POLITECNICO DI TORINO

Master of Science Degree
in Electronic Engineering

Master's Degree Thesis

Simplified Affine Motion Estimation algorithm and architecture for the Versatile Video Coding standard

Supervisors
Prof. Maurizio Martina
Prof. Guido Masera

Candidate
Costantino Taranto

Academic Year 2021-2022

Summary

The demand for higher-quality video content from users grows over time. In this scenario, it becomes essential to rely on efficient video coding algorithms and standards to avoid huge memory and computational resource requirements. Versatile Video Coding (VVC) is the latest video coding standard developed by the Joint Video Experts Team and finalized in July 2020 in ITU-T as Recommendation H.266. It can achieve significant bit rate reductions in video stream storage and transmission, in the neighbourhood of 50% over its predecessor, HEVC, for equal video quality. This compression rate growth comes with increased complexity, lengthening the encoding time in VVC compared to previous standards for the same input data stream. To **reduce** the **computational** complexity **burden** on the encoding processors, a hardware accelerator has been designed.
This thesis work reports all the theoretical analyses and design phases which have brought to the final product: a synthesized netlist for the cited hardware accelerator.

The **first chapter** is an introduction to the whole dissertation. *Versatile Video Coding* (VVC) is presented and its block diagram is illustrated. VVC's main blocks are analyzed from the **complexity** point of view. In fact, to ease the encoding process, it is essential to understand which is the system's bottleneck and act on that consequently. In this chapter, there is the explanation of why the Affine Motion Estimation has been chosen as the subject algorithm for this thesis work: because it is one of the most complex stages of the encoding chain. The chapter is concluded with the presentation of the most complex block's state-of-art and the motivation behind this work.

Initially, in the **second** chapter, the Affine Motion model and Estimation algorithm in VVC are presented. By applying some modifications and simplifications, the proposed method for Affine Motion Estimation (AME) is obtained and presented. The algorithm is divided into two sub-parts, the "construction" and the "estimation", which are described in detail. Subsequently, this simplified and approximated method is compared with the VVC's exact one. The comparison at first is done in terms of **computational complexity**: the savings in terms of elementary operations is shown. The second metric for comparison is the **compression ratio**. A good Motion Estimation algorithm, as explained in this chapter, reduces the amount of information to be transmitted (better compression of the "residual frame"). Therefore, an evaluation of the residual signal energy when using the approximated algorithm is performed. The chapter is concluded with a comparison to other works on AME. What emerges from the study is that, to the best of the author's knowledge, there are no other works on AME that simplify the algorithm in the same way as in this one.

The hardware implementation process and components are described in the **third** chapter. The architecture is implemented for ASIC 45 nm through an RTL description, using VHDL. The chapter is also organized into two parts, one dedicated to the "Constructor" component and the other one for the "Extimator" one, which are the two main sub-blocks of the proposed hardware accelerator. For both the blocks, the Datapath, Timing Diagram, and Control Unit are explained in detail. The chapter is concluded with an analysis of the components' usage percentage, to understand how much each component is exploited. What may happen is that, if this value is too low, there might be the possibility to apply some techniques to save complexity at the cost of a small increase in the delay.

After the design and implementation processes, the Design Flow requires the logical **verification** of the circuit, which is reported in the **fourth** chapter. In this thesis work, the verification step is performed using *MATLAB*, for the generation of the test sequences, and a VHDL *testbench* to read the cited inputs and feed them to the architecture. Here, the main components of the verification environment are explained, and the results of this phase are exposed. Finally, the design is concluded with the **synthesis** step, performed using the 45nm *Nangate* Open Cell Library and the *DesignWare* Library on the Synopsis' *Design Vision* software. The chapter ends with the area, timing, and power dissipation estimation. With this data, the **encoding performance** of the circuit is evaluated. According to the estimation and average computational complexity of the whole VVC encoding chain, the designed component can assist the VTM encoder in the elaboration of video streams with resolutions up to 1920x1080 at 50 frames per second.

The last chapter presents the **conclusions** about the whole study and design. Some considerations are done about the possibility to improve or modify the hardware accelerator. Moreover, other ideas about Affine Motion Estimation architectures are proposed, to experiment with innovative ways for reducing the VVC encoder complexity in future works.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The fast evolution of image and audio acquisition systems' quality over years has brought an increase in the amount of information to be stored and transmitted over time. To understand how heavy is the data flow in the context of *video* media, consider that *Cisco Systems* estimates an Internet video traffic of 187.4 Exabytes per month in 2021 [8].
This number would be even larger if it wasn't for the techniques allowing to reduce the size of video streams. They belong to a large branch of "information coding" called *video coding*.
In this context, K. Sayood in [5] presents an interesting example. To digitally represent 1 second of video without compression (using the CCIR 601 format), more than 20 megabytes are needed. This is a large amount of data considering that it is just one second. At first sight, there are mainly two problems involved in the handling of raw video materials. The first one is the **storage** requirement. Considering the previous example, to store a two hours long movie about $140GB$ of storage would be necessary, which is quite expensive considering modern user devices. The second one is the **transmission** and processing of such data loads. To elaborate and represent $20Mbps$ (*Megabytes per second*) of information, very high-speed devices would be required, consuming also an inexcusable amount of energy. If it wasn't for compression techniques, real-time services like video streaming and videoconferencing would not be possible. Since video content needs to be displayed on different devices produced by distinct companies, **video coding standards** have been developed over the decades by the most important standardization organizations, the *ITU-T* (International Telecommunication Union – Telecommunication Standardization Bureau) and *ISO* (International Organization for Standardization).

1.1 The increasing complexity of video coding standards

M. Wien and B. Bross present in [1] a brief overview about the history of Coding standards, from 1998's $H.261$ to modern $H.266$. Figure 1.1 shows all the standards by *ITU-T* (International Telecommunication Union – Telecommunication Standardization Bureau) and *ISO* (International Organization for Standardization) represented on a timeline.

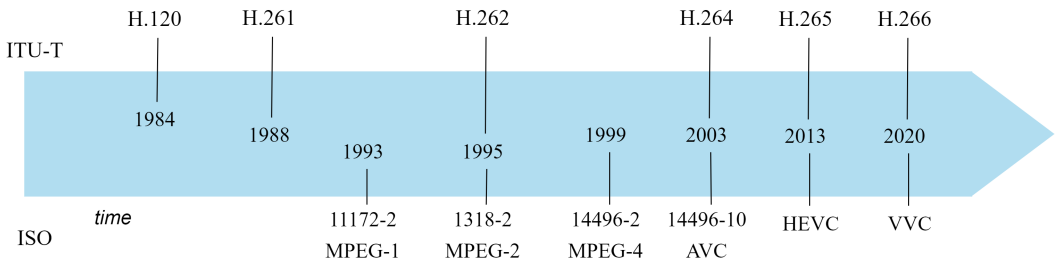

Figure 1.1. Coding standards timeline, from [1].

From 1984's H.120 to 2020's VVC, **new** features and **encoding algorithms** have been introduced. Appendix A shows the most important blocks that have been added, to improve the compression ratio of data streams. This strategy has unavoidably **increased** the standards' **complexity**, that video coding experts try to keep as low as possible over all the new generations. Nowadays, the efforts have moved towards the most recent coding standard, *Versatile Video Video Coding* (VVC).

1.2 The Versatile Video Coding standard

The Versatile Video Coding (VVC) standard was finalized in July 2020 in ITU-T as Recommendation **H.266** and ISO and IEC (International Electrotechnical Commission) as MPEG-I Part 3 [9]. The need for a more efficient coding standard than its predecessor, the HEVC, was due to the requirement for higher-quality video content from modern devices. This is what motivated the ITU-T's Video Coding Experts Group (VCEG) and ISO/IEC Moving Picture Experts Group to collaborate and form a joint group called the Joint Video Exploration Team in 2015, which will become the Joint Video Experts Team two years later, in 2017. After the Joint Call for Proposal (October 2017), the formal project for the development of the VVC standard started in April 2018. The first drafts of the specification document and the software for the VVC test model (VTM) were generated in the same month.
Currently, VVC can achieve significant bit rate reductions in the neighborhood of 50% over its predecessor for equal video quality. Unfortunately, the improved compression efficiency comes at the cost of increased complexity, as described in detail in [10].
The complexity can be both at the encoder and decoder side, but it is more present on the former since it is the least used block and the one determining the size of the data stream to be transmitted or stored.
Like all the previous video coding standards, VVC specifies only the syntax elements and the bit-stream structure, which are used in the decoding process. This means that all the different decoder implementations should be designed according to the specifications

of the standard, being able to reconstruct the same video starting from the same **VVC compliant** bit-stream. The encoder implementation choices, instead, are all left to the designers. In this way, several encoders can be designed as long as they produce bit-streams compliant with the VVC standard.

1.3 VVC Encoder Block Diagram

VVC, like most of its predecessors, relies on the hybrid video coding scheme, based on inter and intra prediction and transform coding. In figure 1.2 the block diagram of the VVC encoder is depicted.

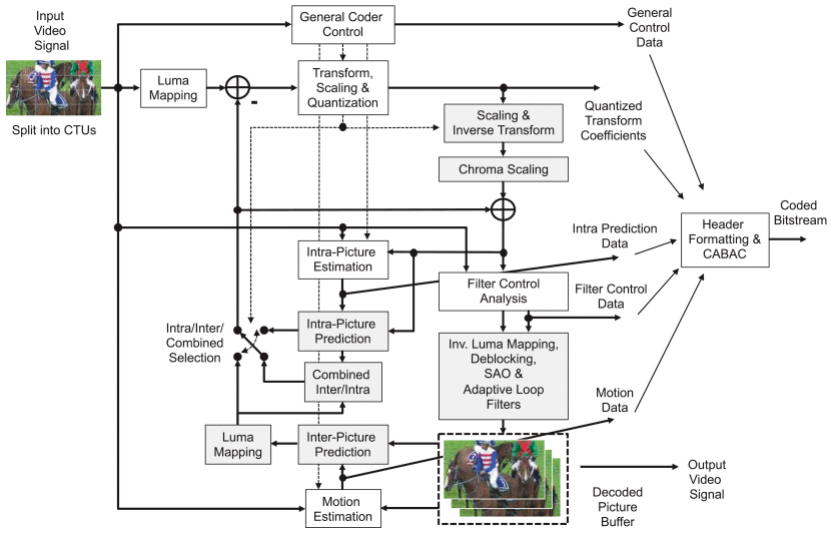

Figure 1.2. The VVC encoder block diagram.

Even though it is similar to one of the previous standards like HEVC, there are many new functionalities, briefly reported in the following. The frame to be encoded (which can be referred to as *"Current Frame"*), is split into blocks called **Coding Tree Unit** (CTU) and sub-blocks called **Coding Units** (CU). Before subtracting the reconstructed Reference Frame, it is processed with a "Luma Mapping with Chroma Scaling" (LMCS) filter, which modifies the dynamic range of the input signal using a luma inverse mapping function [9].
The residual signal is then transformed exploiting different types of DCT and DST. The residual alone is not enough for the decoder to extract the correct video stream, also the Motion information is needed. This latter is produced by reconstructing locally the encoded data.
Reconstruction happens in a few steps, which involve the inverse transform of the reference frame and the use of loop filters like the *Sample Adaptive Offset* (SAO), the LMCS, and the *Adaptive Loop Filters* (ALF). Another important step is the Motion Estimation which allows for reducing the *residual* frame size and produces the Motion Data needed by the

Encoder to perform the Motion Compensation.

1.4 VVC encoder complexity analysis

As already mentioned, the compression rate growth brought by the VVC standard comes with an increased complexity [10]. Figure 1.2 shows that the standard is made in many different stages, each one with its specific purpose and structure. To reduce the computational burden of the VVC encoder, it is essential to understand which of these blocks is more expensive in terms of complexity. In [10] a detailed analysis is reported.
Versatile Video Coding block diagram, as well as all the previous standards, is made by an encoder and a decoder. In this thesis work, the stage considered is the **encoder**. This is because the encoding process is performed a few times compared to the decoding one. Consequently, this former is always designed to be more complex than the latter. This makes the encoder computational burden reduction crucial.

1.4.1 VVC Test Model and Common Test Conditions

Before presenting the complexity breakdown of the VVC encoder, it is important to briefly introduce the two main components in the coding standard test environment. The first one is the **VVC Test Model**, free and open-source software that implements all the functionalities of the VVC encoder and decoder. It is downloadable from Fraunhofer's website [11] and it can be installed on almost any Unix-based Operating system. The main functionality of VTM is to encode and decode raw or encoded video streams with custom parameters for the coding algorithm. For example, it is possible to set the CTU minimum and maximum allowed size, disable or enable some prediction strategies, or set custom profiles and bit-width for the pixel intensity. Moreover, it allows extracting data about the output BDBR or the encoding time, which is useful to make considerations about the complexity burden of each component in the processing chain.
Like its predecessor HEVC, there are three configurations for the encoder depending on the GOP structure that the algorithm must use during its elaboration.
The first one is the **All Intra (AI)**, where all the frames are of type I. This means that no temporal redundancy is exploited. The encoding complexity and time requirements are low, but the compression rate is low too. In figure 1.3 an example of GOP in the AI configuration is shown.

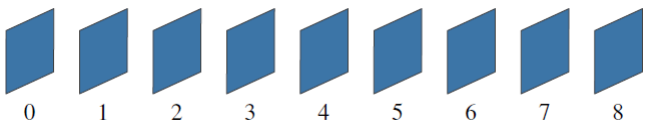

Figure 1.3. An example of GOP in the All Intra configuration. In blue, the I-type frames. From [2].

The second one is the **Low Delay (LD)** one, where only the first frame in the sequence is of the I kind. The subsequent frames are all of type P or B. There is only one prediction direction, which is "backward". This is a slightly more complex setting for the encoder but it is able to achieve higher compression rates [5]. An example of GOP is depicted in figure 1.4.

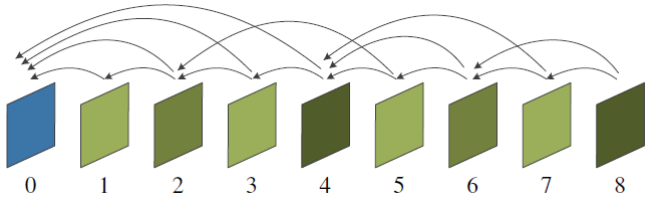

Figure 1.4. An example of GOP in the Low Delay configuration. In blue, the I-type frames while the P and B frames are colored with shaded green. P with the lightest shade, B with the darkest, depending on the temporal layer. From [2].

The last configuration is the **Random Access (RA)**, which exploits all the types of frames and prediction directions. This means that there can be I, P, and B frame types and the prediction can be performed in both forward and backward directions. This is the setting with the highest computational complexity for the encoder but also the one with the best compression rate. In figure 1.5 an example of GOP in the RA configuration is shown.

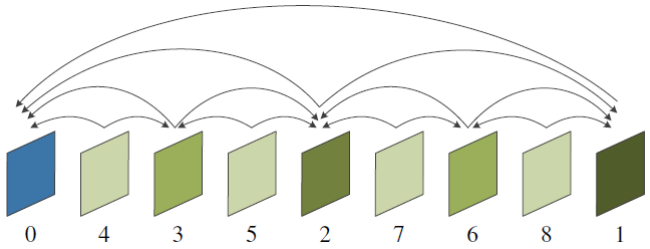

Figure 1.5. An example of GOP in the Random Access configuration. The coloring pattern is the same as figure 1.4. From [2].

When performing tests with the VTM, one of these three configurations can be chosen and the compression rate can be evaluated. Subsequently, considerations can be made about the best one depending on the specific requirements.
Since the processing results depend on the video sequence subject and the reference software configuration parameters, the JCT-VC (Joint Collaborative Team on Video Coding) has defined the **Common Test Conditions (CTC)** [12]. It is a set of directives about how the experiment must be conducted. It contains also a list of test sequences with different resolutions (ranging from 420×240 up to 4096×2160), frame rate ($20fps$ to $60fps$), and bit depth (8 or 10 bits per sample). Two of these sequences are used in this work to test the designed AME Architecture, they are cited in the next sections. In table

1.1 some of the CTC test sequences are reported.

Class	Resolution	Sequence	Frame Count	Frame Rate	Bit Depth
A1	4096 × 2160	*Tango2*	294	60	10
A1	4096 × 2160	*FoodMarket4*	300	60	10
A2	2560 × 1600	*CatRobot1*	300	60	10
A2	2560 × 1600	*ParkRunning3*	300	50	10
B	1920 × 1080	*MarketPlace*	600	60	10
B	1920 × 1080	*Cactus*	500	50	8
C	832 × 480	*BQMall*	600	60	8
C	832 × 480	*PartyScene*	500	50	8
D	416 × 240	*RaceHorses*	300	30	8
D	416 × 240	*BasketballPass*	500	50	8
E	1280 × 720	*FourPeople*	600	60	8
E	1280 × 720	*KristenAndSara*	600	60	8
F	832 × 480	*BasketballDrillText*	500	50	8
F	1024 × 768	*ChinaSpeed*	500	30	8

Table 1.1. Some of the *Common Test Conditions* video sequences

1.4.2 Encoder complexity breakdown

In [13], the complexity breakdown for the encoder of VVC Test Model 6 is reported. The tests are performed on six video sequences with resolutions of 720p, 1080p, and 2160p with the three configurations (LD, RA, AI). A first interesting result is that, depending on the settings used, the VVC encoder can be from 5× up to 31× more complex than HEVC. This remarks on how fast the computational complexity is growing in modern video coding standards and how it's essential to reduce the burden of general-purpose processors.
The test outcome is that on average with LD and RA the heaviest stage is **Motion Estimation (ME)**, taking 47% of the total complexity. It is followed by Intra Prediction (IP), Transform and Quantization (T/Q), Entropy Coding (EC), Loop Filters (LF), and Memory (Mem) operations, with 6%, 28%, 10%, 2%, 3% respectively. In the All Intra configuration, instead, the most complex stage is the **IP** stage with 29%, followed by T/Q with 44% and EC with 55%. This latter result is consistent with the former since in AI there is no forward nor backward prediction, therefore no Motion Estimation is involved. Figure 1.6 presents the complexity breakdown of the encoder with the LD and RA configurations.

Of course, to reduce the complexity burden of the VVC encoder it is essential to act on the most complex stages. With such results, the choice is to focus on the heaviest blocks and, in particular, on Motion Estimation, Transform and Quantization, Entropy Coding, and Loop Filters.

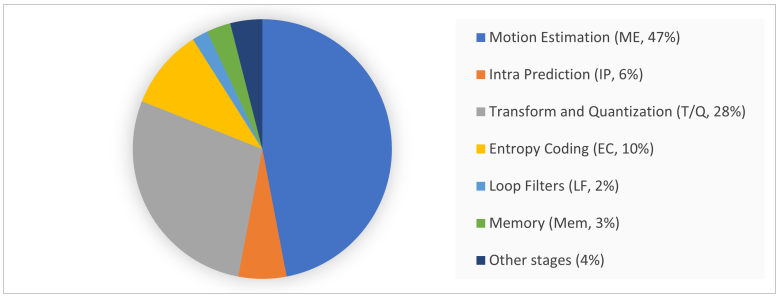

Figure 1.6. VVC encoder complexity breakdown with the LD and RA configurations .

1.5 VVC complex blocks' algorithms and architectures state-of-art

A state-of-art analysis of VVC's most complex blocks' methods and architectures is carried on to select an algorithm on which to take action to design a relative hardware component.

1.5.1 Motion Estimation

One of the most recent references about Motion Estimation in VVC is *"Motion Vector Coding and Block Merging in the Versatile Video Coding Standard"* [14], which is an **overview** on the motion vector coding and block merging techniques. In particular, it shows how the new inter-prediction methods can jointly achieve 6.2% and 4.7% BD-Rate savings on average with the Random Access and Low Delay configurations. It is helpful to understand how Motion Vector Coding and Block Merging work in VVC and which are the differences compared to HEVC.
In *"Decoder-Side Motion Vector Refinement in VVC: Algorithm and Hardware Implementation Considerations"* [15] the focus is on the description of **DMVR**, which is a method to increase the prediction accuracy of the blocks coded in merge mode using a refinement strategy. The focus here is on the hardware implementation considerations, particularly on memory bandwidth requirements and the algorithm's processing latency. There is no hardware implementation presented, all the data are extracted from simulations on the VTM 8.0. Compared to the former HEVC, this new coding tool allows achieving from 4.7% to 6.0% of BD-rate reduction in the Random Access configuration, at the cost of doubling the decoding time. On the contrary, in *"An Approximate Versatile Video Coding Fractional Interpolation Hardware"* [16] an **implementation** on Xilinx VC7VX330T-3FFG1157 **FPGA** is present. In particular, the stage taken into consideration is the fractional interpolation filter, whose equation is simplified before describing the relative hardware component using Verilog HDL. It is interesting to notice how the simplified filter occupies less area and consumes up to 40% less power consumption than the exact fractional interpolation hardware.
Beyond these documents, in literature a set of papers about **Affine Motion Estimation** is present. They are briefly presented in the next sections since they are of particular

interest.

1.5.2 Transform and Quantization

There are many documents about Transform and Quantization algorithms and architectures in the literature. This is because the Discrete Cosine Transform and the Quantizer have been always present in video coding since ITU-T H.261.
"A 2-D Multiple Transform Processor for the Versatile Video Coding Standard" [17], an FPGA implementation of three transforms is present. In particular, the considered algorithms are DCT-II, DST-VII, and DCT-VIII with a supported block size of up to 32x32 pixels. Similar work but for **ASIC TSMC 65 nm** has been done in *"A Pipelined 2D Transform Architecture Supporting Mixed Block Sizes for the VVC Standard"* [18]. Here a high-performance pipelined hardware implementation for 2D DST-VII/DCT-VIII transform operations in VVC is presented. The circuit can support clock frequencies up to 250 MHz.
The mentioned works are all done for the VVC encoder, but there is also some effort put into the decoder in *"Lightweight Hardware Implementation of VVC Transform Block for ASIC Decoder"* [19]. Here the implementation of three inverse transform algorithms, IDCT-II, IDST-VIII, and IDCT-VIII, is presented, targeting an ASIC platform. The transform block supported dimensions range from 4×4 up to 64×64. It is important to notice how the authors support what was previously said about the multitude of works already present for the DCT. Another work for FPGA is produced in *"DCT -II Transform Hardware-Based Acceleration for VVC Standard"* [20], where the focus is on the 1-D and 2-D DCT-II transform. VHDL implementation of the proposed method targets an "Arria 10AX115N3F4512SGES" running at $164MHz$.
From Politecnico di Torino comes a *"Low-Complexity Reconfigurable DCT-V Architecture"* [21]. In this work, a low-complexity and reconfigurable architecture for DCT-V of length 32 are presented. The results obtained when implementing it on 90 nm CMOS technology is that it occupies only 90k eq. gates reaching a clock frequency of 187 MHz.
The task of achieving high performance is present in *"An FPGA-Based Architecture for the Versatile Video Coding Multiple Transform Selection Core"* [22]. A deeply pipelined architecture is proposed that implements Multiple Transform Selection (MTS) for block sizes up to 64×64. It is implemented on a **System on a Programmable Chip (SoPC)** on a Cyclone V device. The system clock domain works at $200MHz$ and it can process 3840×2.160 at $64fps$ for 4×4 transform sizes.
There is also a set of works whose aim is to optimize the execution of transform algorithms [23, 24, 25, 26]. They are not presented since the T/Q stage is not the focus of this thesis work. In fact, the amount of work already present has moved the attention towards other stages of the VVC encoding chain.

