE Design & Test of Computers.

[36] Cooley, James W., Tukey, John W. (1965). "An algorithm for the machine calculation of complex Fourier series". Mathematics of Computation 19 (90): 297301. doi:10.1090 S0025-5718-1965-0178586-1. ISSN 0025-5718.

[37] Miguel Lobato,"Advances on Transforms for High Efficiency Video Coding", Master Thesis, University de Lisboa.

[38] W. H. Chen, C. H. Smith and S. C. Fralick, "A Fast Computational Algorithm for the Discrete Cosine Transform " IEEE Transactions on communications, vol. com-25, no. 9, September 1977

[39] https://vsr.informatik.tu-chemnitz.de/~jan/MPEG/HTML/mpeg_tech.html

[40] Ahmed Hagag, Mohamed Amin, Fathi E. Abd El-Samie "Simultaneous denoising and compression of multispectral images", Journal of Applied Remote Sensing, Aug. 2013

[41] http://www.snipview.com/q/Deblocking_filter

[42] Andreas Unterweger,"Real time H.264 for Telemedicine applications", Salzburg University of Applied Science.

[43] https://en.wikipedia.org/wiki/Inter_frame

[44] https://en.wikipedia.org/wiki/MPEG-1

[45] https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/tags/