1.5.3 Loop Filters

For what concerns Loop Filters, in *"VVC In-Loop Filters"* [27] a detailed overview of the different kinds of filters used in VVC is reported. There are different filters in this coding standard, each one with different purposes. The most common filters are the "deblocking"

ones, whose aim is to reduce blocking discontinuities. Even though this stage contributes only the 2% on encoding computational cost, a brief analysis has been carried on, to have a complete sight of all the VVC's most complex stages.
In *"In-Loop Filter with Dense Residual Convolutional Neural Network for VVC"* a residual **convolutional neural network** (CNN) [28] is proposed which saves computational resources and allows for reduction the encoding time. For LD, RA and AI configurations, the saving is 1.52%, 1.45%, and 1.54% respectively. The tests are performed using the DIV2K dataset produced in the NTIRE 2017 (New Trends in Image Restoration and Enhancement workshop) and the VTM-4.0.
The idea of a neural network is exploited again *"One-for-all: An Efficient Variable Convolution Neural Network for In-loop Filter of VVC"* [29], where a Variable-CNN-based in-loop filter is designed. It can handle videos compressed with different Quantization Parameters (QPs) and Frame Types (FTs) via a single model. This solution can achieve on average 6.42%, 8.08%, and 7.02% BD-rate reduction under AI, LP, and RA configurations, respectively, compared with the HEVC anchor.
Another proposal for VVC Loop Filters is the one in *"Optimized Adaptive Loop Filter in Versatile Video Coding"* [30]. More specifically, the Adaptive Loop Filter (ALF) is the subject of interest. This filter is a new feature in VVC, it is an LF capable of changing its characteristics depending on luma and chroma samples intensity. The proposal is of an optimized ALF framework which is 25% faster than the standard one, with negligible coding performance change under Random Access configuration. In conclusion, the tools for Loop Filtering currently treated in literature are the Deblocking, SAO (Sample Adaptive Offset), and ALF ones. The first two were still present in High-Efficiency Video coding, only the last one is new in VVC. Generally, there is a tendency to exploit Convolutional Neural networks to improve the compression ratio.

1.5.4 Entropy Coding

Like in section 1.5.3, for this stage, there is a complete overview in [31] which is worth mentioning. It shows how much complexity is present in VVC, which is added to improve the compression ratio. And more computational effort is added in [32], where additional **statistical dependencies** between quantization indexes are utilized. This complicates the procedure of entropy coding, increasing the encoding time by 10%, but improves the compression ratio too, with bit-rate savings of 1.5%, 1.0%, and 0.8% (AI, RA, LD).
VVC standard adopts the **trellis-coded quantization**, an algorithm which reduces the size of some DCT coefficients while recovering others to take their place. Like many other stages in the processing chain, it allows high compression efficiency, but also high complexity and low throughput capacity. In *"Low Complexity Trellis-Coded Quantization in Versatile Video Coding"* [33] a low complexity trellis-coded quantization scheme is proposed. There is actually no hardware implementation of the method, only simulations on the VVC Test Model. Results show that the proposed scheme achieves 24% and 27% quantization time savings with All Intra and Random Access configurations, respectively. As predictable, lower computational effort leads to higher memory or bandwidth requirements, in fact, there is a 0.11% and 0.05% BD-Rate increase.

In conclusion, there are a few hardware implementations of VVC's most complex stages, except for the Transform algorithm. Considering all the analyzed works, what happens when acting on the processing chain is that the compression ratio and encoding time or computational complexity, change in the opposite direction. **Adding features** to the algorithm typically **reduces** the **bit-rate**, but complicates the encoding process, like in [15, 29, 32]. On the contrary **simplifying the methods** used, results in **reducing** the **encoding time** but at the cost of higher bandwidth and memory requirements. This happens in [16, 28] and it is the direction taken in this thesis work.

The reason behind this choice is that VVC is already able to achieve approximately 50% of bit rate savings against its predecessor, but with up to 31× the computational complexity [13]. Consequently, the consideration made up is that the real need is to reduce this high complexity rather than increase an already good compression rate. After this brief study on the heaviest VVC encoding components, the choice has been to focus on **Motion Estimation**, since it is the one with higher computational complexity and because it includes a vast family of algorithms on which it is possible to act. In particular, the considered one is *Affine Motion Estimation*.

Chapter 2

Affine Motion Estimation simplified algorithm

2.1 The Affine Motion Model

Pakmadan *et al.* [13] report that, among the sub-parts of Motion Estimation (ME), **affine search** is the most important one in terms of computational requirements. As already mentioned in section A.2.1, ME exploits temporal redundancy by storing or transmitting information about elements' movement in an already decoded frame to reconstruct another frame. For the encoder to recognize these movements it has to search for matching blocks in the current ("to-be-encoded") and reference frames. The searching and consequently the reconstruction algorithms were based on the translation of pixel blocks until HEVC and the early version of VVC. In this case the method is referred to as **Translational** Motion Estimation (TME). It is quite simple in terms of implementation and computational effort but does not allow the identification of complex motions. What happens in practice is that, if objects in a picture perform complex movements like distortion, rotation or shearing, TME will not be able to identify them. Figure 2.4 shows an example of affine motion made by a combination of zoom and shearing.

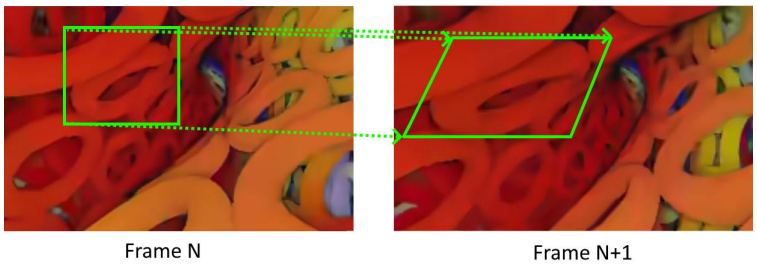

Figure 2.1. An example of pixel block affine motion, made by a combination of zoom and shearing. Video source from "VQ Analyzer" sample streams [3].

If the translational motion model does not include such complex movements, the prediction error increases and so does the bit rate too, thus more sophisticated models have been searched through the years to handle non-translational motions.
Choi *et al.* in [4] present a complete history of how these models changed through time until an Affine Motion model was finally introduced in Versatile Video Coding. They report that at the 11th JVET (ITU's *Joint Video Experts Team*) meeting in July 2018, an AME model was integrated into VTM2 based on [34].

2.2 Affine Motion model in VVC

The Joint Video Expert Team (JVET), which is a partnership of ITU-T Study Group 16 Question 6 (known as VCEG) and ISO/IEC JTC 1/SC 29/WG 11 (known as MPEG), established the Versatile Video Coding (VVC) draft 8 and the VVC Test Model 8 (VTM8) algorithm description and encoding method at its 17th meeting (7–17 January 2020, Brussels, BE) [35]. In this document, a description of the Affine Motion model currently in use in VVC is detailed.
In HEVC only one motion vector is used and that is sufficient since it identifies translational motion only. In the VVC Affine Motion model instead, the motion field of a pixel block is described by motion vectors relative to some specific points called **Control Points (CPs)**. The number of CPs determines the predictable movement complexity. Figure 2.2 shows the motion models types: in HEVC (a); in VVC with two (b) and six (c) control points. When two CPs are used, the model is called a **four-parameter model**, otherwise it's a **six-parameter** one. The parameters are simply the coordinates of the Control Point Motion Vectors (**CPMPs**). Since a Motion Vector mv has two coordinates (mv_h, mv_v) (horizontal ad vertical), the number of parameters in a model is simply double the CPs used.

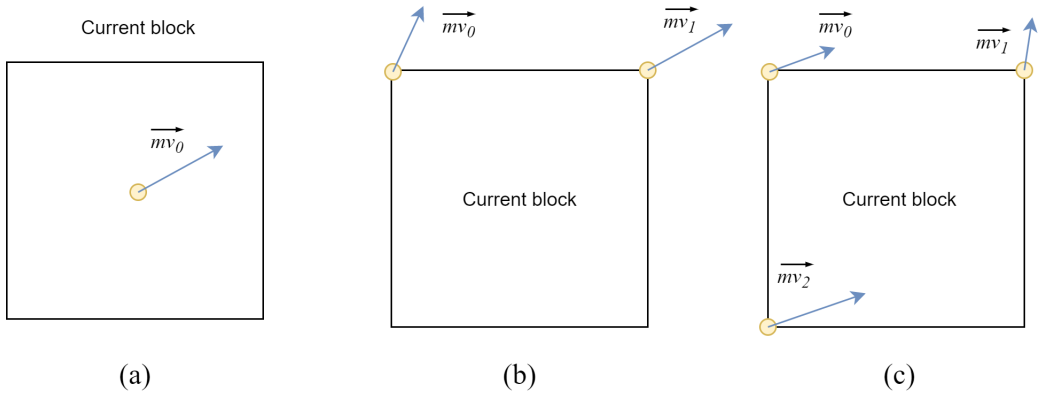

Figure 2.2. Three different motion models types: HEVC Translational Motion Model (TMC) (a); VVC Affine Motion Model with two (b) and six (c) control points.

In VVC, CMPVs for each Coding Unit are derived by the encoder with the Affine Motion Estimation algorithm. Then, they are sent by the decoder which exploits them to

reconstruct the encoded frame using the Affine Motion Compensation (**AMC**) algorithm. Before detailing AME it is essential to present how AMC works. Compensation is not performed at pixel level using the CPMVs, otherwise, the extremely high complexity burden would be unaffordable in practice [34]. Actually, a *sub-block level* motion compensation is employed, where the sub-block dimension is 4×4.
Let mv_0, mv_1, mv_2 be the CPMVs of a CU with height h and width w. The Motion vector $mv = (mv^h, mv^v)$ for a 4×4 sub-block with coordinates (x, y) can be calculated as

$$\begin{cases} mv^h(x,y) = \frac{mv_1^h - mv_0^h}{w}x - \frac{mv_1^v - mv_0^v}{w}y + mv_0^h \\ mv^v(x,y) = \frac{mv_1^v - mv_0^v}{w}x + \frac{mv_1^h - mv_0^h}{w}y + mv_0^v \end{cases} \tag{2.1}$$

when the 4-parameter model is used, while

$$\begin{cases} mv^h(x,y) = \frac{mv_1^h - mv_0^h}{w}x + \frac{mv_2^h - mv_0^h}{h}y + mv_0^h \\ mv^v(x,y) = \frac{mv_1^v - mv_0^v}{w}x + \frac{mv_2^v - mv_0^v}{h}y + mv_0^v \end{cases} \tag{2.2}$$

is applied with the 6-parameter model. In VVC there are two Affine Motion Estimation modes: *AMVP (Advanced Motion Vector Prediction) mode* and *merge mode*. In the first one, an algorithm is designed to predict the MVs of Coding Units. In the second one, the MVs in a CU are derived from the neighboring CU, therefore there are no complex calculations to be performed, the VTM must only look at the neighboring CUs and copy some of their motion vectors depending on the Coding Units are coded. For the affine **AMVP mode** instead, the prediction is generated starting from two triples of MV **candidates** with which Affine Motion Compensation is applied to understand which one could be the best for the motion prediction. The algorithm and so the hardware accelerator has been designed to perform Affine Motion Estimation in the AMVP mode, whose algorithm is described in section

2.2.1 Affine AMVP prediction

Affine AMVP mode can be applied on CUs with both width and height larger than or equal to 16. The upper limit on the CU dimension depends on the VTM settings and can reach up to the maximum allowed Coding Unit size, which is 128×128. The motion model exploited in each case can have 4 or 6 parameters. The prediction is generated starting from triplets (or couples, depending on the number of parameters), of MVs **candidates** called CPMVP (Control Point Motion Vector Predictors). The number of CPMVPs in the candidates list is two and they are selected among four types of Control Point Motion Vectors:

1. **Inherited** affine AMVP candidates. They are derived from the affine motion model of the neighboring Coding units. Figure 2.3 shows the considered blocks. In total there are two possible inherited predictors, one selected from group $A : \{A1, A0\}$, and one from $B = \{B1, B0, B2\}$. The candidate **selection** among each group is done simply by taking the **first available CPMVP** and discarding the other ones, following list order. When a neighboring affine CU is identified, its control point

motion vectors are used to derive the CPMVP (Control Point Motion Vector Predictor) candidate in the candidates list of the current CU.
If from the first selection, not enough MVs are found, the block A0 is checked. If it is coded in affine mode, the CPMVs of the CU which contains the block are attained and set as (v_1, v_2, v_3) (if A0 exploits the 4-par model, the last vector is absent). When block A0 is coded with the 4-parameter affine model, the last CPMV of the current CU is calculated according to v_2, and v_3. In case that block A is coded with 6-parameter affine model, it is calculated according to v_2 , v_3 and v_4.

2. **Constructed** affine AMVP candidates CPMVPs that are derived using the translational MVs of the neighbor Coding Units. With reference to figure 2.3, if any of the neighboring 4×4 sub-blocks belongs to a CU that is inter-coded, their MV is indicated as MV_X, where X is the block name. Three sets are formed, $S_0 = \{B2, B3, A2\}$, $S_1 = \{B1, B0\}$, $S2 = \{A1, A0\}$. The two CPMVPs are the first and second triplet of candidates picked from $S0$, $S1$, and $S2$ groups' MV_X, following list order. If the CU is coded with the 4-parameter model, $S2$ is discarded.

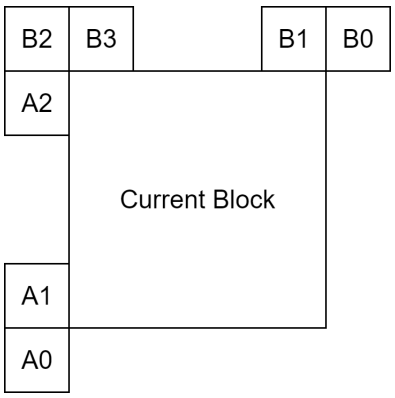

Figure 2.3. CU neighboring blocks in a frame.

3. **Translational** MVs from neighboring CUs. They are collected if affine AMVP list candidates are still less than two after Inherited affine AMVP candidates and Constructed AMVP candidates are checked. When available, mv_0, mv_1, mv_2 are added, in order, as translational MVs to predict all control point MVs of the current CU.

4. **Zero** MVs: if the affine AMVP list is still not full, zero MVs are used to fill it.

The listing order is important since candidates of higher types in the list, if available, have priority on the lower types. When the list is full of candidates of the higher kind, no more MVs are considered. At this moment, Motion Compensation is applied to each one of the two candidates and the Sum of Absolute Transformed Differences (SATD) is calculated. The candidate providing the minimum SATD cost is chosen as the "best" one and set as $\{mv_0^{(0)}, mv_1^0\}$.

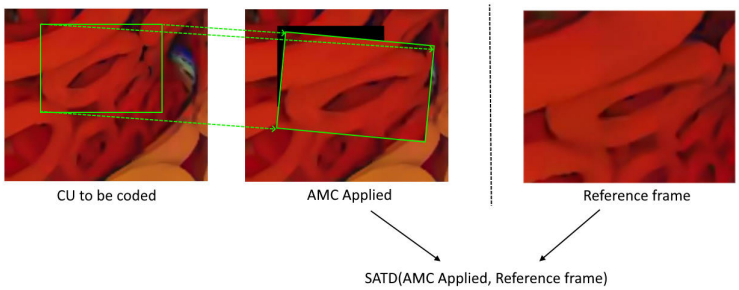

Figure 2.4. What happens in the VVC encoder when choosing the best candidate. The process in figure is repeated for each CPMVP and the the one with the lowest SAD is chosen. Video source from "VQ Analyzer" sample streams [3].

Subsequently, an **iterative algorithm** is applied [34]. In each *i-th* step, (starting with $i = 0$), a new $\{mv_0^{(i+1)}, mv_1^{(i+1)}\}$ is computed, which replaces $\{mv_0^{(i)}, mv_1^{(i)}\}$ if it provides a lower SATD cost. This refinement loop il repeated N times in the worst case, with N adjustable in the VTM settings. It consists in many steps which require to:

- Apply the **Sobel** operator on the CU to be encoded. That means filtering a 128×128 pixel block by multiplying it with a 128×128 square matrix, in the worst case.
- Solve a linear system of M equations, with M equal to the number of model parameters. In the worst case, with 6-parameter model, the solution of a 6 equations linear system must be computed.
- Performing AMC on $\{mv_0^{(i+1)}, mv_1^{(i+1)}\}$ and compute the SATD cost to decide if these CPMVP constitute a better estimation than $\{mv_0^{(i)}, mv_1^{(i)}\}$.

This last iterative algorithm has high computational complexity compared to the previous ones. Consider that the best-candidate choice requires just one AMC while the refinement process performs it N times. Each iteration requires the calculation of tens of motion vector using 2.1 or 2.2 and then performing $128 \cdot 128 = 16384$ memory accesses just for the AMC. Finally, 16384 differences and a 128×128 matrix multiplication must be performed for the SATD [36].
In this thesis work a simplified algorithm is proposed compared to the one currently present in the VTM.

2.3 The proposed algorithm

The idea of a simplified algorithm starts considering the possibility of **leaving out** the **iterative refinement** process described in 2.2.1 while making the candidate **construction** method more sophisticated. In addition, the AMC process is simplified too, to make also the best-candidate choice more lightweight in terms of complexity.
Like the exact AMPVP algorithm, the simplified one can be applied on CUs with both

width and height larger than or equal to 16, but no greater than 64. This choice has been done in order to simplify the hardware architecture. Also here both 4 and 6 parameter models are supported, and prediction is generated starting from two MVs **candidates** called CPMVP (Control Point Motion Vector Predictors).
The list is built in the same way as in the exact VTM method (section 2.2.1), the only change is the way of constructing the candidates.

2.3.1 Proposed candidate construction method

The proposed method is a slight modification of the one in [34]. It starts considering the neighboring 4×4 blocks of the CU to be encoded, as shown in figure 2.5, together with their respective Motion Vectors. Indicating the constructed candidate MV triplet as $\{mvp_0, mvp_1, mvp_2\}$, the algorithm works as below:

1. With reference to figure 2.5, collect motion vectors in three groups:
   $S_0 = \{MV_A, MV_B, MV_C\}$, $S_1 = \{MV_D, MV_E\}$ and $S_2 = \{MV_F, MV_G\}$, where MV_X is the MV from block X. If one neighboring block is not inter coded, its MV is set as unavailable and therefore discarded.

2. $\forall\ (MV_{S0}, MV_{S1}, MV_{S2})$, with $MV_y \in y$, derive the **distrortion**

$$D^2(MV_{S0}, MV_{S1}, MV_{S2}) = |MV_{S2} - MV_p|^2 \tag{2.3}$$

   MV_p here is computed from an approximated version of 2.1 with $mv_0 = MV_{S0}$, $mv_1 = MV_{S1}$ and $(x, y) = (0, h)$:

$$\begin{cases} MV_p^h = -\lfloor \frac{(MV_{S1}^v - MV_{S0}^v)}{w} h \rfloor + MV_{S0}^h \\ MV_p^v = +\lfloor \frac{(MV_{S1}^h - MV_{S0}^h)}{w} h \rfloor + MV_{S0}^v \end{cases} \tag{2.4}$$

3. Set $\{mvp_0, mvp_1, mvp_2\}$ as the $(MV_{S0}, MV_{S1}, MV_{S2})$ triplet with the lowest distortion value.

In [34], which presents a similar algorithm, the distortion is defined as

$$D(MV_{S0}, MV_{S1}, MV_{S2}) = |MV_{S2} - MV_p| \tag{2.5}$$

Since

$$D = |MV_{S2} - MV_p| = \sqrt{(MV_{S2}^h - MV_p^h)^2 + (MV_{S2}^v - MV_p^v)^2} = \sqrt{D^2} \tag{2.6}$$

considering the square value of D allows saving the computation of a square root.

In Equation 2.4, the use of the $floor(x) = \lfloor x \rfloor$ function is made to model the **truncation** rounding method applied, which saves two bit in the number representation, since the minimum value of $\frac{h}{w}$ is

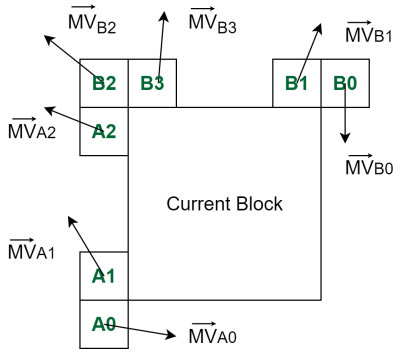

Figure 2.5. CU tu be encoded with its neighboring blocks and relative motion vectors.

$$\min\left(\frac{h}{w}\right) = \frac{h_{min}}{w_{max}} = \frac{16}{64} = \frac{1}{4} \tag{2.7}$$

This means that the maximum number of shifts towards the right is 2. The **rounding bias** is the average error when discarding d digits after a rounding operation. It measures the tendency of a round-off scheme towards errors of a particular sign. For truncation, with $d = 2$, the Rounding Bias is $-3/8$, as shown in table 2.1, where X is the integer part of any bit length. This value becomes a problem when the output of the rounding is the input of a feedback chain. Since this is not the case, this Rounding Bias value is considered satisfactory.

Number	$\lfloor x \rfloor$	Error
$X.00$	X	0
$X.01$	X	$-1/4$
$X.10$	X	$-1/2$
$X.11$	X	$-3/4$

Table 2.1. Error in the truncation rounding method

2.3.2 Best-candidate choice simplified algorithm

The exact VTM best-candidate choice algorithm requires performing AMC of the current CU to be encoded and compute the SATD with the reference frame. To perform Motion Compensation, **all the MVs** of each 4×4 block must be computed. In the worst case, with a 128×128 Coding Unit, $(128 \times 128)/(4 \times 4) = 1024$ MVs must be computed using equation 2.1 or 2.2. Moreover, affine movements include rotation and shearing, therefore to resolve pixel overlapping, **6-tap filtering** is required [9].
The proposed simplified algorithm **reduces** the number of Motion Vectors to be computed

and **saves** computational complexity by avoiding the use of the 6-tap filter. The algorithm works as follows

1. Divide the Current CU to be encoded in 16×16 sub-blocks.
2. For each 16×16 sub-block, collect their upper-left, upper-right, lower-left, lower-right 4×4 sub-blocks and label them as the **representative** blocks for the 16×16 sub-block. Figure 2.6 shows an example of 32×32 CU split into four 16×16 sub-blocks (in blue), each one with its four representatives (in red).
3. Perform **Transational** Motion Compensation (TMC) on each representative block, computing the representative's Motion Vectors using the approximated version of the Affine Model Equations (2.1 and 2.2) shown in 2.8.

$$\begin{cases} mv^h(x,y) = a_2 x + b_2 y + mv_0^h \\ mv^v(x,y) = a_1 x + b_1 y + mv_0^v \end{cases} \tag{2.8}$$

with

$$a1 = \lfloor \frac{mv_1^v - mv_0^v}{w} 2^4 \rfloor \frac{1}{2^4} \qquad a2 = \lfloor \frac{mv_1^h - mv_0^h}{w} 2^4 \rfloor \frac{1}{2^4} \tag{2.9}$$

$$b_1 = \begin{cases} \lfloor \frac{mv_1^h - mv_0^h}{w} 2^4 \rfloor \frac{1}{2^4} & if\ sixPar = 0 \\ \lfloor \frac{mv_2^v - mv_0^v}{h} 2^4 \rfloor \frac{1}{2^4} & if\ sixPar = 1 \end{cases} \quad b_2 = \begin{cases} -\lfloor \frac{mv_1^v - mv_0^v}{w} 2^4 \rfloor \frac{1}{2^4} & if\ sixPar = 0 \\ +\lfloor \frac{mv_2^h - mv_0^h}{h} 2^4 \rfloor \frac{1}{2^4} & if\ sixPar = 1 \end{cases} \tag{2.10}$$

where

- (x, y) are the representative block's top-left pixel horizontal and vertical coordinates.
- $mv_0 = \{mv_0^h; mv_0^v\}$, $mv_1 = \{mv_1^h; mv_1^v\}$, $mv_2\{mv_2^h; mv_2^v\}$ is the MV triplet of the current MV candidate for the Motion Estimation.
- h and w are the height and with of the Current CU, respectively.
- $sixPar$ is a flag indicating whether the motion model used has four ($sixPar = 0$) or six ($sixPar = 1$) parameters.

The multiplication and division by a factor 2^4 take into account the intention of discarding two LSBs in the binary expression of $(a_{1,2}, b_{1,2})$ while keeping four fractional bits. This allows saving two bits in some components in the hardware architecture while still keeping a low Rounding Bias. Since these bits are placed after four fractional bits, the Rounding Bias here is 2^{-4} times the one computed in Section 2.3.1. Figure 2.7 shows how TMC is applied on the example CU depicted in figure 2.6. In red, the representative blocks of the CU to be encoded, in green their relative blocks in the reference frame, in the position calculated applying TMC.

4. Compute the SAD between the 4×4 motion-compensated representative blocks and their relative reference frame blocks. The candidate associated with the **lowest** SAD value is the output of the motion estimation.

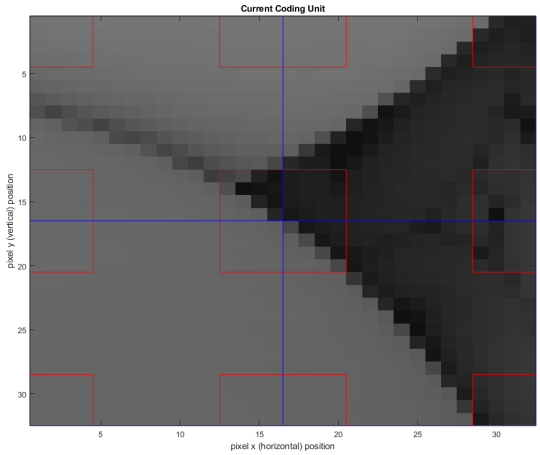

Figure 2.6. an example of 32×32 CU split into four 16×16 sub-blocks (in blue), each one with its four representatives (in red).

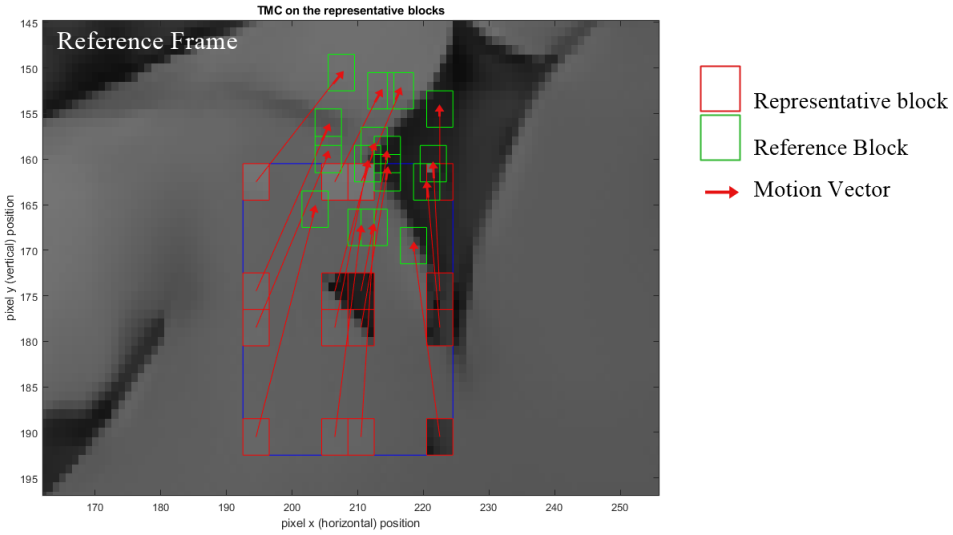

Figure 2.7. How TMC is applied on an example CU.

2.4 Comparison between the proposed algorithm and the exact one

2.4.1 Computational complexity

The approximations introduced in the simplified algorithm allow to save computational resources. Since Motion Compensation is not applied on the whole CU but the representative blocks only, the number of MVs to be computed is **reduced** by a factor of 4. The same holds for the SAD algorithm: the number of subtractions is reduced by the same factor. Moreover, since the SAD is computed instead of the SATD, the Hadamard transform is **no** longer **required**. Finally, since the TMC is applied instead of AMC, the 6-tap filter is avoided. Table 2.2 is a summary of the cited advantages.

Requirements for a $N \times N$ CU	VTM exact AME	Proposed Algorithm
MV to be computed	$\frac{N^2}{16}$	$\frac{N^2}{16} \cdot \frac{1}{4}$
Subtractions required	N^2	$N^2 \cdot \frac{1}{4}$
6-tap filter required	Yes	No
$N \times N$ Hadamard transform required	Yes	No

Table 2.2. Complexity savings of the Approximated AME algorithm

Another advantage is that this algorithm can be implemented in hardware without using complex components. In fact, the operations involved include additions, products, and subtractions only. There are no complex functions or division units required. Since the divisors are all powers of two, the calculations are reduced to **shift operations** towards the left or right.
When the algorithm is implemented as a hardware accelerator, the computational load of the processor running the VTM software is reduced, being all the AME operations handled by the former, the AME contribution to its computational effort is almost zero.

2.4.2 Compression Ratio

To compare the VTM exact AME to the proposed method in terms of compression ratio, this latter is implemented in software using *MATLAB* tool. The construction algorithm script is reported in (B.1) while the Affine Motion Estimation one is in (B.2). To perform the tests under the Common Test Conditions, described in section 1.4.1, two video streams have been coded using the VVC Test Model 8.0.

The installation files of the VTM-8.0 can be downloaded from *VVCSoftware_VTM*'s GitLab repository[1]. After cloning it on a Linux system, having *make* tool and Python already present on it, the following script is enough to complete the installation.

[1] `https://vcgit.hhi.fraunhofer.de/jvet/VVCSoftware_VTM/-/tree/VTM-8.0/`

```
JOBS_NUMBER=4 #Jobs number
mkdir build
#Generate Linux Release Makefile:
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
#Generate Linux Debug Makefile:
cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug
#Start the installation
make -j JOBS_NUMBER
```

For the last command, it is important to set the number of jobs that *make* uses for the installation. Otherwise, the process would not limit that: this could result in some machines in the procedure failure. This is done by setting the `JOBS_NUMBER` variable to an appropriate value. For the Linux system used for this thesis work (a *Lubuntu* 64-bit distribution), four jobs are sufficient.
After the installation, the VTM encoder can be used to issue the `./EncoderAppStaticd` command. The input and output files are *YUV* files. They contain non-encoded video streams where each frame is represented explicitly and each pixel is associated with a triplet of integers representing the luma and chroma components. Assuming an input video stream named `RAW_INPUT.yuv` at a framerate `FRAMERATE` and resolution `FR_W`×`FR_H`, the command to encode a number `FRAME_NUMBER` of frames is the following.

```
./EncoderAppStaticd -c CONFIG_FILE.cfg -i RAW_INPUT.yuv \
-o CODED_OUT.yuv -wdt FR_W -hgt FR_H -fr FRAMERATE -f FRAME_NUMBER
```

The VTM encoder allows to generate two internally reconstructed files named `CODED_OUT.yuv` and a `CODED_OUT.bin`. The second one in particular is essential to analyze the **encoding information**, like the type of each frame or, most importantly for this thesis work, when Affine Motion Estimation is used and which are the Motion Vectors involved in each case. Notice that `./EncoderAppStaticd` requires a `CONFIG_FILE.cfg`, like the one in Report 2.1. It is essential to configure the encoding parameters, like

- `MaxCUWidth, MaxCUHeight`: They are the maximum allowed values for the Coding Unit Width and Height. They are set to 64 since the maximum allowed CU size in the proposed algorithm is 64×64.
- `InternalBitDepth`: Is the codec operating bit depth of both the input and the internally reconstructed file. It is set to 8 since the test video sequences used are on 8 bits. This means that the (Y,U,V) components of the *.yuv* files can assume values in the range [0,255].
- `Affine`: Specifies if Affine Motion Estimation is allowed. Since the whole study is based on that kind of ME, it is important to enabling this flag by setting it to 1.

Report 2.1. `CONFIG_FILE.cfg` example

```
#======== Unit definition ========
MaxCUWidth                    : 64          # Maximum coding unit width in pixel
MaxCUHeight                   : 64          # Maximum coding unit height in pixel

#======== Coding Structure ========
IntraPeriod                   : -1          # Period of I-Frame ( -1 = only first)
DecodingRefreshType           : 0           # Random Accesss 0:none, 1:CRA, 2:IDR
GOPSize                       : 8           # GOP Size (number of B slice = GOPSize-1)

#======== Quantization ========
QP                            : 32          # Quantization parameter(0-51)
MaxDeltaQP                    : 0           # CU-based multi-QP optimization

#========== Misc. ==========
InternalBitDepth              : 8       # codec operating bit-depth

#========== NEXT ==========

# General
CTUSize                       : 128
LCTUFast                      : 1
Affine                        : 1
SbTMVP                        : 1
IBC                           : 0       # turned off in CTC
AffineAmvr                    : 0
LMCSEnable                    : 1       # LMCS: 0: disable, 1:enable


FastMrg                       : 1
AMaxBT                        : 1
FastMIP                       : 0
FastLocalDualTreeMode         : 2

# Encoder optimization tools
AffineAmvrEncOpt              : 0
MmvdDisNum                    : 6
ALFAllowPredefinedFilters     : 1
ALFStrengthTargetLuma         : 1.0
ALFStrengthTargetChroma       : 1.0
CCALFStrengthTarget           : 1.0
### DO NOT ADD ANYTHING BELOW THIS LINE ###
### DO NOT DELETE THE EMPTY LINE BELOW ###
```

Exploiting the VTM reference software, two example streams from the Common Test Conditions [12] have been coded. Moreover, an additional test video sequence has been used for the proposed algorithm performance estimation. It is a video stream taken from ViCueSoft's "VQ Analyzer" sample streams [3]. This latter is considered relevant when studying Affine Motion Estimation since it presents a considerable amount of objects moving in different ways: rotation, distortion, and zooming. Table 2.3 summarizes the three test sequences used for estimating the proposed algorithm performance.

The encoded *.bin* files are given as input to VQ Analyzer. Figure 2.8, shows the software's GUI (Graphical User Interface) and an example of a dialog screen showing motion information. This functionality is exploited to extract MVs candidates of **thirty** different Motion Estimation cases, which are the **test cases** used for the algorithm performance estimation (and later as test sequences for the hardware implementation logical verification). In table C.1 the main examples identifiers are shown for each test case. In particular, the

Class	Resolution	Sequence	Frame Count	Frame Rate	Bit Depth
D	416×240	*RaceHorses*	300	30	8
D	416×240	*BasketballPass*	500	50	8
-	432×240	*VQ_sample*	50	50	8

Table 2.3. The video test sequences used.

reported fields are:

- **Ex. No.**: "Example Number" is a unique identifier for each test case. Numbers start from 3 since examples (0,1,2) were initial tests for the constructor only. They have been discarded.
- **Stream**: It addresses the particular test sequence (among the ones in table 2.3) from which the Current (to-be-encoded) and Reference frames have been extracted.
- **POC**: "Picture Order Count", is the POC of the Current frame in the given Stream.
- (x_0, y_0) are the coordinates of the Current CU (the Coding Unit considered) in the Current frame.
- **Par**: The number of model parameters in the example
- (CU_w, CU_h): Current CU width and height, respectively.

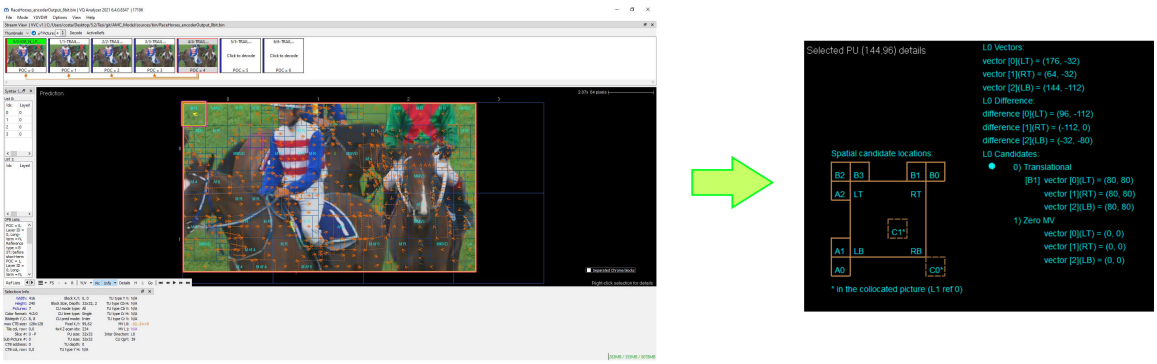

Figure 2.8. VQ Aalyzer software. On the left, the *"Prediction Mode"* view. On the right, some extracted motion information.

Using the test cases with all the Motion Vector candidates on the Matlab Model, some data about the Construction and Estimation algorithms presented in sections 2.3.1 and 2.3.2 have been gathered. At first, some considerations are done about the proposed **candidate construction** algorithm. In particular, table C.2 is reporting:

- **VQ Comply** : This flag indicates if, for the given example, the motion estimation output from the VTM is the same as the approximated algorithm, considering that for this latter the constructed candidates are given from the proposed construction method. In the last row, **"Comply Ratio"** informs that the AME output is matched in the **73%** of the total tests. What happens when the matching does not occur is discussed subsequently.

- **"Any Constructed"**: The value of this field is "V" when at least one of the candidates MV must be computed using the constructor. If the construction algorithm is not required, for example, because all the candidates are "Inherited affine AMVP" ones, "Any Constructed" is set to "X". In the last row, **"Constructor usage"** is computed as 0.7. This means that the constructor algorithm is used in 70% of the cases. This high value shows how much the constructor algorithm is used when performing Affine Motion Estimation.

- **Constr. Match**: "Construction Match" is "V" if the Motion Vectors constructed by the exact VTM are equal to the proposed constructor output, otherwise it's "X". In the last row, **"Cons. Match Ratio"** indicates that in **0.35%** of the tested cases this situation happens. This mismatch in the construction is reasonable considering that the two methods are different.

- **"VQ C. w/o constr."**: "VQ Compliance without constructor" column brings some considerations about the effectiveness of the proposed constructed method. This flag is "1" when, for the given example, the motion estimation output from the VTM is the same as the approximated algorithm, considering that for this latter the constructed candidates are given from the **exact VTM** constructor. When this happens, the "Comply Ratio" rises from 0.73 to 0.80. In short, the proposed constructor makes the approximated estimation algorithm diverge from the exact one. Whether this can bring benefit to the encoding performance or not, depends on the achieved **compression ratio** when this constructor is enabled. This is discussed subsequently.

After these considerations about the constructor behavior, a **metric** for the estimation of the compression ratio achieved with the algorithm is searched. What is done in other works about Affine Motion Estimation [34, 4, 37, 38] is to integrate the proposed procedures in the VTM software directly acting on its source code. After that, the **Bjontegaard Delta Bitrate** (BD-BR) [39] is calculated. This is a commonly used metric to measure the compression gain when some modifications are brought to an encoding algorithm. This would be feasible since VTM is an Open Source software, but little documentation is available about it. This has made impossible the most common option.

The metric used in this thesis work is a **signal energy ratio**: it is computed as follows. Let RCU_1, RCU_2 be the residual blocks for each example from table C.1, when Affine Motion Compensation is done using the MVs from the approximated algorithm and the exact one, respectively. If $N_{dct,1}$ and $N_{dct,2}$ are the number of DCT-II coefficients to store 99% of RCU_1, RCU_2 signal energy, the energy ratio is calculated as

$$E_r = \frac{N_{dct,1}}{N_{dct,2}} \tag{2.11}$$

If $E_r > 1$, using the approximated algorithm increases the energy content of the residual signal, thus reducing the compression ratio. Otherwise, the amount of information to be transmitted is lowered.
Moreover, an additional parameter has been evaluated. Let C_1, C_2 be the number RCU_1, RCU_2 pixels whose intensity if different from zero respectively. The considered parameter can be computed as

$$C_r = \frac{C_1}{C_2} \tag{2.12}$$

If $C_r < 1$ it means that the affine motion-compensated CU is closer to its corresponding block in the reference frame when the approximated algorithm is used.
Both E_r and C_r are evaluated using scripts (B.2,C.1,C.2) reported in the appendix. Table C.3 summarizes the test results on the 30 cases reported in Table C.1. Besides the already explained fields, there is the **Parameters** one which reports the models used for the specific example. "6 percentage" indicates that "50%" of the tested cases exploit the 4-parameter model, while the other one uses the 6-parameter one. This is intentional, to have a balanced variety of example situations. The field "**No Compliance Reason**" reports the reasons why there is no matching between the motion estimation output from the VTM and the approximated algorithm. They are summed up in short identifiers, whose meaning is the following:

- **Constructed Candidate Mismatch (CCM)**: the estimation algorithm correctly chooses the Constructed candidate, but this is not equal to the one constructed by the exact algorithm.
- **Wrong choice (X instead of Y)**: The constructed candidate, if present, is equal to the one obtained from the exact algorithm. Nevertheless, the proposed estimator chooses the candidate of type X, instead of Y. Candidates can be of type I (Inherited), C (Constructed), or T (Translational).
- **CCM + Wr. Ch. (X instead of Y)**: it is the same as the previous case, but this identifier informs also that if there was no Constructed candidate mismatch the proposed estimation algorithm would have complied with the exact one.

These additional notes could be useful to tune the algorithm in an eventually modified version, to improve the E_r value.
Two important indicators from Table C.3 is $E_{r,AVE}$ and $C_{r,AVE}$, which are the **average** values of E_r and C_r **among** all the tested cases. The measured values are $E_{r,AVE} \simeq 1.02$ and $C_{r,AVE} \simeq 0.99$. This means that the average energy of the residual signal is slightly increased by 2%, while the Motion Compensated CUs are almost unchanged, on average, when using the approximated algorithm instead of the exact one.
Since the bit-rate depends on the DCT of the residual signal, the main parameter to be taken into account to estimate the compression ratio of the algorithm is the energy ratio

E_r. The measured value is greater than one: this means that the **compression ratio** is slightly **increased** when using the approximation algorithm. This can be considered the computational complexity reduction (described in Section 2.4.1) drawback. As already mentioned and confirmed by other works on VVC analyzed in section 1.5, the compression ratio and the computational complexity, change in the opposite direction. In this case, **simplifying the methods** used, results in **reducing** the complexity but at the cost of higher bandwidth and memory requirements. Whether this proposed method can be adopted in an encoding system or not, depends on the target requirements.

2.5 Comparison with other works on AME

2.5.1 Other works on Affine Motion Estimation

A thorough search of the relevant literature yielded other articles about affine motion estimation which propose modifications on parts of the AME algorithm, in some cases in a way similar to what happens in this thesis work.
In *"An Improved Framework of Affine Motion Estimation in Video Coding"* [34], several modifications to the standard AME are proposed.

1. Some changes to the Affine Inter Mode algorithm are proposed. These modifications make the method similar to the Affine Merge mode, reducing the computational complexity.

2. The authors show with some example cases that the CPMVs (Control Point Motion Vectors) tend to be similar to each other. Relying on this fact, they propose to transmit CPMVs **differences** (MV_0,$MV_0 - MV_1$,$MV_0 - MV_2$). These latter are smaller than the actual MVs: with this method the compression ratio increases.

3. The number of CPMVP (Control Point Motion Vector Predictors) in the VTM standard **candidate list** of is two. The authors propose to **increase** this value to four, increasing the complexity of the algorithm but improving the compression gain by adding more chances to find the optimal MVs.

4. There is a modification in the iterative refinement algorithm described in Section 2.2.1. In particular, the way in which $\{mv_0^{(i+1)}, mv_1^{(i+1)}\}$ is calculated starting from $\{mv_0^{(i)}, mv_1^{(i)}\}$ is changed. This new method is claimed to improve estimation accuracy.

5. The authors propose to merge the Normal (from Translational Motion Estimation) and Affine Merge Mode into a "Unified Merge Mode". According to their considerations, this should be done to simplify the syntactic structure of the encoded stream.

To measure the performance of this framework, it has been implemented on JEM-7.0 [11]. There is an average **BD-BR reduction** of 0.82%, while the encoding and decoding **time** ($T_{enc,AVE}, T_{dec,AVE}$) is **increased** of 1% and 3% on average, respectively.

In *"Design of efficient Perspective AME/AMC for VVC"* [4], there and **additional CPMV** is added to the three CPMVs already present in the Affine Motion Model. This makes the modified algorithm an "8-parameter model", it is shown in figure 2.9. Similar to what happens when moving from a 4-parameter model to a 6-parameter one, introducing the 8-parameter possibility enables to predict of more complex movements, like **blocks distortion**. This complicates the AME algorithm while achieving an average BDBR reduction: $\Delta BDBR_{AVE} = -0.1\%$.

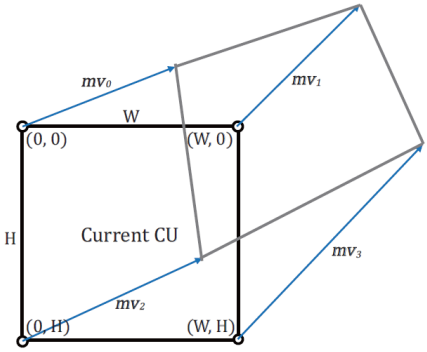

Figure 2.9. The 8-parameter model proposed in [4]. Figure from [4].

In *"Fast Affine Motion Estimation for VVC"* [37], the idea proposed is to **reduce** the number of times the **AME** is performed by avoiding it when it does not bring significant advantages in terms of compression capabilities.
Let J_{AME} and J_{CME} be the Rate-Distortion cost of Affine Motion Estimation and Conventional Motion Estimation, respectively. If for a given Coding Unit $p(A|S_{par})$ is the probability that $J_{AME} < J_{CME}$ when the CU's parent prediction mode is "Skip Mode", the authors have calculated $p(A|S_{par}) = 9\%$ on average. With such a result, their Fast Affine Motion Estimation algorithm avoids AME for all the sub-blocks in a Coding Unit whose prediction mode is "Skip Mode". This modification results in a reduction of the encoding time spent performing AME (T_{AME}) by 37%. In terms of compression ratio, there is a $\Delta BDBR_{AVE} = +0.1\%$.

In *"An improved Fast AME Based on Edge Detection Algorithm for VVC"* [38] there are two proposed techniques:

1. *Fast gradient prediction*: This is a technique that can be applied to the iterative refinement algorithm described in Section 2.2.1. This one is a gradient descent algorithm, whose speed is constant. The authors propose the introduction of a coefficient called "momentum", which can **accelerate** or **decelerate** the search for the best MV.

2. *AAMVP for 6-parameters*: In VVC the AAMVP (Affine Advanced Motion Vector Prediction) is used for the 4-parameter model only: the third motion vector is used for understanding the quality of the other two MVs. In the way this algorithm is

designed, it cannot be used for the 6-parameter model. The authors propose a way to use AAMVP with the 6-parameters model too: they use the motion-compensated block proportions to estimate how much the usage of 3 MVs is appropriate.

These two modifications to the VVC standard complicate the encoding process, with an average encoding time increase of the $\Delta T_{enc,AVE} = +15\%$, but with a BDBR variation of $\Delta BDBR_{AVE} = -23\%$.

2.5.2 Comparison and conclusions

Table 2.4 summarizes the main ideas and results about the other works on AME presented in section 2.5.1, together with the results from this thesis work.

Work	Proposal	Performance
"An Improved Framework of Affine Motion Estimation in Video Coding"	Simplified Affine Inter mode MV difference coding Increase the candidate list size Iterative refinement algorithm improvements Unified Merge Mode	$\Delta BDBR_{AVE} = -0.82\%$ $\Delta T_{enc,AVE} = +1\%$ $\Delta T_{dec,AVE} = +3\%$
"Design of efficient Perspective AME/AMC for VVC"	New 8-parameter model	$\Delta BDBR_{AVE} = -0.1\%$
"Fast Affine Motion Estimation for VVC"	AME usage optimization	$\Delta BDBR_{AVE} = +0.1\%$ $\Delta T_{AME} = -37\%$
"An improved Fast AME Based on Edge Detection Algorithm for VVC"	Fast gradient prediction AAMVP for 6-Parameters	$\Delta T_{enc,AVE} = +15\%$ $\Delta BDBR_{AVE} = -23\%$
Proposed method	New candidate construction algortihm Simplified AME algorithm	$\Delta E_{r,AVE} \simeq +2\%$

Table 2.4. Main ideas and results about all the works on AME.

This brief study about other works on AME remarks on what has been observed in section 1.5, when studying VVC's most complex blocks. When acting on the processing chain, the compression ratio and encoding time behave the opposite way. **Adding features** to the methods used typically **reduces** the **bit-rate**, but complicates the encoding process, like in [34, 4, 38]. On the contrary **simplifying the algorithm**, results in a **reduction** of the **encoding time** but at the cost of higher bandwidth and memory requirements, like in [37]. Each one of the presented studies does this in its original way, working on different stages and achieving distinct results.
Since the other works about AME act on algorithm parts that are not the one in this work, the proposed method is **orthogonal** to the others. It is possible to integrate more than one solution from Table 2.4 and eventually evaluate the performance.

Chapter 3

Hardware implementation

The simplified Affine Motion Estimation algorithm proposed in section 2.3 can be straightforwardly implemented in hardware following the well-known electronic systems design flow. The starting point is the Matlab software model reported in B.1 and B.2 appendices. The whole algorithm is composed by two methods, which are the "candidate construction" and "best-candidate choice", described in sections 2.3.1 and 2.3.2 respectively. Their functionality is independent of each other, they are independently replaceable with the exact VTM software. In order to preserve this **modularity** of the system, the hardware architecture has been divided into three sub-blocks (shown in figure 3.1):

- The **Constructor** block, which implements the construction algorithm. It takes as inputs the Neighbor CUs motion vector candidates ($CMV_{0,1,2}$) and computes the constructed MVs ($cMV_{0,1,2}$) according to the procedure described in section 2.3.1.

- The **Extimator** unit. It performs the best-candidate choice algorithm described in 2.3.2. It starts reading the CPMVPs (Control Point Motion Vector Predictors) which can be supplied by the constructor or the VTM software depending on the candidate type. Subsequently, it performs affine motion estimation with the mentioned motion vectors. To do so, it reads the Coding Unit and Reference Frame's pixel intensity values from the system's memory.

- The **Memory** stores the Coding Unit of the frame to be encoded and part of the Reference frame. In the subsequent sections, some considerations about its size and organization are addressed.

The first two blocks can be considered a sub-part of the "**AME Architecture**", which is the component designed using the Very high speed integrated circuit Hardware Description Language (VHDL) and then synthesized. The memory unit in this thesis work is a virtual component used for the AME Architecture verification.

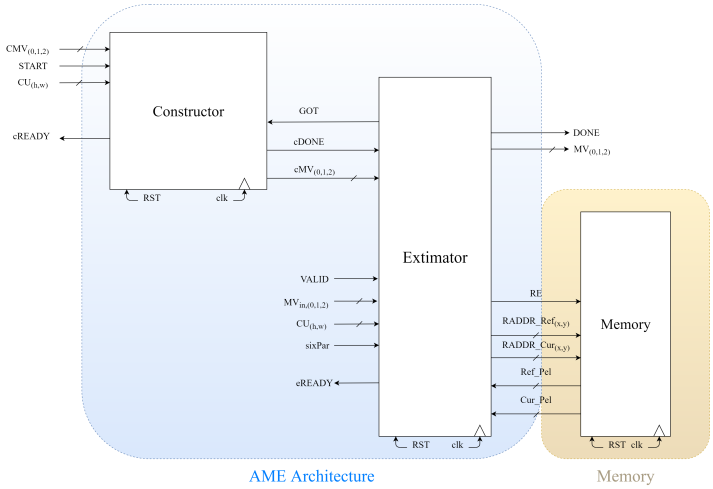

Figure 3.1. Proposed architecture top-level blocks.

3.1 The Constructor component

The constructor component is divided into a **Control Unit**, which handles the communication with the other system elements, and a **Datapath**, responsible for elaborating the input data and producing the construction result. Figure 3.2 shows the mentioned units.

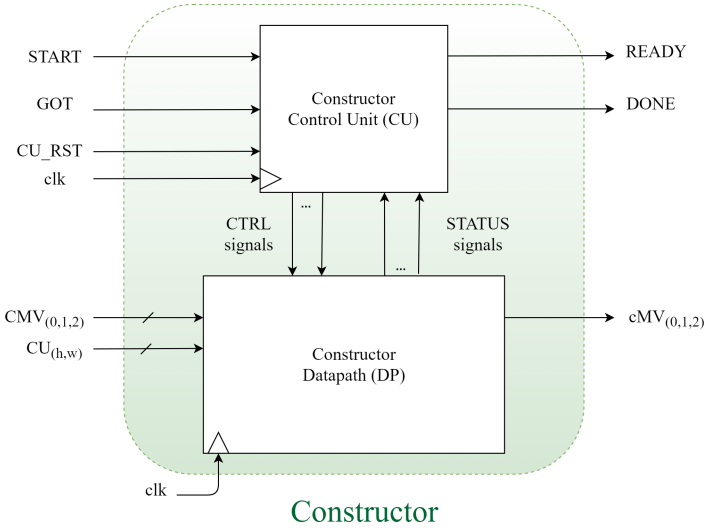

Figure 3.2. Constructor high-level block diagram

3.1.1 Constructor Datapath

The Constructor Datapath has been designed to implement the equations (2.3-2.4). Consideration must be done about data parallelism. In VVC, motion vectors are expressed in units of $1/16th$ of a pixel [9]. This is done to improve the image quality in the video decoding process. The range of values Motions Vectors can assume depend on the **Search Window** (SW) size. This parameter is the maximum allowed distance for which the VTM considers a possible Motion Estimation. This size is set in the VTM parameters (for example using the *config* file in report 2.1). A too-large SW complicates the Motion Estimation process, a too-small one makes the encoder incapable of performing Motion Estimation. For this thesis work, its size has been left to its default value of 64×64.
This means that the maximum number of pixel movements a MV can represent is 64 pixels in any direction (where the movements upwards and towards the left are represented by a negative sign). Considering the MV resolution, the range of values a motion vector can assume is $[-1023; 1023]$. Using a 2's Complement representation, **eleven** bits are sufficient to store a Motion Vector with this Search Window.
According to the algorithm specifications the height and width values (CU_h, CU_w) a CU can assume are (16,32,64). To represent these numbers as unsigned, seven bits would be required. Using a simple coding strategy, **two** bits only have been used to represent (CU_h, CU_w). This is simply achieved by taking the two MSBs of the unsigned binary representation. Table 3.1 shows how the coding is performed

$CU_(w, h)$	Base-10 unsigned	Encoded
16	0010000	00
32	0100000	01
64	1000000	10

Table 3.1. How the CU width and height are encoded.

Figure 3.3 shows the entire Constructor Datapath. Some components are described in detail later in this section. Besides the most common operators, like multipliers; register files; pipeline registers (represented by the red dotted lines in the figure), and comparators, there are some complex components specifically designed for handling the construction operation. Their functionality and structure are briefly illustrated subsequently.

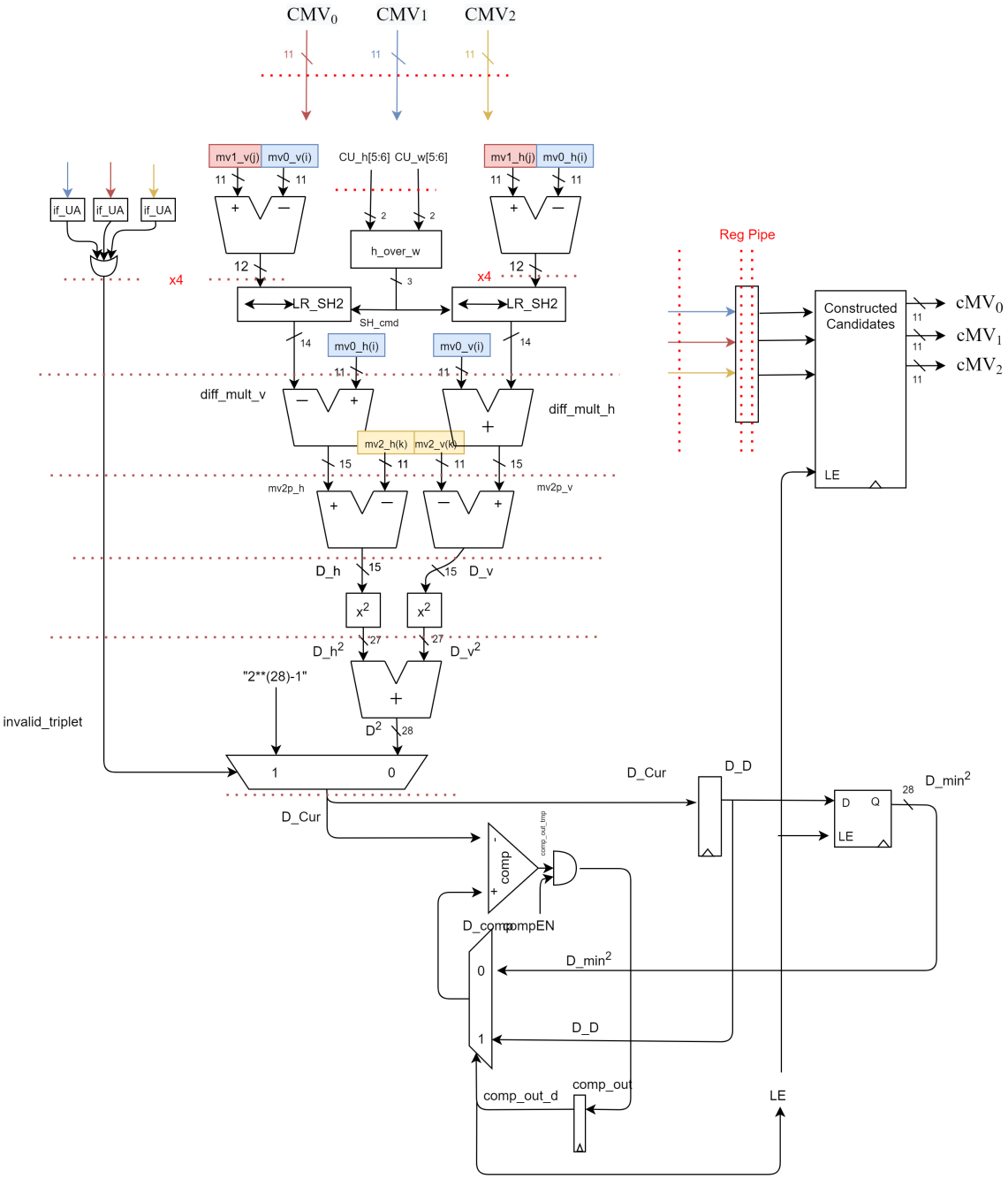

Figure 3.3. Constructor Datapath. The dotted in lines in red represent pipeline registers.

h_over_w
Equation 2.4 requires to compute the product of motion vector difference (dMV) by the factor $\frac{h}{w}$. This would normally require a multiplication and a division. This operations can be avoided, saving precious resources, by considering that both

$$(w, h) \in \{16,32,64\} \tag{3.1}$$

Therefore, the cited multiplication can result to be only by a factor 2^n with

$$n \in \{-2, -1,0,1,2\}. \tag{3.2}$$

The component *h_over_w* **computes** the value of n, with the purpose of feeding another unit, the *LR_SH2*, which simply executes a shift on dMV.
Figure 3.4 shows the *h_over_w* block diagram, where *"RSH"* and *"cmd_RSH"* are two right shifters whose enable signals are controlled by the Control Unit. *"SH_cmd"*, instead, is the mentioned n, represented on three bits in "magnitude an sign" representation. This allows *LR_SH2* to perform its operation more easily.

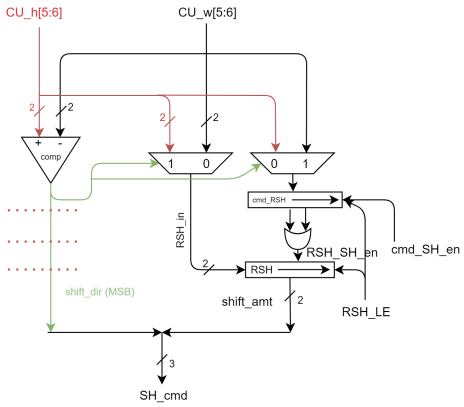

Figure 3.4. The *h_over_w* block diagram.

LR_SH2
This component name stands for *"Left or Right Shift register by 2 position at most"*. It **shifts** the previously mentioned dMV by $|n|$ positions left or right depending on $sign(n)$. The output of this component is thus $dMV \cdot 2^n$.
Figure 3.5 shows the *LR_SH2* block diagram, where *RSH* and *LSH* are right and left shift registers, respectively, whose enabling signals are derived from *SH_cmd*.

The latency of this component, as well as for *h_over_w*, is 2 clock cycles. In each clock cycle, the value of w and h is observed and the choice of shifting towards a certain direction is taken. This latency could be reduced by using a barrel shifter which can perform a shift by any amount of position in 1 clock cycle only. The drawback is that barrel shifters are complex components that can **increase** the area

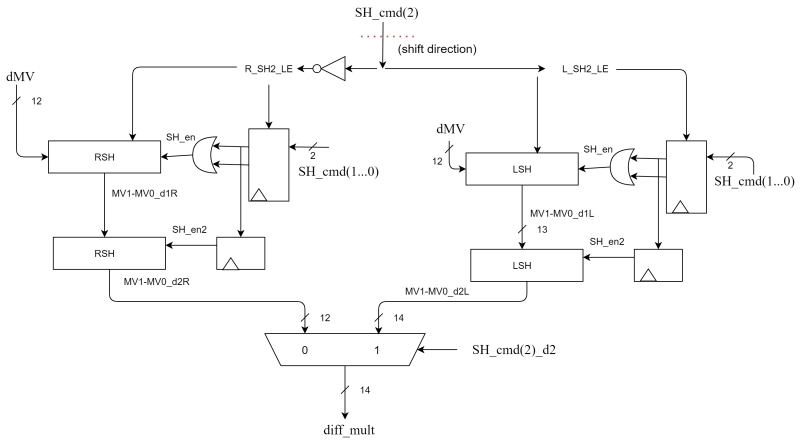

Figure 3.5. The *LR_SH2* block diagram.

and delay of the circuit. Since the value of n is 2 at most, the use of two registers cascaded is considered a good choice. Notice that the fractional bits in the result are discarded. This solution agrees with the floor function included in the approximate construction model.

if_UA
This component, whose name is short for *"if UnAvailable"*, calculates if one or more MV of each triplet is unavailable. In fact, unavailable MVs are set to $(-1024)_{10}$. Inside *if_UA* there is simply a **comparator** which checks if one of the input MV is equal to $(-1024)_{10}$. If yes, the flag "invalid_triplet" is set to 1. In this case, the value of the distortion associated with that triplet is set to the maximum possible value. In this way, the triplet is discarded by the construction process.

3.1.2 Constructor Timing diagram

Having all the datapath components with their respective control signals defined, a **timing diagram** (**TDG**) has been laid down. It defines how data and control signals must behave during execution such that the functional specification (defined by the Matlab model) is met. Figure D.2 shows the designed TDG. The data and control signals have been grouped into functional blocks. For example, signals belonging to the *h_over_w* and *LR_SH2* are highlighted. Due to the circuit pipelining, the constructor **latency** is 25 clock cycles. This information is essential to evaluate the circuit performance later.

3.1.3 Constructor Control Unit

The Constructor Control Unit, as shown in figure 3.2, is responsible for the communication with the outside circuits and for producing correct Datapath control waveforms. The first task is achieved using the following signals:

- **START**: When asserted, the feeding circuit informs the CU that the first MV triplet is available and the construction process can start.
- **READY**: With this signal, the CU asserts that it is available for a new elaboration.
- **DONE**: When the construction process is terminated and $cMV_{(0,1,2)}$ are valid, the CU asserts this signal.
- **GOT**: This signal is used by the outside circuits (for example, the "Extimator"), to inform the constructor that the output constructed MVs have been correctly stored. In this way, the CU can start a new execution without overwriting a result that has not been sampled yet.
- **CU_RST** : This is an asynchronous reset that forces the CU to move to its reset state `"ON_RESET"`.

The Control Unit design is mainly focused on the **state diagram** design. It describes which are the available CU states, the input signals which trigger the state transitions, and the output signals in each state. It is reported in figure D.1. Notice how the already mentioned control signals make the CU move from one state to another one.

3.2 The Extimator component

The Extimator component implements the "best-candidate choice" algorithm described in section 2.3.2 which concludes the simplified AME process. Figure 3.6 shows the system block diagram. It is mainly composed of two groups of blocks: the first one is the **Control Unit block**, which includes the Control Unit (CU), the CU_Adapter and the Ready Handler (it is colored in green in the figure); the second one is the **Datapath** block, which contains the Datapath (DP) only (it is highlighted in orange).

3.2.1 Extimator Datapath

The Extimator Datapath is depicted in figure D.3. Notice that the "Memory" drawing has been split into two parts to optimize the visualization. This choice in the representation highlights also the two main purposes of the Extimator unit. The first one is to compute the **Relative Addresses** to access the Memory at the correct location, which depends on the representative blocks' position and the motion vectors. The second one is to compute the **Sum of Absolute Differences** for each candidate and choose the one with the lowest one.
Besides the most common operators, like multipliers; register files; pipeline registers (represented by the red dotted lines in the figure) and comparators, there are some complex components specifically designed for handling the estimation operation. Their functionality and structure are briefly illustrated subsequently.

firstPelPos
This component name stands for "First pixel position calculator". Its purpose is

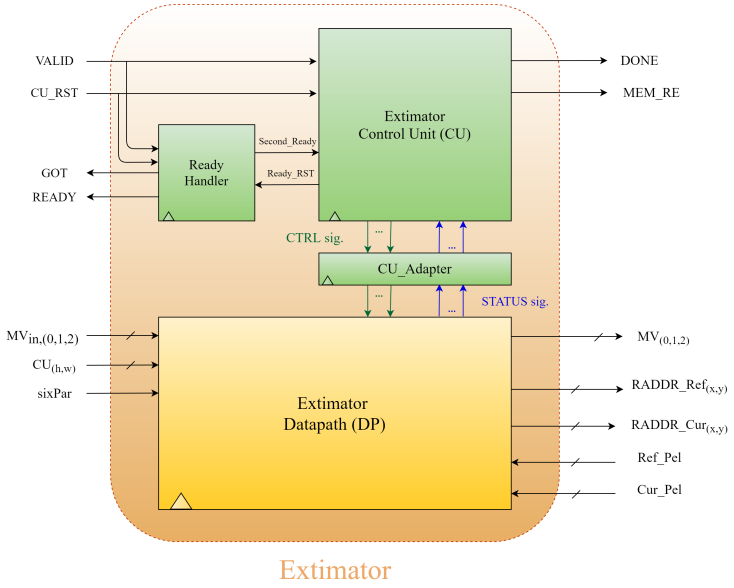

Figure 3.6. Extimator high-level block diagram

to compute all the representative blocks' first pixel position relative to the current Coding Unit. The reference system used assumes (0,0) as the current CU position in the current to-be-encoded frame. For example, with reference to the 16×16 CU in figure 3.7, the output pattern that *firstPelPos* must generate is

$$(x_0, y_0)_0 = (0,0); (x_0, y_0)_1 = (12,0); (x_0, y_0)_2 = (0,12); (x_0, y_0)_3 = (12,12) \qquad (3.3)$$

This pattern is obtained by strategically using two T-type "flip flops" and two modulo 4 counters. With reference to figure 3.7, the first two compute the current representative x and y positions (in blue), and the other two address the current 16×16 block in the CU (in orange). The two pieces of information are wrapped up together to obtain (x_0, y_0). Figure 3.8 shows the *firstPelPos* part for the calculation of x_0. The circuit which produces y_0 is identical, it is not reported to optimize the graphical representation.

Since x_0 and y_0 are CU coordinates, they can assume values in the range $[0; 63]$. Therefore, they are represented on six bits as **unsigned** integers. The $last\_block_{(x,y)}$ flag informs the Control Unit if the current 16×16 block x or y component has reached its maximum allowed value. This information is essential for the CU for generating the correct $(x_0; y_0)$ sequence depending on the Coding Unit size. In fact the signals $\texttt{CE\_REP}_{\texttt{x,y}}$, which increase the $CurRep_{x,y}$ value; and $\texttt{CE\_BLK}_{\texttt{x,y}}$, $\texttt{RST\_BLK}_{\texttt{x,y}}$ which increase or reset $CurBlock_{x,y}$, are generated by the Control Unit depending on the $last\_block_{(x,y)}$ values. The way this process is carried on is explained in the section relative to the Extimator Control Unit (3.2.3).

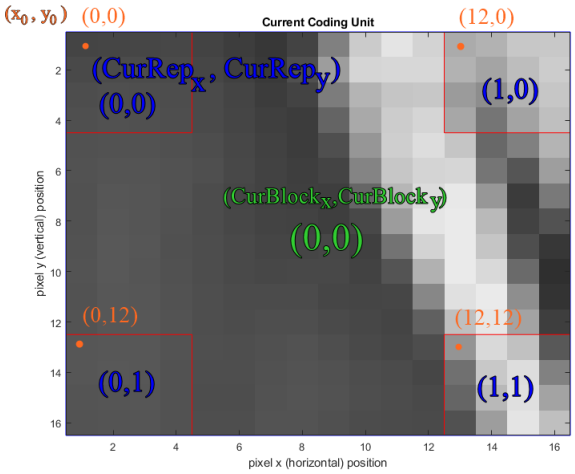

Figure 3.7. A 16×16 example CU with the block, representative and pixel coordinates reported.

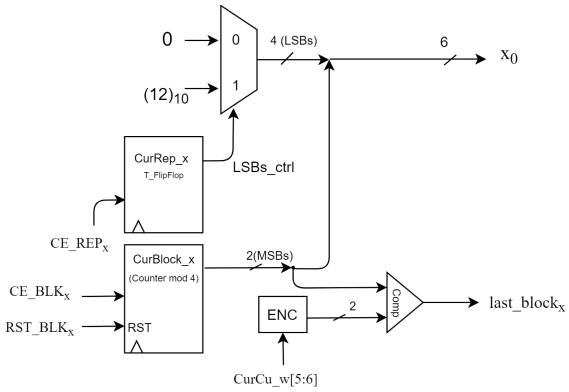

Figure 3.8. The *firstPelPos* component for x_0

R_SH2

This component is similar to the Constructor's *LR_SH2*, with the difference that it handles only the "right shift" operation. Having in input a motion vector difference $dMV = mv_{(1,2)}^{(h,v)} - mv_0^{(h,v)}$, this block computes

$$\{a,b\}_{(1,2)} = \pm\lfloor \frac{dMV}{(h,w)} 2^4 \rfloor \frac{1}{2^4} \tag{3.4}$$

for the equations (2.8-2.10). Notice that $(h,w) \in \{16,32,64\}$, this means that the input must be **shifted** by at least 4 positions. Thus, instead of being {4,5,6} the number of possible shifts towards the right, it is {0,1,2} and the result is expressed in **fixed point** representation `FXP 8.4` (8 bits for the integer part and 4 for the

fractional one). The last two bits in the result are discarded, as already mentioned in section 2.3.2. This allows to save two bits in the following components' parallelism while still keeping a low Rounding Bias. In fact, since these bits are placed after four fractional bits, the Rounding Bias here is 2^{-4} times the one computed in Section 2.3.1. The **choice** of discarding no more than two bits comes from some tests performed on the Matlab model. A variable `fxp_prec` is used, it stores the number of fractional bits in the fixed point representation. What has been discovered is that if `fxp_prec` is less than 4, the estimation results of the Matlab model on the 30 test sequences start to change significantly with respect to the exact one with 6 bits of fractional part. Figure 3.9 shows the *R_SH2* block diagram, where *RSH* components are shift registers whose shift is enabled by `SH_en` and `SH_en2`.

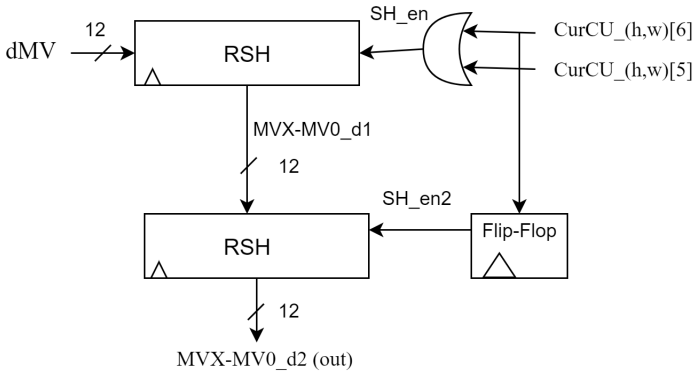

Figure 3.9. The *R_SH2* component.

MULT_1

This component handles the product $\{a, b\}_{(1,2)} \cdot \{x, y\}$ of equation 2.8. The coefficients $\{a, b\}_{(1,2)}$ are signed fixed point numbers represented on twelve bits with notation `FXP 8.4`. $\{x, y\}$ components, instead, are the $\{x_0, y_0\}$ CU coordinates generated by *firstPelPos*. They are represented as unsigned integers on 6 bits. To multiply these values, $\{x_0, y_0\}$ are properly extended to assume the same representation as the $\{a, b\}_{(1,2)}$.

In order to reduce the multiplier cost and maximize the **usage percentage** of the component, Instead of using a twelve-bit multiplier with unitary latency, a six-bit one with four-cycles latency (neglecting the fifth one needed to load the output register) is introduced. Its Register Transfer Level (RTL) representation is reported in figure D.4. Together with the 6-bit multiplier (labeled as *Mult0*), there are several registers and multiplexers to handle the partial products sum. Figure 3.10 reports a multiplication example's timing diagram. The operands as $op1$ and $op2$, both on 12 bits, are split in two halves of 6-bit-length, named $(op1[0], op2[0])$ and $(op1[1], op2[1])$. With this concept clarified, the signals involved in the multiplication are:

– **mult_in**: Contains the current *Mult0* input. At each cycle, new couple *(op1[i],op2[i])* halves are fed to *Mult0*, to compute the partial product **mult_out**.

- **is_signed**: The 6 LSBs of $op1$ and $op2$, $(op1[0], op2[0])$, must be treated as unsigned numbers. The signal $is\_signed[i]$ informs *Mult0* whether the input $op-i[(0,1)]$ must be treated as unsigned or not.
- **SH_en**: The equation the component implements when splitting $op1$ and $op2$ into two 6-bit halves is the following

$$op1 \cdot op2 = op1[0] \cdot op2[0] + 2^6(op1[1] \cdot op2[0] + op1[0] \cdot op2[1]) + 2^{12} op1[1] \cdot op2[1] \tag{3.5}$$

  The multiplication by 2^{12} is obtained by shifting the partial product of 6 bits to the correct cycle. This is achieved by using a shift register whose enabling signal is **SH_en**.
- **product_int_rst**: resets the partial products register at the end of the elaboration
- **sum**: Stores the temporary result of the multiplication.

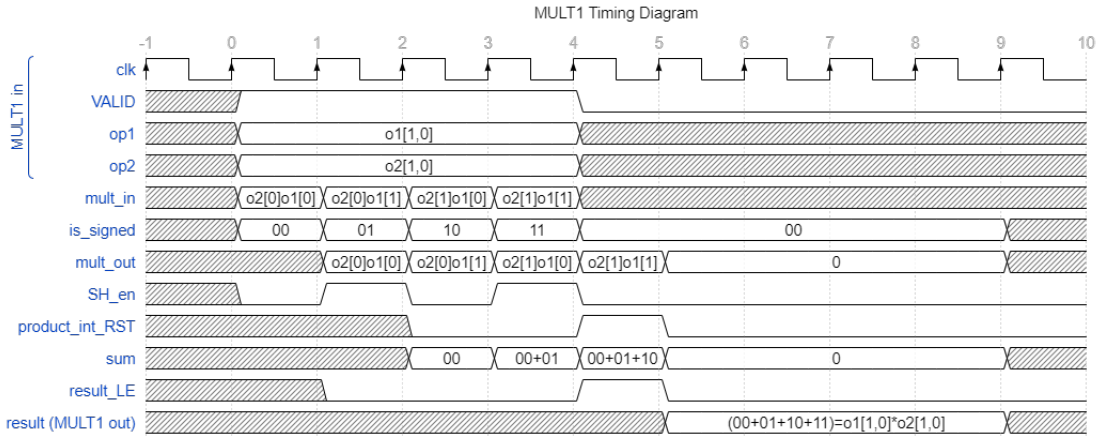

Figure 3.10. A MULT_1 multiplication example's timing diagram.

Notice that the only input signals are $clk, op1, op2$ and *VALID*. The other control signals are internally generated using counters.

ADD3

This adder computes the sum $(a_{(1,2)}x + b_{(1,2)}y + mv_0^{(h,v)})$ of equation 2.8. A straightforward implementation of a three-input adder could be made by two 2-inputs adders cascaded. The solution proposed in the extimator datapath is **one 2-input adder** re-used. This allows to save one adder and maximize the usage percentage of the component. Figure 3.11 shows the ADD3 block diagram. Notice that, similarly to MULT_1, ADD3 also needs a VALID signal to be issued when the operands are provided. This is essential to move the multiplexer at the inputs.

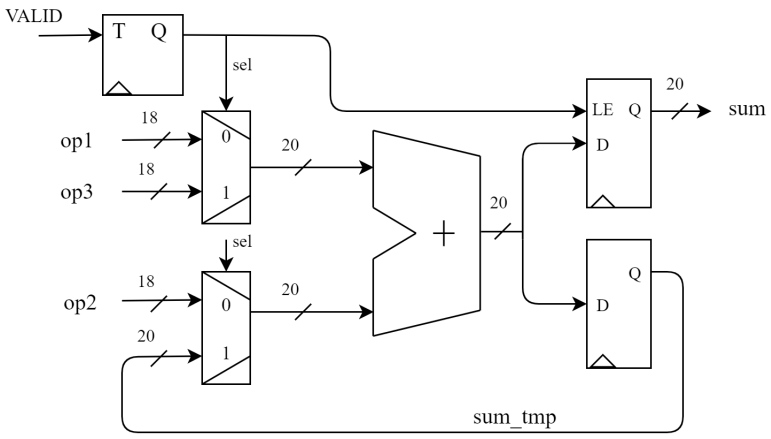

Figure 3.11. ADD3 component's block diagram.

Round
As already mentioned, the MVs are expressed in the VTM as multiples of $1/16 - th$ of a pixel. This form can be called the **exact** expression of the motion vector $(MV_{ex} = mv^h_{(ex)}, mv^v_{(ex)})$. To perform simple Translational Motion Compensation on the CU representative blocks, MVs expressed as multiples of 1 pixel are needed. They are obtained from MV_{ex} simply by discarding the 4 LSBs (this operation corresponds to a division by $2^4 = 16$) and **rounding** the result. The rounding method uses, in this case, is the **round to nearest** scheme. The choice of this method is done because it has a low maximum rounding error, corresponding to 1/2 post-truncated LSB, which means 1/2 pixel here.

3.2.2 Extimator Timing diagram

The extimator is supplied with MV candidates from the VTM and the constructor. This latter produces a candidate of the "Constructed" (C) type, while the VTM software provides the "Inherited" (I) and "Translational" (T) ones (see section 2.2.1). The gathering process of the I and T type candidates is simpler than the one of the C types. Therefore, in this thesis work, it is assumed that, at the start of the estimation process

- the I and T-type candidates are **ready**
- the Constructor starts the construction process, so **25 clock cycles** are needed before the C-type candidate MVs are valid

There are two possible scenarios about how the candidates are presented to the extimator:

- In the first one, the MVs are all of the Inherited and Translational types. This means that, at the start of the estimation process, they are all ready and fed to the

Extimator as soon as it is available. Figure 3.12 shows how the inputs are presented in this case.

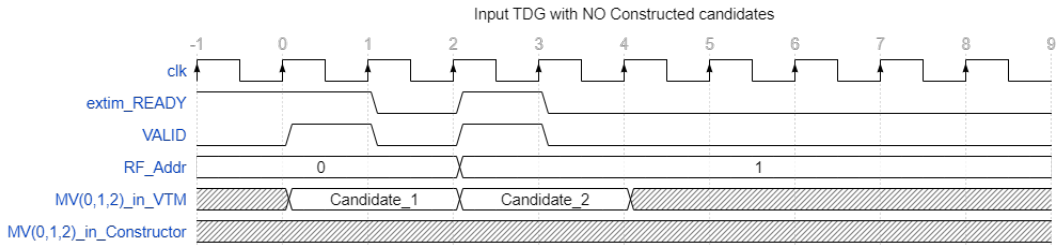

Figure 3.12. How input MVs are presented to the Extimator when there are no C-type candidates.

The protagonist signals in this timing diagram are

- **extim_READY**: The Extimator READY signal, which informs if it is available to accept candidates. Notice that in clock cycle "1" it is negated (signaling that no MVs are accepted). That is because the Control Unit's **reactivity** is such that two clock cycles are needed to move the Register File input address (**RF_Addr**). Therefore, if a new candidate was supplied during that clock cycle, it would have overwritten the first one.
- **$MV_{(0,1,2),in}$_VTM**: The input candidates fed by the VTM software. According to the previous assumptions, they are all available at the start of the estimation process and fed to the Extimator as soon as it is available.
- **$MV_{(0,1,2),in}$_Constructor**: The input candidates fed by the Constructor. Since this is an example with no C-type candidates, the value of this signal is always *"Don't Care"*.
- **VALID:** This is an input signal for the Extimator. It is obtained by the logic `OR` between the constructor *DONE* output and the VTM software's *VALID*.

- In the second scenario, one candidate is of the Inherited or Translational type, so it is available at the start of the estimation process. The other one is of Constructed type, so it is available **after** the Constructor **latency**, which is of 25 clock cycles. Figure 3.13 shows how the inputs are presented in this case.

  The signals meaning are the same as the one previously mentioned. **Among** the 30 test sequences used for the algorithm and architecture analysis, summarized in Table C.1, this is the most frequent situation (70% of the sequences).

The main difference between the two cases is that when the Constructor is needed, the second candidate is not immediately ready, creating the risk of a **bubble** in the Extimator's estimation process. In fact, if the Constructor's latency is **longer** than the number of clock cycles required to elaborate the first candidate ($N_{el,1}$), the Extimator is

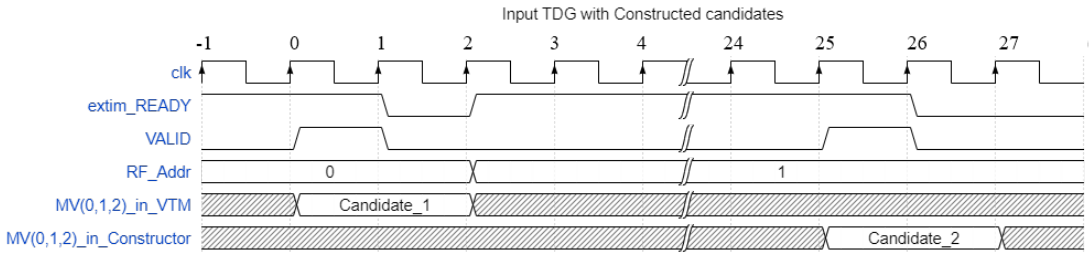

Figure 3.13. How input MVs are presented to the Extimator when there is a C-type candidate.

stalled until the constructed MVs are ready.
Being the Constructor latency equal to 25 clock cycles, the only situation in which this happens is when the Coding Unit size is 16×16. In fact:

$$N_{el,1} = N_{el,SB} \frac{CU_w \cdot CU_h}{16 \times 16} \tag{3.6}$$

where

- $N_{el,1}$ is the number of clock cycles required to elaborate the first candidate.
- $N_{el,SB}$ is the number of clock cycles to elaborate a 16×16 sub-block. This is constant and equal to 16, assuming that the memory is able of reading four pixels per frame per clock cycle.
- (CU_w, CU_h) are the Coding Unit width and height respectively.

Being $(CU_w, CU_h) \in \{16,32,64\}$, the only case for $N_{el,1} < 25$ is $CU_w = CU_h = 16$. In this situation only, there is an additional delay N_{add} to be considered in the Extimator latency, equal to

$$N_{add} = 25 - N_{el,1}(CU_w = CU_h = 16) = 25 - 16 = 9 \tag{3.7}$$

The Extimator latency T_{ext} can be computed as

$$T_{ext} = P_{depth} + 2 \cdot N_{el,1} + c \cdot d \cdot N_{add} \tag{3.8}$$

Where

- P_{depth} is the pipeline depth of the circuit, equal to 22.
- c is equal to 1 if one of the candidates is of the Constructed type, otherwise it is 0.
- d is equal to 1 if the CU dimension is 16×16, otherwise it is 0.
- $N_{el,1}$ and N_{add} are already been defined in equations 3.6 and 3.7.

CU size	$N_{el,1}$	T_{ext}
16×16	16	63
32×32	64	150
64×64	256	534

Table 3.2. Extimator possible latency values.

For all the subsequent analyses the "worst-case" scenario is assumed, with $c = 1$. In table 3.2, some examples of Extimator latency values have been calculated.

These data are essential to evaluate the architecture performance. This analysis is performed later in the next chapters.

Figure D.5 reports an **example** of Extimator **timing diagram**. It has been drawn assuming as input the test sequence number 26 from table C.1. Here, with reference to equation 3.8, there are $(c = 1; m = 1)$ and $T_{ext} = 63$, in agreement with the result in table 3.2.

3.2.3 Extimator Control Unit

The Extimator component is more complex than the Constructor one. This is reflected in the control circuit, colored in green in figure 3.6. The control system in fact is made of three components, each one with its purpose.

Ready Handler
The name of this Finite State Machine (FSM) is because it is responsible for generating the `READY` signal. As already shown in the timing diagram in figure 3.12, this signal must be negated one clock cycle after the `VALID` signal is issued. This cannot be done by the CU due to its slow reactivity.
The "Ready Handler" (RH) is indeed more reactive but does not have its input signals sampled. In general, when this is done, there is the probability that the Control Unit's output generator net could increase the critical path delay. Since the RH is simpler than the CU, this risk is reduced.

Besides the already mentioned `READY`, there are other signals which are generated by the "Ready Handler" which are essential for correct processing:

- `Second_Ready`: Informs the Control Unit that the second candidate has been correctly sampled and stored in the Datapath's Register File. The second candidate can be provided at any given moment during the Motion Estimation, even when the **CU** is **busy** elaborating the first one. When the CU is no longer occupied, it looks at the *Second_Ready* signal to know whether it has to wait for the second candidate or it can start its elaboration.
- `GOT`: This signal goes directly in the Constructor's CU, to inform it that its produced constructed candidate has been correctly sampled. In this way, it can

start a new execution without overwriting a result that has not been sampled yet.

Figure 3.14 shows the "Ready Handler" FSM state diagram. Notice that it has six possible states only, and three signals as output. This confirms what was previously said about its low complexity compared to the Control Unit, which can assume 27 states and generates 23 output signals.

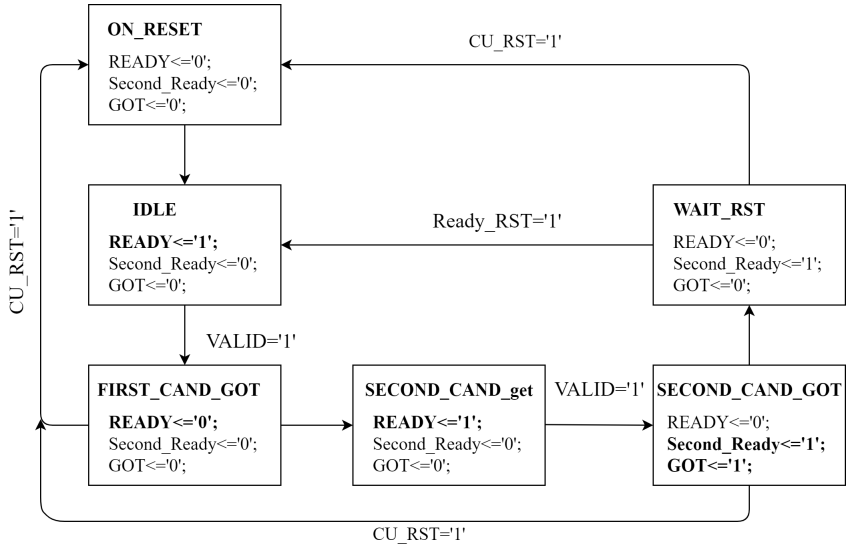

Figure 3.14. "Ready Handler" FSM state diagram.

Control Unit (CU)

The Extimator's Control Unit (CU) generates 23 signals which are essential for the correct Motion Estimation process. It can assume 27 states as shown in its state diagram in figure 3.15, which have been divided in five groups according to their purpose in the processing chain:

1. **Initialization and Idle** states (in red). They are the `ON_RESET` and `IDLE` states. In the first one, all the sequential components in the Extimator are reset using an asynchronous signal.
   In the `IDLE` state, the reset from the first two pipeline layers is removed to compensate for the CU's low reactivity. The *READY* signal is asserted, and the Extimator is ready for a new elaboration.
2. 16×16 **sub-block processing** states (in white). For each candidate, the simplified Translational Motion Compensation is performed looping over each 16×16 sub-block. Each four-state group in white can be identified by the name `(X)REP_WAIT(N)`, where `X` is related to the representative's number (S: Second, T: Third ...) and `N` is the representative block's 4-pixel row which is elaborated in that state.

3. **New action choice** states (in purple). After the elaboration of a 16×16 Coding Unit (CU) sub-block, the Control Unit chooses the new state depending on the CU size and how much candidates have been currently completed their SAD calculation.
   If there are sub-blocks yet to be processed, the new state must be `NEW_LINE` or `NEXT_BLOCK` depending on the position of the next block. The movement is performed by acting on the *firstPelPos*' $\texttt{CE\_REP}_{x,y}$, $\texttt{CE\_BLK}_{x,y}$, $\texttt{RST\_BLK}_{x,y}$ control signals.
   If there are no more 16×16 sub-blocks left, the new state must be `TERM_CAND1` or `TERM_CAND2` ("Terminate candidate 1/2") depending on how many candidates have been already processed. In these states, the current SAD (*Cur_SAD*) value is compared with the lowest already calculated (*SAD_min*): if the first is lower than the second, the "Best candidate" is set as the current one and *SAD_min* is set as *Cur_SAD*.

4. **Second candidate wait** states (in yellow). When the first candidate elaboration is completed, the second one could be ready or not depending on its type and the Coding unit size (as already presented in section 3.2.2).
   `SREP_CAND2` is equivalent to `SREP_CAND1` and it is taken when the second candidate is already available at the end of the first one's processing. `CAND2_WAIT` is taken when the Constructor has not completed the construction algorithm yet. In this state, the Extimator waits for the *VALID* signal to be asserted to start the second candidate's elaboration.

5. **Estimation completion** stages (in green). After the second candidate's elaboration is completed, the "Best Candidate" register contains the Register File (RF) address where the best candidate is stored. The last steps towards the estimation process completion are taken by this group's stages. The operations performed are:

   - Wait for the DP's final stage latency needed to compute the second candidate's SAD (`WAIT_COUNT`).
   - Issue the RF's Best candidate reading (`READ_BEST`).
   - Write the MVs result in the output Register (`WRITE_BEST`).
   - Show the result at the Extimator output by enabling the output register reading and asserting the *DONE* signal (`ESTIMATE`).

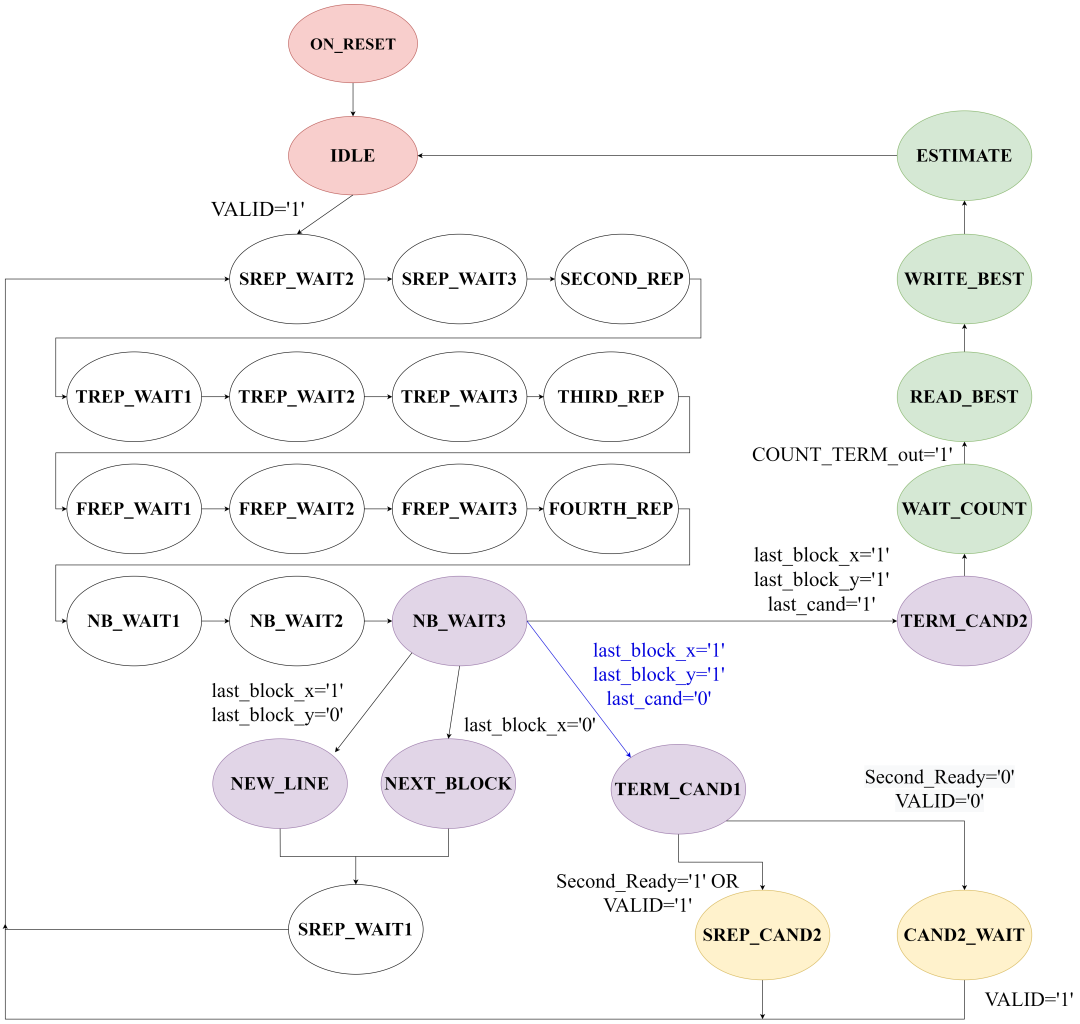

Figure 3.15. Extimator's Control Unit state diagram.

Control Unit Adapter
Due to the applied pipelining technique, some signals generated from the Control Unit in a given state are needed by some components several clock cycles later. The "Control Unit Adapter" block is responsible for adapting their timing using **preskew/deskew** pipe registers.

3.3 Components Usage percentage

In both the Extimator and Constructor, some Datapath (DP) elements are more involved in the data elaboration than others. It is useful to perform a **Usage percentage** (U_p) evaluation on them, to understand how much each component is exploited. If this value

is too low, there might be the possibility to apply some techniques to save complexity and increase U_p, at the cost of a few additional clock cycles delay.
This is what is done with the components **MULT_1**, **ADD3**, **LR_SH2**, **R_SH2**, where some resources are re-used with an increase in the usage percentage, a cost reduction and a little increase in the Extimator and Constructor's latency.

3.3.1 Definition

Let U_i be the usage of the i^{th} datapath component, defined as the number of clock cycles during which it is needed for the elaboration. If T_{el} is the component latency in clock cycles, the **usage percentage** $U_{p,i}$ of the component can be defined as

$$U_{p,i} = \frac{U_i}{T_{el}} \cdot 100 \tag{3.9}$$

The higher the $U_{p,i}$, the more the component usage is optimized for the elaboration.

3.3.2 Evaluation in the Extimator and Constructor

For both the Extimator and Constructor's usage percentage analysis, the T_{el} value in equation 3.9 has been set as the Extimator latency T_{ext} of equation 3.8. This is done because the **elaboration period** of Affine Motion Estimation is the Extimator latency. Moreover, while the Constructor delay is constant, T_{ext} depends on the Coding Unit size. It is interesting to observe how $U_{p,i}$ varies with T_{ext} and consequently with the CU size. Table 3.3 reports the Constructor components' usage percentage variation with different Coding Unit sizes.

Component	Units	Parallelism	U_i	$U_{p,i}$ $CU : (16 \times 16)$ $T_{el} = 63$	$U_{p,i}$ $CU : (32 \times 32)$ $T_{el} = 150$	$U_{p,i}$ $CU : (64 \times 64)$ $T_{el} = 534$
Register_in	1	66 b	12	19,05	12,70	2,25
Signed Adder	6	11 to 27 b	12	19,05	12,70	2,25
Right Shifter	2	2 b	3	4,76	3,17	0,56
Right Shifter	2	12 b	12	19,05	12,70	2,25
Left Shifter	2	12 b	12	19,05	12,70	2,25
Unsigned Comparator	1	28 b	12	19,05	12,70	2,25
Unsigned Adder	1	27 b	12	19,05	12,70	2,25
Signed Multiplier	2	15 b	12	19,05	12,70	2,25

Table 3.3. Constructor usage percentage analysis with different Coding Unit sizes.

What shows up is that the usage percentage **reduces** with the CU size. That is because the usage U_i of the Constructor components is independent of the CU size, while the elaboration time of the CU increases.
To **increase** $U_{p,i}$, some component re-usage techniques could be applied, but this would

increase the latency of the circuit. This could as a consequence increase T_{el} of the Extimator when one of the candidates is of the constructed type and the CU size is sufficiently small (see section 3.2.2). A trade-off must be considered in this situation.

Table 3.4 reports the Extimator components' usage percentage variation with different Coding Unit sizes.

Component	#	Par.	U_i 16×16	U_i 32×32	U_i 64×64	$U_{p,i}$ 16×16 $T_{ext} = 63$	$U_{p,i}$ 32×32 $T_{ext} = 150$	$U_{p,i}$ 64×64 $T_{ext} = 534$
Signed Subtractor (SUB1)	4	11 b	2	2	2	3,17	1,33	0,37
Right Shifter (R_SH2)	2	12 b	4	4	4	6,35	2,67	0,75
Signed Multiplier (MULT1)	4	6 b	32	128	512	50,79	85,33	95,88
Signed Adder (ADD3)	2	19 b	16	64	256	25,40	42,67	47,94
Round	2	19 b	8	32	128	12,70	21,33	23,97
Signed Adder for x (ADD1_x)	1	12 b	8	32	128	12,70	21,33	23,97
Signed Adder for y (ADD1_y)	1	12 b	32	128	512	50,79	85,33	95,88
Memory	1	64b	32	128	512	50,79	85,33	95,88
Subtractor (Pel_Sub)	4	8 b	32	128	512	50,79	85,33	95,88
Absolute Value Operator (ABS)	4	9 b	32	128	512	50,79	85,33	95,88
4-Input Adder (Pel_add)	1	8 b	32	128	512	50,79	85,33	95,88
Unsigned Adder (CurSAD_ADD)	1	18 b	32	128	512	50,79	85,33	95,88
Unsigned Comparator (Comp)	1	18 b	2	2	2	3,17	1,33	0,37

Table 3.4. Extimator usage percentage analysis with different Coding Unit sizes.

Notice that, differently from what happens in the Constructor, the **usage** U_i factor of most components **changes** with the Coding Unit size. Table 3.4 is divided into four layers since the behavior of components in the same layer is similar.

- In the **first** layer, the components `SUB1` and `R_SH2` have a constant U_i since they are used only once or twice at the beginning of a new candidate elaboration. This is why their usage percentage reduces with the CU size.

- In the **second** layer, there are the components relative to the MV calculation of each representative block. Therefore, their usage is

$$U_i = N_{rep} \cdot N_c \cdot R_f$$

  where N_{rep} is the number of representatives; N_c is the number of MV triplets in the candidates' list (equal to 2); R_f is how may times the component is re-used for a single result calculation. The multiplier `MULT1` is the one with the highest U_p since its $R_f = 4$, but it also adds 4 latency clock cycles. Notice how in this component group the usage percentage **increases** with the CU size. This is because with more representatives come more MVs to be computed, so the usage increases. The elaboration time increases too, but as the $N_{el,1}$ in the expression of $T_{ext} = T_{el}$ (equation 3.8) starts dominating over P_{depth}, T_{ext} becomes similar to U_i.

- In the **fourth** layer, there are the components responsible for computing the SAD value for the two CPMV candidates. They are needed every time a reading operation is performed on memory, this is why their $U_{p,i}$ is high. For the same reason as the previous layer, here the usage percentage **increases** with the CU size.
  This is because with more representatives come more pixels to be processed, thus the usage increases. The elaboration time increases too, but as the $N_{el,1}$ in the expression of $T_{ext} = T_{el}$ (equation 3.8) starts dominating over P_{depth}, T_{ext} becomes similar to U_i.
- The **fifth** layer contains the final SAD comparator only. It performs its operation only once per SAD value computed, so its U_i is constant and equal to 2. This is why it $U_{i,p}$ reduces as the CU size increases.

For the Extimator component, the usage percentages are quite high, except for the first and last elements which operate only once or twice per candidate. Therefore it has been chosen to leave the Datapath as it is, without applying further techniques. For future works, there is the possibility to change some components and, after modifying the control unit coherently, observe how the situation changes.

Chapter 4

Verification, Synthesis, and Performance

4.1 Logical Verification

After the design and implementation processes, the Design Flow requires the logical **verification** of the circuit. In this thesis work, the verification step is performed using *MATLAB*, for the generation of the test sequences, and a VHDL *testbench* to read the cited inputs and feed them to the architecture. Finally, a bash script automatically loops over all the 30 cases from table C.1. Figure 4.1 shows how this verification environment is set. Besides the testbench's **driver**, which is included in the former's VHDL description, the other components are all described in separate entities.

4.1.1 Simulation Script

As shown in figure 4.1, the set of test sequence inputs and expected outputs are collected together as a single large dataset. The purpose of the **simulator script** is to perform the simulation of a given test from table C.1 using the testbench compiled and loaded on Siemens's *QuestaSim*. In report D.1 the script content is shown. This program's aim is achieved by performing a sequence of steps, illustrated in its flow chart in figure 4.2.

4.1.2 Memory

As previously mentioned, the system's Main memory is virtual and used for design verification only. However, a brief **analysis** on the memory size and performance **requirements** has been carried out. The motivation behind this study is to give a specification for eventual future implementations.
The Memory component stores the Current Coding Unit and the Reference frame, which in this verification environment are loaded by the simulation script. Considering a search window of 64×64 pixels, **not** the **whole** Reference frame must be stored in memory, but just a sub-block whose size is, in the worst-case 320×320 pixels (obtained by

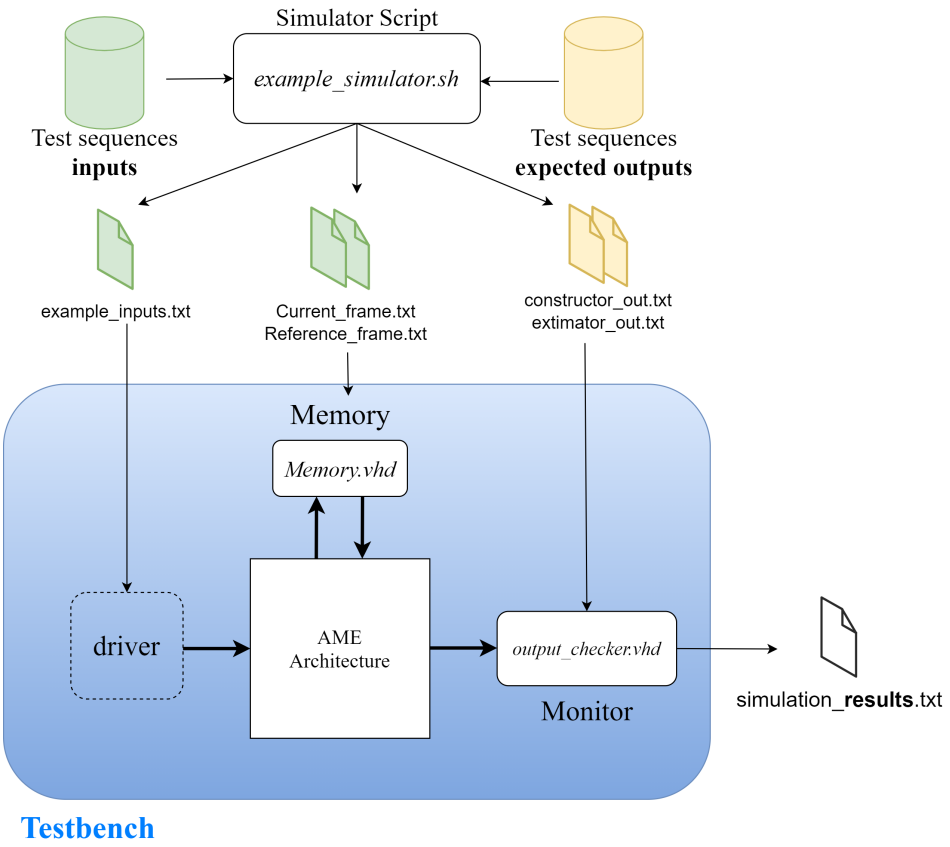

Figure 4.1. The verification environment.

translating a 64×64 block of 64 pixels in all the possible directions). Therefore, only $(320 \cdot 320)\,\mathrm{B} = 102400\,\mathrm{B}$ are required. Additional $(64 \cdot 64)\,\mathrm{B} = 4\,\mathrm{kB}$ must be considered, to store a Coding Unit in the worst case.
In total, a 128 kB memory is necessary. Moreover, the system timing has been designed assuming a reading operation latency of 1 clock cycle. Therefore, the **reading time** must be sufficiently low for the memory to sustain the hardware accelerator's pace. For the proposed architecture, the clock frequency is (evaluated later in this chapter) in the order of 350 MHz, which is considered a reasonable clock frequency requirement for a main memory.
About **parallelism** specification, the system is designed to process 8 pixels at a time (four from the Reference frame, 4 from the Current CU). This means that the memory output parallelism must be of 64 bit.

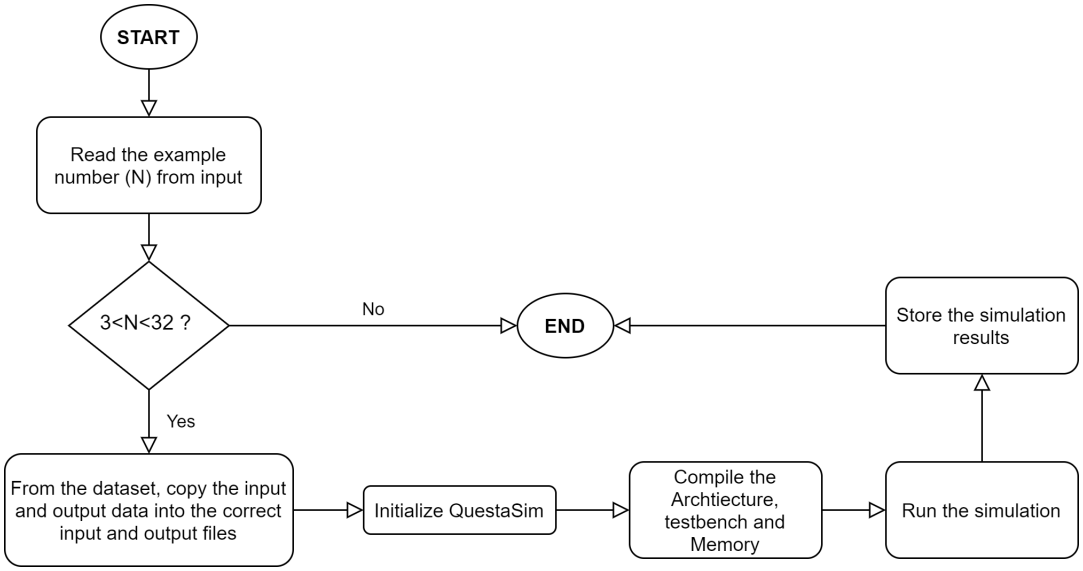

Figure 4.2. The simulator script flow chart.

4.1.3 Monitor

The Monitor (*output_checker.vhd*) reads the architecture's outputs and compares them with the expected output prepared by the simulator script. The comparison results are stored in an output file called *results_N.txt*, where N is the example identifier.

4.1.4 Verification Results

With a verification environment set up to perform a simulation on a single test sequence, the logical verification of the circuit over all the thirty test sequences (table C.1) is straightforward. The script in report 4.1 is exploited: it **loops** over all the example cases and, at each iteration, checks whether all the outputs are matched or not.

Report 4.1. The scripts `multiple_examples_simulator.sh` which loops the script simulator over the 30 test sequences.

```
#!/bin/bash
#Run "example_simulator.sh" for the examples in the interval "firstExample to
↪  lastExample"; check the correctness.
firstExample=3
lastExample=32
waitingTime=10 #time to wait for the simulations to terminate, in seconds
errors=0 #indicates if there have been errors
echo "Starting the simulations from example $firstExample to $lastExample"
for exampleNum in $(seq $firstExample $lastExample) ; do
        eval "./example_simulator.sh $exampleNum"
```

```
10          sleep $waitingTime
11          curResultsFileName="../tb/results/results_ex$exampleNum.txt"
12          lastLine=$(tail -n 1 $curResultsFileName)
13          if [[ $lastLine == "SIMULATION ENDED SUCCESSFULLY" ]] ; then
14                  echo "Simulation for example $exampleNum terminated
                    ↪  successfully."
15          else
16                  echo "Simulation for example $exampleNum terminated with
                    ↪  ERRORS."
17                  errors=1
18          fi
19  done
20  if [[ $errors -eq 0 ]] ; then
21          echo "All the simulations completed successfully."
22  else
23          echo "Some simulations terminated with ERRORS."
24  fi
```

The verification process using this method results in all the simulation **successfully completed**. In report D.2, the output of the `multiple_examples_simulator.sh` is shown. In the following report 4.2, the results file of a single test (example 14 from the test sequences) is shown. The fields meaning are explained by the adjacent comments.

Report 4.2. The logical verification simulation results over the 30 test seqences.

```
EXPECTED OUTPUT, OUTPUT, TEST RESULT
52,52,OK          ##<--- Twelve Constructor distorsion
776,776,OK        #       values
20826,20826,OK    #
23722,23722,OK    #
257,257,OK        #
2141,2141,OK      #
24641,24641,OK    #
28697,28697,OK    #
29,29,OK          #
785,785,OK        #
24125,24125,OK    #
27053,27053,OK    #----------
-5,-5,OK          #<--- Constructed MV0_h
42,42,OK          #                 MV0_v
-90,-90,OK        #<--- Constructed MV1_h
46,46,OK          #                 MV1_v
-5,-5,OK          #<--- Constructed MV2_h
-6,-6,OK          #----------       MV2_v
5103,5103,OK      #<--- Extimator SAD_1
4972,4972,OK      #<--- Extimator SAD_2
-5,-5,OK          #<--- Extimated MV0_h
42,42,OK          #               MV0_v
-90,-90,OK        #<--- Extimated MV1_h
46,46,OK          #               MV1_v
-5,-5,OK          #<--- Extimated MV2_h
-6,-6,OK          #-----          MV2_v
SIMULATION ENDED SUCCESSFULLY
```

4.2 Synthesis and Area, Timing and Power evaluation

Having the RTL description of the Affine Motion Estimation (AME) Architecture and the appropriate design libraries, the circuit synthesis is straightforward. For this thesis work, the design is synthesized for an ASIC (*Application-Specific Integrated Circuit*) using the **45nm Nangate Open Cell Library** and the **DesignWare** Library.
The first one is an open-source standard-cell library developed by the American company *Nangate* and donated to aid university research programs. Currently, it contains more than 100 different cells which come in multiple strength variants. The second one, instead, is a library from Synopsis which includes a collection of IP (intellectual properties) integrated into the Design Compiler environment, which is the software used in this work. Thanks to DesignWare, the synthesizer automatically detects the most common basic blocks (e.g. multipliers, adders) and optimizes the synthesis to meet the constraints imposed by the user.

4.2.1 The Synthesis process

The design logic Synthesis maps the logic RTL description of a digital circuit to physical components taken from the design libraries. The result is a **netlist** containing cells that together perform the same operations as the HDL description, with known characteristics in terms of area, delay, and parasitic elements. Therefore, having the architecture netlist, some estimations about its **area** occupation, critical path **delay** and **power** consumption can be carried out.
The synthesis process consists of the following sequence of steps:

1. **Prepare the Design Compiler**. In this case an initialization script is sourced as follows
   `source /eda/scripts/init_design_vision`.
   It is also important that the **setup** file is located in the correct directory. This file, called `.synopsys_dc.setup`, is essential to inform the Design Compiler about the design libraries and where to find them.

2. **Run the Design Compiler**. Here this is done using the following command
   `dc_shell -f synopsys_commands_custom.tcl`
   where `synopsys_commands_custom.tcl` is a script containing all the required commands to perform the subsequent steps automatically. The `-f` option programs the Design Compiler in shell mode to run the script. Report D.3 shows the mentioned Tcl script for the synthesis automation.

3. **Read VHDL source files**. In this step, the VHDL source files providing the circuit RTL description are read. This phase is dived into an **analysis** and an **elaboration** phase. In the first one, the files are checked for syntactic and semantic errors (the syntactic rules depend on the language used).
   In the second one, the HDL description is expanded. All instances of all entities

are represented as unique objects. Moreover, this phase involves the evaluation and propagation of ports, constants, and generics present in the description.

4. **Apply constraints**. Among all the possible constraints that Design Compiler allows the user to apply to a design, only some constraints on the clock signal and the output load are exploited. In this work, there is a clock signal period set as 10 ns for the first synthesis, with a clock uncertainty of the 7% and a maximum delay of 0.5. The output load is a *BUF_X4* ("fan-out of four" load).

5. **Start the synthesis and save the results**. Issuing the `compile` command, the synthesis is performed. After this step, some results can be saved, like the **timing report**, and the **area report** (showing information about the critical path delay and area occupation). Reports 4.3 and 4.4 show an excerpt of the timing and area report respectively when the clock period is $T_{clk} = 2.93ns$.

Report 4.3. Synthesized AME Architecture **timing** report (excerpt)

```
[...]
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : AME_Architecture_expanded
Version: R-2020.09-SP2
Date   : Tue Jul  5 11:16:08 2022
****************************************
 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top

  Startpoint: constructing_unit/Datapath/D_v_sample/Q_int_reg[1]
              (rising edge-triggered flip-flop clocked by MY_CLK)
  Endpoint: constructing_unit/Datapath/D_v_sq_sample/Q_int_reg[26]
            (rising edge-triggered flip-flop clocked by MY_CLK)
  Path Group: MY_CLK
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  AME_Architecture_expanded
                     5K_hvratio_1_1        NangateOpenCellLibrary

  Point                                                   Incr       Path
  ------------------------------------------------------------------------
  clock MY_CLK (rise edge)                                0.00       0.00
  clock network delay (ideal)                             0.00       0.00
  constructing_unit/Datapath/D_v_sample/Q_int_reg[1]/CK (DFFR_X1)
                                                          0.00 #     0.00 r
  constructing_unit/Datapath/D_v_sample/Q_int_reg[1]/Q (DFFR_X1)
                                                          0.10       0.10 r
  [...]

  constructing_unit/Datapath/D_v_sq_sample/D[26] (REG_N_N27_1)
                                                          0.00       2.82 r
  constructing_unit/Datapath/D_v_sq_sample/Q_int_reg[26]/D (DFFR_X1)
                                                          0.01       2.82 r
  data arrival time                                                  2.82

  clock MY_CLK (rise edge)                                2.93       2.93
```

```
clock network delay (ideal)                                    0.00       2.93
clock uncertainty                                             -0.07       2.86
constructing_unit/Datapath/D_v_sq_sample/Q_int_reg[26]/CK (DFFR_X1)
                                                               0.00       2.86 r
library setup time                                            -0.03       2.83
data required time                                                        2.83
-------------------------------------------------------------------------------
data required time                                                        2.83
data arrival time                                                        -2.82
-------------------------------------------------------------------------------
slack (MET)                                                               0.00
```

From the timing report what emerges is that the **critical path** is across the 26-bit multiplier which belongs to the **constructor** datapath. It is a predictable result since it is the most complex block in the whole circuit which is not internally pipelined. With all the constraints set, the clock period for which the slack is zero is $T_{clk} = 2.93ns$. This means the **maximum clock frequency** for this architecture is $f_{clk} = 341\,\text{MHz}$.

Report 4.4. Synthesized AME Architecture **area** report

```
****************************************
Report : area
Design : AME_Architecture_expanded
Version: R-2020.09-SP2
Date   : Tue Jul  5 11:16:08 2022
****************************************

Library(s) Used:

    NangateOpenCellLibrary [...]

Number of ports:                        11755
Number of nets:                         23491
Number of cells:                        10828
Number of combinational cells:           7598
Number of sequential cells:              2719
Number of macros/black boxes:               0
Number of buf/inv:                       1812
Number of references:                       5

Combinational area:               9516.948023
Buf/Inv area:                     1042.454002
Noncombinational area:           14372.512424
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:                 23889.460447
Total area:                 undefined
```

The **area occupation** of the circuit is $23889\,\mu\text{m}^2$, with 60% on sequential blocks and 40% on combinational blocks. This proportion is probably because there is a generous amount of register file and pipe register throughout the whole circuit, which results to waste more area than the combinational part itself.

4.2.2 Power consumption estimation

The power consumption estimation process is carried out jointly using *QuestaSim* and *Design Compiler*. It is divided into the following steps

1. **Compile** the netlist produced in the synthesis process and the testbench which includes it, using *QuestaSim*. It is important to **link** the cells library of using the option `-L` and the delay file (*.sdf* file) generated by the Design Compiler. QuestaSim gives as output a *value-change-dump* (*vcd*) file, which contains information about the circuit's switching activity.

   Note: When estimating the power consumption of a digital circuit, the **switching activity** plays an important role. It depends on the input test sequence: for different stimuli during this phase, slightly different results may be obtained. For the sake of completeness, it is pointed out that the test sequence **4** from table C.1 is used for this step.

2. **Convert** the *vcd* file to **saif**, which is a format that Design Compiler can handle. This is done using the `vcd2saif` command.

3. **Perform** power consumption **estimation** with Synopsys Design Compiler. The software loads the netlist and the *saif* file and produces a power report.

Report 4.5 shows the generated power report for the whole AME Architecture.

Report 4.5. Synthesized AME Architecture **power** report (excerpt).

```
[...]
****************************************
Report : power
        -analysis_effort low
Design : AME_Architecture_expanded
Version: R-2020.09-SP2
Date   : Tue Jul  5 11:55:29 2022
****************************************

Library(s) Used:

    NangateOpenCellLibrary [...]


Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top

Design          Wire Load Model             Library
------------------------------------------------
AME_Architecture_expanded
                        5K_hvratio_1_1     NangateOpenCellLibrary


Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000ff
    Time Units = 1ns
    Dynamic Power Units = 1uW    (derived from V,C,T units)
    Leakage Power Units = 1nW

  Cell Internal Power  =   1.2190 mW   (88%)
  Net Switching Power  = 169.4913 uW   (12%)
                         ---------
Total Dynamic Power    =   1.3885 mW  (100%)
Cell Leakage Power     = 435.1093 uW
```

Power Group	Internal Power	Switching Power	Leakage Power	Total Power	(%)	Attrs
io_pad	0.0000	0.0000	0.0000	0.0000	(0.00%)	
memory	0.0000	0.0000	0.0000	0.0000	(0.00%)	
black_box	0.0000	0.0000	0.0000	0.0000	(0.00%)	
clock_network	0.0000	0.0000	0.0000	0.0000	(0.00%)	
register	1.0694e+03	14.9613	2.0288e+05	1.2872e+03	(70.59%)	
sequential	0.0000	0.0000	0.0000	0.0000	(0.00%)	
combinational	149.6187	154.5309	2.3223e+05	536.3785	(29.41%)	
Total	1.2190e+03 uW	169.4922 uW	4.3511e+05 nW	1.8236e+03 uW		

There are three power contributions extracted with the `report_power` command and shown in report 4.5:

1. **Internal Power**: It is the short-circuit power. This is due to the direct connection between the power supply and the voltage reference that is created when, during a transition, some transistors inside the circuit are conducting at the same time.

2. **Switching Power**: Due to the charging and discharging of internal nodes and load capacities.

3. **Leakage Power**: Due to the sub-threshold current of the transistors, which does not assume the ideal null value when this is off ($V_{gs} = 0$).

The first two power types sum up to form the **Dynamic** Power, while the last one is associated with the **Static** one. What happens in this circuit is that the Internal power is the **dominant** power consumption component (with its 1.22 mW, 67% of the total), followed by the Leakage power, with 435 µW, 24% of the overall consumption. The last component is the Switching power, which covers the 9%.

4.2.3 The Constructor and Extimator separate contributions

As shown in figure 3.1, the AME Architecture is composed of two distinct hardware accelerators (the "Constructor" and the "Extimator"), which are independent of each other. The Constructor could be replaced with the VTM software construction algorithm exposed in 2.2.1. In the same way, the Extimator could be replaced with the exact estimation method shown in the same section.
With such degrees of freedom, it is possible to choose which component to use depending on the specific application. Therefore it is important to have an idea of **how** the two components **separately** contribute to the whole AME Architecture characteristics.

To obtain the area, timing, and power report of the two components individually, the same processes reported in sections 4.2.1 and 4.2.2 are performed on the Constructor and Extimator. Figure 4.3 shows the maximum clock frequency, area, and power consumption estimation results.

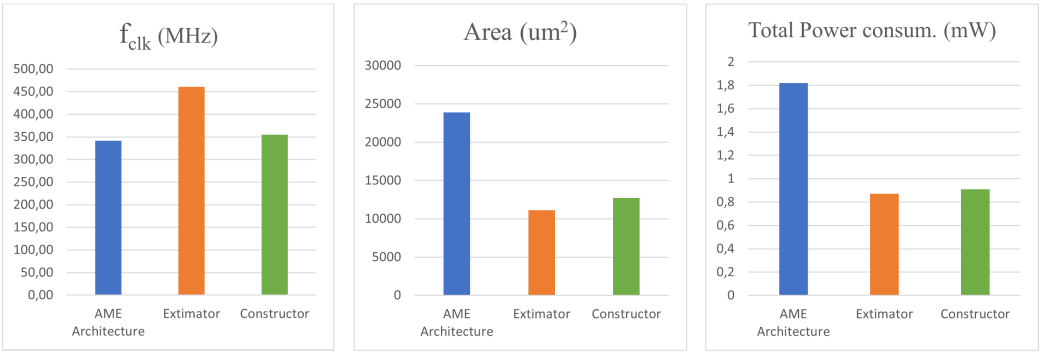

Figure 4.3. Maximum clock frequency, area and power consumption of the proposed architecture compared to its main sub-blocks.

Separate considerations must be done about the obtained estimations.

- About the **clock frequency**, being the critical path entirely in the Constructor, the maximum f_{clk} in this latter is nearly the same as the whole architecture's one. In the Extimator, instead, the circuit speed **increases** of the 35%. This is a considerable improvement and could be a good motivation to use only the Extimator if the system requirement is execution speed.
- From the **area** occupation side, the Extimator, and the Constructor cover the 46% and 54% of the total area. This means that the two components have roughly the **same cost** in terms of area. This is an interesting parameter to consider when choosing which one of the components is essential or not.
- For what concerns the total **power consumption**, both the Extimator and Constructor consume half the amount of the total architecture. Again, the two components achieve roughly the same results, bringing to the conclusion that they could be considered components with **similar complexity**.

Table 4.1 summarizes the results obtained for the three blocks.

	T_{cp}(ns)	f_{clk}(MHz)	$Area$(µm)	Total Power (mW)
Extimator	2,17	460,83	11099	0,87
Constructor	2,82	354,61	12723	0,91
AME Architecture	2,93	341,30	23889	1,82

Table 4.1. Critical path delay; maximum clock frequency; area and power consumption of the proposed architecture and its main sub-blocks.

4.3 Architecture encoding performance

The circuit performance in terms of encoding capability can be analyzed starting from the maximum clock frequency evaluated during the synthesis process. The main metric on which this property is measured is the **maximum** video **resolution** and **framerate** that the architecture can process without slowing down the VTM software.
As already mentioned in the previous sections, the Affine Motion Estimation (AME) process is not the only algorithm in the VVC encoding chain. As *Pakdaman et al.* in [13] have measured, the AME is 17% of the ME complexity, which is, in turn, the 47% of the total encoding complexity. Therefore, it is calculated that Affine Motion Estimation is approximately 8% of the total VVC encoder computational burden.

The **clock frequency requirement** variation with the video stream parameters is calculated under the following conditions.

- Assume a video stream with resolution $F_w \times F_h$ and framerate fr.
- All the blocks are coded with Affine Motion Estimation.
- The block size is fixed at $CU_w \times CU_h$ and equal for all the Coding Units.
- Assume that the minimum clock frequency required increases *linearly* with the computational complexity.

Under the previous constraints, the minimum frequency f_{min} required to the AME Architecture is:

$$f_{min} = \frac{1}{0.08} \frac{F_w \cdot F_h}{CU_w \cdot CU_h} \cdot T_{ext} \cdot fr \tag{4.1}$$

Where T_{ext} is the Extimator latency shown in equation 3.8. Using the relation in 4.1, f_{min} has been calculated with three different CU sizes and with the resolution and framerate examples from the Common Test Conditions [12]. The results are shown in table 4.2.

Notice how f_{min} depends on the CU sizes. In this thesis work, the simplified algorithm is implemented in a way that allows for a maximum frequency of 341 MHz. Considering

			CU: $(16 \cdot 16)$	CU: $(32 \cdot 32)$	CU: $(64 \cdot 64)$
F_w	F_h	fr	f_{min}	f_{min}	f_{min}
2560	1600	30	378,00	225,00	200,25
1920	1080	60	382,73	227,81	202,75
1920	1080	50	318,94	189,84	168,96
1920	1080	30	191,36	113,91	101,38
1920	1080	24	153,09	91,13	81,10
1280	720	60	170,10	101,25	90,11
1280	720	30	85,05	50,63	45,06
1280	720	20	56,70	33,75	30,04
1024	768	30	72,58	43,20	38,45
832	480	60	73,71	43,88	39,05
832	480	50	61,43	36,56	32,54
832	480	30	36,86	21,94	19,52
416	240	60	18,43	10,97	9,76
416	240	50	15,36	9,14	8,14
416	240	30	9,21	5,48	4,88

Table 4.2. Clock frequency requirement on the AME Architecture (f_{min}), calculated with three different CU sizes.

the results in table 4.2, in the worst case where all the Coding Units are 16×16 sized, the maximum frame rate and resolution supported by the architecture is 1920×1080 @ $50 fps$. Whether this result cold be sufficient or not, depends on the particular application considered.

4.3.1 Comparison with other architectures for VVC

In this thesis dissertation, pre-existing works for VVC have been analyzed from scientific literature (sections 1.5 and 2.5). For some of them, an hardware implementation is produced, and results in terms of area, throughput and cost are reported. Table 4.3 summarizes these features and compares them with the proposed architecture's ones.

Work	Algorithm	f_{clk}(MHz)	Throughput	P_{tot}	Cost	Target
[40]	ME	435	$1920 \times 1080@88fps$	467.93 mW	37.6 kgates	ASIC 90 nm
[16]	ME	227	$1920 \times 1080@47fps$	320.18 mW	-	FPGA
[17]	T/Q	576	$3840 \times 2160@86fps$	-	-	FPGA
[18]	T/Q	250	-	69.5 mW	-	ASIC 65 nm
[19]	T/Q	600	$4096 \times 2160@48fps$	-	96849 μm^2	ASIC 28 nm
[20]	T/Q	164	-	-	-	FPGA
[21]	T/Q	187	-	13.10 mW	90k eq. gates	ASIC 90 nm
[22]	T/Q	143	$3840 \times 2160@64fps$	-	5.2 kgates	SoPC 40 nm
[41]	LF	367	$4096 \times 2160@120fps$	-	369 kgates	ASIC 65 nm
Prop.	AME	341	$1920 \times 1080@50fps$	1.82 mW	23889 μm^2	ASIC 45 nm

Table 4.3. Comparison between pre-existing architectures for VVC and the proposed one.

Some papers do not report all the architecture characteristics. This is why some data is missing.
What emerges is that most of the works handle the "Transform and Quantization" (T/Q) stage. That is because DCT (and in some cases DST) are commonly studied algorithms, that exist since the first video coding standard.
The operating **clock frequencies** are in the range of $[164 \div 600]$ MHz: the proposed architecture, with its 341 MHz, is placed in the middle.
From **power** estimations, the proposed work dissipates 86% less than the second less consuming one. The motivation can be seen in the algorithm simplicity, that implements only one part of a lager block that is Motion Estimation.
This low complexity is highlighted also by the low **cost** in terms of area.

Chapter 5

Conclusions

In this thesis dissertation, a simplified Affine Motion Estimation algorithm and its hardware implementation are presented. First of all, the motivation behind the work is carried out: the increasing **computational** resources **requirement** of the new encoding standard, Versatile Video Coding. The idea of relieving part of the software complexity burden by simplifying the algorithm and introducing a hardware accelerator could be exploited in other works by acting on other complex blocks. For this purpose, the VVC complexity analysis carried out in chapter 1.2 could be exploited. An ideal point of arrival could be a **complete** hardware implementation of the VVC encoder, where every block is low-power and high-performance optimized.

As already mentioned, the proposed algorithm and architecture are composed of two distinct and independent sub-parts. The first one is the **construction** algorithm (hardware-implemented as the "Constructor"): it provides part of the Motion Vector (MV) candidates which will participate in the motion estimation process. The second one is the **estimation** algorithm (hardware-implemented as the "Extimator"), which chooses from the candidates' list the MVs that are the result of the Motion Estimation (ME) process: the Motion information for the current Coding Unit. The better the ME algorithm, the lower the residual information to be transmitted: this is what distinguishes a good Estimation method from a bad one. Therefore, in order to evaluate the goodness of the proposed algorithm, the amount of residual information after the motion estimation is evaluated. But this is not a standard procedure. An interesting work that could be performed on the algorithm could be to **implement** it in the **VTM software** and extract its performance in terms of Bjontegaard Delta Bit-Rate (BDBR). That is what is done in the other works on AME, which act on different parts of the algorithm. This property makes the proposed method **orthogonal** to the cited works ([34, 4, 37, 38]). It is possible to integrate more than one solution from Table 2.4 and evaluate the performance of the new combined algorithm. Eventually, many hardware implementations could come out, each one with its advantages and flaws.

The **hardware implementation** of the algorithm is based on a strong hierarchical structure. Everything starts from the smallest inverter and is organized up to the largest AME Architecture, which includes the Constructor and the Extimator, each one with

its Datapath, Control Unit, and signal adapters. However, this is not the only possible implementation of the algorithm. Some changes can be applied to the signals' timing, the component structure, and so on. These architectural modifications, in **future works** could be applied with the aim of improving one of the three main system characteristics: *cost* in terms of area, *performance* in terms of critical path delay (T_{cp}) and *power consumption*. Usually what happens is that, when improving one of the three parameters, there is another one that degrades. A trivial but effective strategy is to **reduce** the architecture **critical** path **delay** acting on the 26-bit multiplier in the constructor, because the critical path starts and ends in this component. One method to do so is to internally pipeline it, or reduce its parallelism (and thus complexity) by using a pipelined structure similar to Extimator's "MULT1" component described in section 3.2.1. Both the solutions increase the circuit latency, reducing at the same time the system's performance. An analysis of the new implementation delay should be carried out to understand whether this could bring advantages or not. Ideally, if the Constructor's T_{cp} became lower than the Extimator's one, the new operating frequency could be this latter component's one, that is $f_{clk} \simeq 460\,\text{MHz}$, a growth of the 35%. This would allow the AME Architecture to elaborate streams up to 2560×1600 @ $30fps$, according to the considerations in section 4.3.

Another possible architectural modification is related to the circuit input **timing**. In the implementation proposed here, a new Motion Estimation process can start only when the preceding one is completed. This is a waste in terms of execution time since, thanks to the used pipelining technique, the first components in the pipeline are available even when the elaboration is not completed. To improve the execution speed, a "non-stop" execution mode could be implemented. The drawback, in this case, would be the added complexity in both the Extimator's Datapath and Control Unit. Another idea for future work could be to produce a hardware implementation of the exact VTM Affine Motion Estimation algorithm shown in section 2.2.1 and to compare its characteristics with the proposed simplified one.

Finally, this study is concluded with an **analysis** on the **encoding performance**, having in mind the possibility to integrate this component into a system with a processor running the VTM software. It is shown that the architecture is supposed to support the main encoding algorithm in the elaboration of video streams with resolution up to 1920×1080 @ $50fps$. However, this estimation is obtained by applying some constraints to ease the frequency requirement calculation. Therefore, one last idea for future work could be to perform tests with the VTM software to verify if the proposed architecture really achieves this performance or if it behaves worse, or even better.

Appendix A

Video Coding overview

A.1 Image representation in Video Streams

Before introducing the very first video coding standard, it is essential to show how images are digitally represented in a video stream.
To understand deeply why the standard currently in place is this one, the history of TV broadcasting must be dug up. The first electronic black-and-white television sets were produced and released commercially in the late 1930s [42]. Back then picture was generated by exciting the phosphor of a screen using an electron beam driven by the video signal. The path that modulated electron beam traces was linear and it is shown in figure A.1.

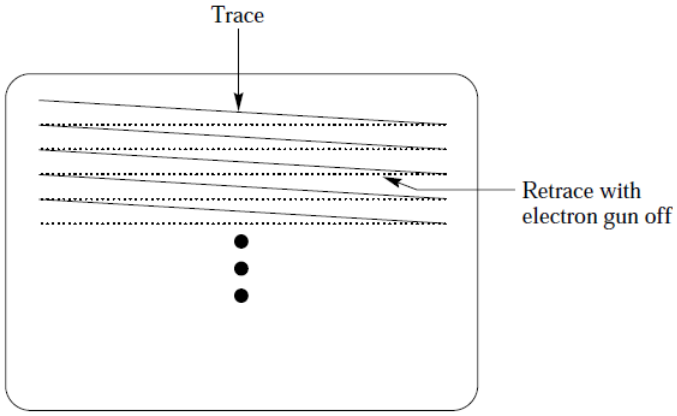

Figure A.1. The path traversed by the electron beam in a television. From [5]

In the late 1950s, the first colour television based on a system designed by RCA (Radio Corporation of America) began commercial broadcasting [43]. Of course, there is more information to be transmitted in this case, and in fact, instead of a single electron gun,

there are three of them, each one exciting a different colour among red, green and blue. This change in the way of representing pictures led shortly to two main problems. The first one was the increased bandwidth due to new colour information that has to be transmitted. The second one was backward compatibility. In fact, by the advent of colour television systems, there were still many black-and-white TV sets users, which television stations did want to reach too using the same signal of color TV ones.
Both problems found a solution when the **composite color signal** was introduced. It is composed of the **luminance** signal (luma), which is the same used for the intensity in black-and-white television and the two **chrominance** (chroma) signals, used to represent color.
In the late 1990s, digital television started spreading [44], the mentioned signals began to be sampled and stored digitally. This new transition was regulated by International Consultative Committee on Radio (CCIR) recommendation 601-2. In this standard the sampling rates for the cited operation were defined, all multiple of the base frequency 3.725 MHz. Each sample is arranged on a rectangular array, becoming a **pixel** of the picture.
The three components of the composite color signal are indicated with the letters Y,C_b,C_r to indicate the luma and the two chroma components respectively. These signals can be collected with a different sampling rate, a triplet of integers indicates which choice has been made and it may vary from one standard to another. A typical instance is the 4:2:0, which means that the sampling frequency of the chroma components is one-half of the luma one. This particular choice is addressed to as **chroma subsampling**, it allows to reduce the bandwidth requirement by halving the number of chroma samples while achieving quite the same image quality. This is because the human eye is more sensitive to intensity than color variation [45].

A.2 Video Encoders basic blocks

Leaving out the first standard H.120, which is "not used much" [1], the first DCT-based video coding standard is the ITU-T H.261 standard. Its block diagram is shown in figure A.2. This first, simple standard is useful as a base for describing the basic principles of video coding.

A.2.1 Motion-compensated prediction

In most video sequences there is little change in content between one frame and the next one. This phenomenon is what can be called **temporal redundancy**. Keeping in mind that the main task of Video Coding is to reduce the amount of information to be stored and transmitted, the idea behind motion-compensated prediction is to take advantage of this redundancy by using a frame to generate a **prediction** for the next one using **Motion Estimation**. If the decoder knows how to perform the prediction, always having stored the previous frame it will easily recover the current one using **Motion Compensation** exploiting the prediction information sent by the encoder. This encoded data is more lightweight than an entire frame.

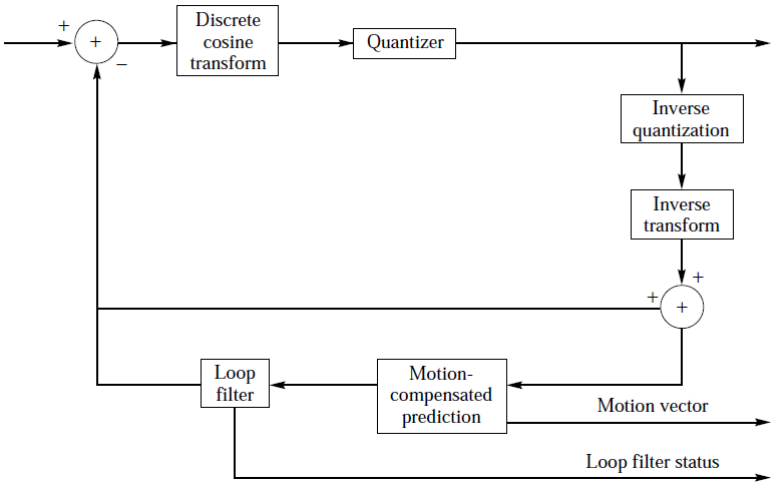

Figure A.2. Block diagram of the ITU-T H.261 encoder, from [5].

In practice, this "prediction information" consists of one or more vectors called **motion vectors** (MVs), which simply inform the decoder about the movement of some pixel blocks. Having this information, the decoder applies the motion vectors to the previously decoded frame. Of course, this is not enough to reconstruct faithfully a frame, since what can happen to elements in a frame is more complex than a simple movement (change of brightness, appearance or disappearance of objects and so on). To improve the frame reconstruction accuracy another piece of data is transmitted along with the MVs, the "prediction error" or **residual**.
The residual is the difference between the motion-compensated reference frame and the current frame. This is what is actually sent as information into the stream. If the prediction is sufficiently good, the motion-compensated picture is quite similar to the one to be encoded, leading to a very low-content residual and as consequence a reduced amount of data out of the video encoder.
It is not mandatory to perform always Motion Estimation/Compensation. There may be some cases in which it is not applied and the residual coincides with the current frame to be encoded. This may happen, for example, when there is an abrupt change of scene in a movie. If the two consecutive frames of the previous and next shot are totally different, there is no way to identify the movement of objects. Motion prediction is useless in such an occurrence. In this case, the encoder is said to be working in **intra mode**, otherwise, it's in **inter mode**.

A.2.2 Discrete Cosine Transform

Since H.261 the coded residuals are not represented directly with their pixel intensity, they are **transformed** using transforms which may change among standards. The most used ones represent the signals in the **frequency** domain. Indeed image, like most information,

is a signal that varies with a certain parameter (in this case, *space*). The speed of variation of the signal is strictly related to its frequency components. If it was possible to extract these components it would be allowed to cut out the higher ones, reducing the payload while keeping almost the same visual quality. This is because the human eye is not capable of perceiving very fast variations of intensity in a frame.
In H.261 as well as most of the following standards, the **discrete cosine transform** (DCT) is exploited. It gets its name from the fact that the rows of the NxN transform matrix C are obtained as a function of cosines [5].

$$[C]_{i,j} = \begin{cases} \sqrt{\frac{1}{N}} \cos \frac{(2j+1)i\pi}{2N} \; i = 0, j = 0,1, ..., N-1 \\ \sqrt{\frac{2}{N}} \cos \frac{(2j+1)i\pi}{2N} \; i = 1, j = 0,1, ..., N-1 \end{cases} \tag{A.1}$$

The DCT of an image can be computed by matrix product with C. The result represents how much each frequency component is present in the picture. The coefficients of the result can also be applied to the **bases matrices** to reconstruct the original image.
The base matrices for a NxN transform are a set of NxN matrices that can be summed up to reconstruct a frame if they are weighted using the coefficient of the DCT of an image. In figure A.3, an example of an $8x8$ set is reported.

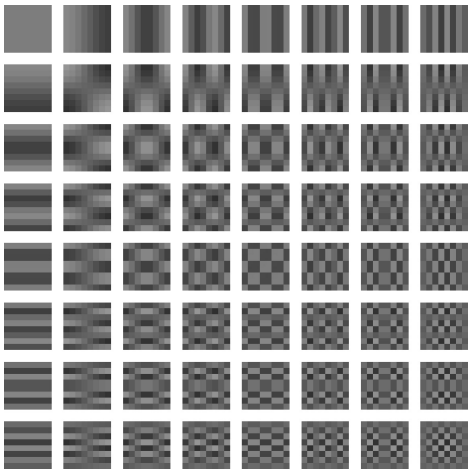

Figure A.3. The bases matrices for the DCT. From [5]

To show how image transform and inverse transform work and how DCT can be exploited to reduce the amount of information in an image while preserving quality, an interesting example can be shown. Assume the current frame to be the $8x8$ *pixel* smile reported in figure A.4, on the left. Using the *Matlab* tool, its DCT-II is computed and reported in table A.2.

$$\begin{pmatrix} 560{,}73 & 471{,}64 & 390{,}67 & 548{,}36 & 534{,}21 & 416{,}48 & 514{,}77 & 546{,}94 \\ 82{,}18 & 60{,}80 & -21{,}57 & 99{,}82 & 87{,}32 & -15{,}21 & 129{,}99 & 94{,}36 \\ 152{,}28 & -28{,}11 & -183{,}05 & -149{,}56 & -163{,}80 & -173{,}48 & -8{,}14 & 146{,}19 \\ -169{,}15 & -129{,}43 & 58{,}62 & 106{,}41 & 95{,}86 & 89{,}73 & -112{,}97 & -164{,}79 \\ -13{,}43 & 197{,}98 & 146{,}72 & 9{,}54 & 14{,}49 & 141{,}42 & 166{,}87 & 1{,}76 \\ 120{,}79 & 29{,}23 & -0{,}37 & 72{,}80 & 74{,}68 & 13{,}61 & -37{,}08 & 88{,}17 \\ -46{,}24 & 74{,}40 & 102{,}22 & 59{,}27 & 76{,}65 & 117{,}01 & 43{,}17 & -34{,}15 \\ -36{,}24 & 70{,}43 & -37{,}81 & 21{,}29 & 26{,}38 & -42{,}31 & 20{,}69 & -23{,}16 \end{pmatrix} \tag{A.2}$$

Performing the inverse transform of the DCT coefficients allows for reconstructing accurately the original image, having the information about each frequency component. Graphically, what happens is that each base in figure A.3 is multiplied by the respective coefficient in table A.2, then each $8x8$ block is summed up to produce the original "smile" image.
Having explicitly shown each frequency component, it is possible to cut out the highest ones and obtain the matrix in A.3.

$$\begin{pmatrix} 560{,}73 & 471{,}64 & 390{,}67 & 548{,}36 & 534{,}21 & 416{,}48 & 514{,}77 & 546{,}94 \\ 82{,}18 & 60{,}80 & -21{,}57 & 99{,}82 & 87{,}32 & -15{,}21 & 129{,}99 & 94{,}36 \\ 152{,}28 & -28{,}11 & -183{,}05 & -149{,}56 & -163{,}80 & -173{,}48 & -8{,}14 & 146{,}19 \\ -169{,}15 & -129{,}43 & 58{,}62 & 106{,}41 & 95{,}86 & 89{,}73 & -112{,}97 & -164{,}79 \\ -13{,}43 & 197{,}98 & 146{,}72 & 9{,}54 & 14{,}49 & 141{,}42 & 166{,}87 & 1{,}76 \\ 120{,}79 & 29{,}23 & -0{,}37 & 72{,}80 & 74{,}68 & 0 & 0 & 0 \\ -46{,}24 & 74{,}40 & 102{,}22 & 59{,}27 & 76{,}65 & 0 & 0 & 0 \\ -36{,}24 & 70{,}43 & -37{,}81 & 21{,}29 & 26{,}38 & 0 & 0 & 0 \end{pmatrix} \tag{A.3}$$

Performing the anti-transform (IDCT-II, *Inverse Discrete Cosine Transform - Type II*) of this latter, the "smile" in figure (A.4, on the right) is obtained.

Figure A.4. The $8x8$ *pixel* smile, scaled 2x. On the left, the original figure; on the right the same image after cutting out the highest frequency components

Even though the two figures are not identical, the result obtained may be sufficient considering the reduced amount of information needed to represent it. The number of coefficients in matrices A.2 and A.3 are still the same, it could be argued that there is no payload reduction. The advantage of the second matrix is that the last coefficients are all equal to zero: the **quantization and coding** block is in charge of exploiting this property to increase the compression of images.

A.2.3 Quantization and Coding

The matrix of DCT coefficients of the transformed frame (like the one in A.2) has to be manipulated to reduce the amount of information to be transmitted.
The transform operation does not introduce error when reconstructing the image. Ideally, applying the DCT on an image and the IDCT on the result would result in the same starting image. The operation issued in this block, instead, does introduce an error. It consists in dividing each coefficient by a number (called **quantization parameter (QP)**) and rounding the results in a way that is different from one standard to the next one. In H.261, the QP can change among 32 possible values [5], depending on how much each frequency component is considered important in the final reconstruction. If a DCT coefficient is smaller than the quantization parameter, its value could be rounded to zero. Usually, this is the case with higher frequency components, which are placed in the bottom-right positions of the matrix.
After the quantization operation, the matrix is encoded in a 1-D array to be transmitted or stored. The order the coefficients are collected follows a particular path called **zig-zag scan** which may slightly vary among the different coding standards. An example of zig-zag scan is shown in figure A.5 [6].

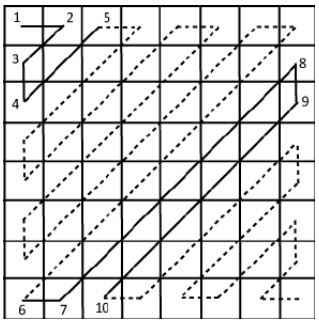

Figure A.5. The *zig-zag scan* which arranges the quantized DCT coefficients in a 1-D matrix. From [6]

This kind of path allows collecting a large number of zeroes consecutively. Taking the matrix example A.3, the last elements would be lined up as

$$[...,1.76,0,0,26.38,0,0,0,0,0,0] \tag{A.4}$$

This sequence is fed to a block called **Coder**. Its purpose is to code this stream in order to remove the redundancies, thus reducing the amount of information. For example, in H.261 the nonzero coefficients are coded along with the number, or **run** of coefficients quantized to zero. Moreover, the 20 most commonly occurring combinations of (run, label) are coded with a single variable-length codeword. This last principle is the one on which the **Entropy Coder** works. The idea is to represent the most common labels with the shortest symbols in order to reduce the total amount of information.

A.2.4 Frame Partitioning

All the operations described so far (*Motion Compensation*,*Tansform and Quantization*), are not performed on the whole frame at the same time during the encoding/decoding process. Since H.261, the frames in video streams are split into **blocks** of small dimensions. In the first standard, there was only one choice for the block dimensions, which was 8×8. This technique is called **frame partitioning** and is used to find the optimum in terms of computation cost. Performing the DCT on a frame of size 1920×1080 which is the resolution of standard HD format would be very intense in terms of computational complexity. This is why the frame is divided into blocks. Blocks size may not be too small, since it increases the number of transform operations to be performed. There is no fixed solution in finding the best block dimension, a tradeoff has to be done depending on the encoder structure and performance.
While the video coding standards became more complex, frame partitioning evolved too. An interesting example is the HEVC standard, where the block division follows a *tree structure*, in which the largest possible block is the *Coding Tree Unit* (CTU) with a size chosen by the encoder which may range from 16×16 to 64×64 [46]. The CTU can be divided into *Coding Units* (CUs) according to a quad-tree-based algorithm.
The advantage of this strategy is to adapt the video content to the granularity of the operation to be applied. For example, high-detail areas may need a fine granularity in Motion Prediction, since each detail could move in a different direction. Areas that are uniform in terms of intensity may have very few, low-frequency DCT coefficients, so it is profitable to include them in the same large block. In figure A.6 an example of frame partitioning in HEVC is shown. It can be noticed how regions with higher details are associated with smaller Coding Units.

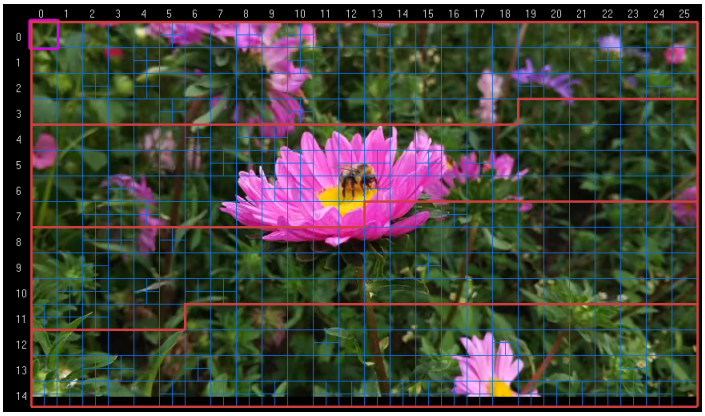

Figure A.6. An example of Frame Partitioning in HEVC

A.2.5 Loop Filter

The use of frame partitioning described in section A.2.4 may lead to discontinuities of the prediction error at the edges of coding units. As already mentioned in section A.2.2, fast variation of intensity leads to large values for the frequency coefficients, thus increasing the transmission rate. This problem is solved by smoothing the pixel intensity of the coding units using a spatial filter called the **loop filter**. The coefficients of the filter may change among the different coding standards, but its purpose remains the same.

A.2.6 Random Access Capability

One of the most important concepts in video coding is the use of residuals, differences between two frames, which are sent to the encoder since they are more lightweight than the whole frames. What the encoder performs is to apply those differences to the picture it already recovered to reconstruct the video stream. The only problem with this approach is that, to reconstruct a given frame, all the previous frames are needed. This means that it's not possible to decode a video sequence starting from any frame. On the contrary, this capability is mandatory in services like broadcast, where the users do not necessarily want to access the video stream starting from the first frame.
The ability to start decoding a video sequence at, or close to, some arbitrary point is called **random access capability**. This is what was introduced in the MPEG-1 video standard by the ISO/IEC MPEG (*Moving Picture Experts Group*). The idea proposed is quite simple, and it relies on the use of frames that are periodically coded without any reference to past frames, called **I frames** (*Intra coded frames*). The number of I frames in a video sequence affects the random access capability and the transmission rate in the opposite direction. Since the I frames do not exploit information of the adjacent ones, their compression ratio is quite high, they are the heaviest frames of a video sequence. To reduce the transmission rate, it's important to reduce the number of I frames in a video sequence. However, if there are few I frames, it is not possible to start decoding a video sequence close to any given frame, so the random access capability is negatively affected. There are two extreme cases related to this choice. The first one, where all the frames are I frames, where it is possible to view the video sequence starting from any frame, but the amount of information in the coded stream is high. The second one, where only the first frame is of type I, where the transmission rate is the lowest, but the video sequence can only be viewed starting from the beginning. Of course, there is a trade-off, in this case, to be considered.
The frames that are not of the I type are divided in **P** (*Predictive coded*) and **B** (*Bidirectionally predictive coded*) ones. The P frames are coded using motion-compensation prediction applied to the closest I or P frame, thus their compression efficiency is better than the one of the I kind. The B frames, instead, rely on the closest I or P frame, which is a past frame or a **future** frame. This improves further the compression efficiency but complicates unavoidably the coding standard. If there are frames that, to be displayed, require that future frames are decoded, the decoding order is different from the order in which the pictures are displayed. Thus, since MPEG-1 a distinction between the *decoding* order (or **bistream order**) and the **display order** was made. The frames in video

streams are organized in *group of pictures* (**GOP**), defined as "the smallest random access unit in the video sequence" [5]. In a GOP, the first I frame is either the first one or is preceded by B frames that use motion-compensated prediction only from this I frame. An example of GOP is shown in figure A.7. In tables A.1 and A.2, instead, two possible display and bitstream order for the GOP are shown.

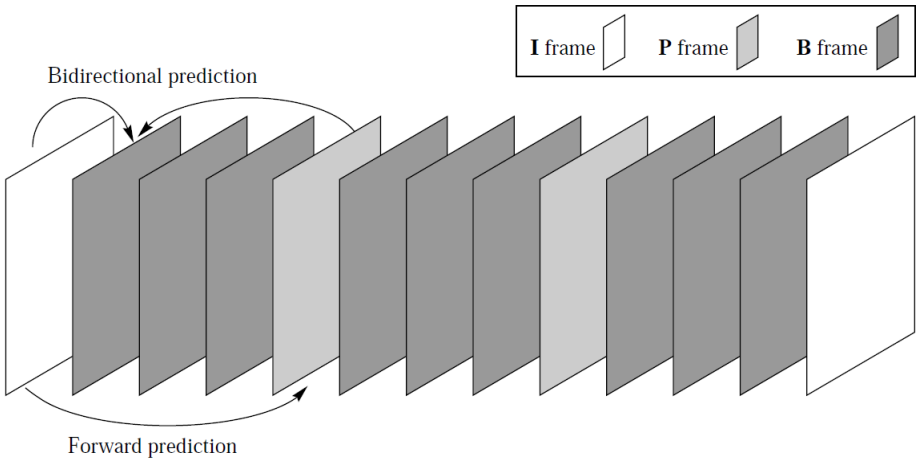

Figure A.7. A possible arrangement for a group of pictures. From [5]

I	B	B	B	P	B	B	B	P	B	B	B	I
1	2	3	4	5	6	7	8	9	10	11	12	13

Table A.1. Display order for the GOP in figure A.7

The GOP structure is an important setup for the encoder since it is a trade-off between the high compression efficiency of motion-compensated coding and the versatility of random access capability of intra-only coding.

A.2.7 Profiles and Levels

Considering all the stages of video coding along with their complexity, it shows up that encoding and decoding a video stream may require a powerful processing system. Currently, there is a large variety of internet-connected devices, each of them with its computational capability [8]. This variety of devices arose the need for a coding standard suitable to each device, which could change its performance requirements according to the available resources. This is what led the "Moving picture experts group" to introduce in their new standard MPEG-2 the concept of *profiles* and *levels*, which is still in use so far.
Profiles allow for reducing or increasing the algorithmic complexity of the coding standard. The first profiles included in MPEG-2 are worth mentioning since they are similar to the ones still used in modern standards. The first one is the *simple*, which avoids the

I	P	B	B	B	P	B	B	B	I	B	B	B
1	5	2	3	4	9	6	7	8	13	10	11	12

Table A.2. Bitstream order for the GOP in figure A.7

use of **B** frames since they are the most complex to be processed. Then there is the **main** profile, which includes the use of all the blocks as they were described so far. The next one is the *snr-scalable*, where two different streams are transmitted. The first one contains the essential information to reconstruct the encoded stream, while the second one allows enhancing the quality of the reconstruction. This solution is called **layered approach** and can be used to transmit also some additional frames to increase the framerate (*temporally scalable* profile) or to improve picture resolution (*spatially scalable* profile). This is a versatile approach since low-performance devices can reconstruct the video using only the first stream, while high-performance ones could improve de video quality including the second one in the elaboration. Notice that the order of profiles is important since each of them is also capable of decoding the other ones. For example, a decoder designed for the *spatially scalable* profile could decode a video encoded in *main*.
Levels allow to reduce or increase the resolution or the frame-rate, to tune the burden on the processing elements which work on the video stream. In MPEG-2, the levels with their relative frame sizes are: *low*, with 352×240, *main* with its 720×480, *high 1440* with its 1440×1152, *high*, corresponding to 1920×1080. All of these levels are defined with a framerate of 30 frames per second.

A.2.8 Intra prediction

In the ITU-T Recommendation H.264, ISO/IEC MPEG-4 Part-10 also called **AVC** (*Advanced Video Coding*), an innovative way of processing the I frames was introduced. Previous standards had no way to introduce predictive coding when considering Intra-frames. Consequently, the number of bits required to store I frames was substantially higher than for the other kind of pictures. To reduce the bitrate, the AVC introduced **spatial prediction modes** for Intra prediction. This means that in a single frame, the pixel in a block of a certain dimension (in AVC, 4×4) are predicted using the value of boundary pixels belonging to the same frame. Depending on the direction of prediction, a number is associated with the prediction mode. Figure A.8 shows the modes available in AVC.

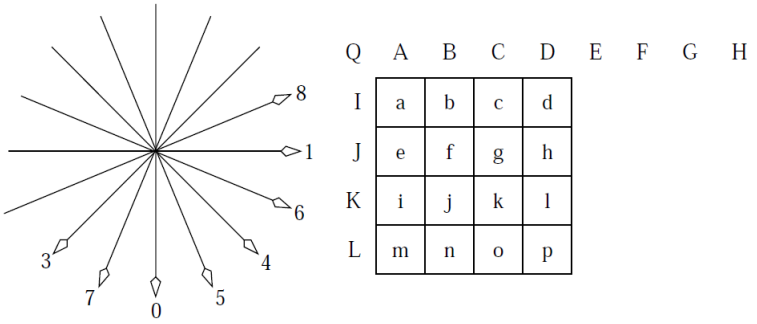

Figure A.8. Prediction modes in AVC. In small letters the pixel of the block to be encoded, in capital letters the neighbor pixels from the same frame. From [5]

There is an additional mode which is not shown in figure A.8, the **DC mode** associated to the number 2. In this mode, all the pixels of the block to be encoded are predicted using the average of left and top neighbor pixels.

Appendix B

Proposed model Matlab Implementation

B.1 Candidate construction

```
%% candidate_search (example 31)
% This script selects the best candidate(s) for the affine AMVP prediction

clear

%Example-specific parameters

h=64;   %Current block height
w=16;   %Current block width

%We are computing the third element in a triplet, the
%MV'S2 described in paper (2), which imposes that x=0 while y=h
x=0;
y=h;

% Neighboring blocks motion vectors
%Group 0
mv0_h(1)=32;   %A2 x component
mv0_v(1)=8;    %A2 y component

mv0_h(2)=32;  %B2 x component
mv0_v(2)=0;    %B2 y component

mv0_h(3)=32;   %B3 x component
mv0_v(3)=8;    %B3 y component

%Group 1
mv1_h(1)=32;   %B1 x component
```

```
29  mv1_v(1)=8;     %B1 y component
30
31  mv1_h(2)=32;    %B0 x component
32  mv1_v(2)=8;      %B0 y component
33
34  %Group 2
35  mv2_h(1)=-32;  %A1 x component
36  mv2_v(1)=-8;     %A1 y component
37
38  mv2_h(2)=32;    %A0 x component
39  mv2_v(2)=8;    %A0 y component
40
41  D_min=121237;
42  D_min2=121238;
43
44  %Rows C(1) and C(2) contain the first and second best candidates respectively
45  %The third row contains the third vector for which the distortion D is the
46  %minimun one
47  C=zeros(2,3);
48
49  % Best candiadate search
50  for i=1:length(mv0_h)
51      for j=1:length(mv1_h)
52          %Calcolo qui MVS2' perché nel prossimo step, con gli MVS2
53          %calcoleremo solo le distortion
54          mv2p_h(i,j)= -bitshift((mv1_v(j)-mv0_v(i)),log2(h/w),'int16') +
    ↪  mv0_h(i);
55          mv2p_v(i,j)= bitshift(+(mv1_h(j)-mv0_h(i)),log2(h/w),'int16') +
    ↪  mv0_v(i);
56          for k=1:length(mv2_h)
57              D=sqrt((mv2p_v-mv2_v(k))^2+(mv2p_h-mv2_h(k))^2);
58              if D<D_min
59                  %Set the best candidate couple as the current one,
60                  %the previous best couple becomes the second one
61                  D_min2=D_min;
62                  D_min=D;
63                  C(2,1)=C(1,1);
64                  C(2,2)=C(1,2);
65                  C(2,3)=C(1,3);
66                  C(1,1)=i;
67                  C(1,2)=j;
68                  C(1,3)=k;
69              elseif D<D_min2
70                  %Check if this candidate couple is not the same as the best one
71                  if mv0_h(i)~=mv0_h(C(1,1)) || mv0_v(i)~=mv0_v(C(1,1)) ||
    ↪  mv1_h(j)~=mv1_h(C(1,2)) || mv1_v(j)~=mv1_v(C(1,2))
72                      %If they're not the same, you can update the second
73                      %best candidate
74                      D_min2=D;
```

```
                    C(2,1)=i; %Second best candidates
                    C(2,2)=j;
                    C(2,3)=k;
                end
            end
        end
    end
end
```

B.2 Affine Motion Estimation

```
%% AMC for canditates choice multiparameter model
%This model chooses between the different candidates the one whose SAD is
%lower than the others

close all
clear

%Add the scripts folder to the MATLAB path
addpath '.\YUV'
addpath '.\myMatlabLib'

%Acquire example name from input
prompt = {'Enter example file path:'};
dlgtitle = 'Example file';
dims = [5 100];
definput = {'.\AMC_examples_data\AMC_examples_data_ex26.xlsx'};
examplefile = inputdlg(prompt,dlgtitle,dims,definput);

%Load example data
data=readtable(char(examplefile));

%fame parameters
frame_h=data.frame_h(1);
frame_w=data.frame_w(1);
sixPar=data.sixPar(1);

%Block parameters
w=data.w(1);
h=data.h(1);
x0=data.x0(1);
y0=data.y0(1);

%Raw image extraction
numfrm=1;%Extract just one frame
```

```
35 startfrm=data.startfrm_cur(1);  %POC of the current frame
36 [Y]=yuv_import( char(data.file_cur(1)),[frame_w frame_h],numfrm,startfrm);
37 Curframe=cell2mat(Y);
38 figure('Name','Frame to be encoded (original file)')
39 colormap('gray');
40 image(Curframe)
41 %CU to be encoded
42 %NOTA: In Matlab le immagini, se indicate come matrici di pixel, hanno gli
43 %indici che si riferiscono prima alla coordinata y e poi alla x. Questo
44 %perche' si mette sempre prima l'indice della riga e poi quello della
45 %colonna!
46 CurCu=Curframe((y0+1):(y0+h),(x0+1):(x0+w));
47 figure('Name','CU to be encoded (original file)')
48 colormap('gray');
49 image(CurCu)
50
51 %Reference frame extraction
52 numfrm=1;
53 startfrm=data.startfrm_ref(1);
54 [Y]=yuv_import( char(data.file_ref(1)),[frame_w frame_h],numfrm,startfrm);
55 Refframe=cell2mat(Y);
56 figure('Name','Reference frame (prev. frame from the encoded file)')
57 colormap('gray');
58 image(Refframe)
59 %Same CU but from the reference frame
60 % RefCu=Refframe((y0+1):(y0+h),(x0+1):(x0+w));
61 % figure('Name','Same CU but from the reference frame')
62 % colormap('gray');
63 % image(RefCu)
64
65 %Representative Blocks
66 figure('Name','Reference blocks highlighted (original file)')
67 colormap('gray');
68 image(CurCu)
69 title('Current Coding Unit')
70 xlabel('pixel x (horizontal) position')
71 ylabel('pixel y (vertical) position')
72 hold on
73
74 for j=0:floor((h-1)/16) %Vertical control
75     for i=0:floor((w-1)/16) %Horizontal control
76         rectangle('Position',[0.5+16*i 0.5+16*j 4 4],'EdgeColor','r')   %Plot
    ↪  the first representative 4x4 block in a 16x16 block
77         rectangle('Position',[(0.5+12)+16*i 0.5+16*j 4 4],'EdgeColor','r')
    ↪  %Plot the second representative 4x4 block in a 16x16 block
78         rectangle('Position',[0.5+16*i (0.5+12)+16*j 4 4],'EdgeColor','r')
    ↪  %Plot the third representative 4x4 block in a 16x16 block
79         rectangle('Position',[(0.5+12)+16*i (0.5+12)+16*j 4 4],'EdgeColor','r')
    ↪  %Plot the fourth representative 4x4 block in a 16x16 block
```

```
80      end
81  end
82
83  %Plot the 16x16 blocks
84  for j=0:floor((h-1)/16) %Vertical control
85      for i=0:floor((w-1)/16) %Horizontal control
86          rectangle('Position',[0.5+16*i 0.5+16*j 16 16],'EdgeColor','b')   %Plot
    ↪  the first 4x4 block in a representative 16x16 block
87      end
88  end
89
90  %Applico trasformazione affine "semplificata"
91
92  cand_num=data.cand_num(1); %Number of candidates
93
94  %Candidates MV and Refframe initialization
95  Refframe_AMC=zeros(frame_h,frame_w,cand_num);
96  mv0_v=zeros(cand_num,1);
97  mv0_h=zeros(cand_num,1);
98  mv1_v=zeros(cand_num,1);
99  mv1_h=zeros(cand_num,1);
100 mv2_v=zeros(cand_num,1);
101 mv2_h=zeros(cand_num,1);
102 for i=1:cand_num
103     Refframe_AMC(:,:,i)=Refframe;        %Affine compensated reference frame
104         %CPMV 4-parameter
105         mv0_v(i)=data.mv0_v(i);
106         mv0_h(i)=data.mv0_h(i);
107         mv1_v(i)=data.mv1_v(i);
108         mv1_h(i)=data.mv1_h(i);
109     %If CMPV 6-parameter, add MV2 too
110     if sixPar==1
111             mv2_v(i)=data.mv2_v(i);
112             mv2_h(i)=data.mv2_h(i);
113     end
114 end
115
116
117 %Number of representatives in a 16x16 block
118 rep_num=4;
119 %Number of 16x16 blocks
120 block_num=(w*h)/256; %(Total n of pixels)/(pixels in a 16x16 block)
121 %SAD(n) contains the SAD for the n-th candidate
122 SAD=zeros(cand_num,1);
123 %Relative mv's (mvr) matrix
124 %First index: Candidate identifier (da 1 a cand_num)
125 %Second index: Representative identifier (da 1 a rep_num, typ: rep_num=4)
126 %Third index: 16x16 block identifier (da 1 a block_num)
127 %Fourth index: First coordinate: y [v], second coordinate: x [h]
```

```
128 mvr=zeros(cand_num,rep_num,block_num,2);
129 %Exact values
130 mvr_ex=mvr;
131
132 fxp_prec=4;%Alla fine 4 è la precisione totale, non ho approssimato
133 %con 3 anche se avrei potuto per motivi spiegati nel datapath
134
135
136 for curcand=1:cand_num
137     curbloc=0; %Current block index
138     %Motion vector matrices
139     for j=1:(h/16) %Vertical control
140         for i=1:(w/16) %Horizontal control
141             curbloc=curbloc+1;
142             for currep=1:rep_num
143                 %Compute x and y coordinates of the current block
144                 offset_x=0;
145                 if mod(currep,2)==0
146                     offset_x=12;
147                 end
148                 offset_y=0;
149                 if currep>2
150                     offset_y=12;
151                 end
152                 x=16*(i-1)+offset_x;
153                 y=16*(j-1)+offset_y;
154
    ↪ a_1=bitshift((mv1_v(curcand)-mv0_v(curcand)),-(log2(w)-4),'int16')/16;
    ↪ %a_v_hw
155
    ↪ a_2=bitshift((mv1_h(curcand)-mv0_h(curcand)),-(log2(w)-4),'int16')/16;
    ↪ %a_h_hw
156                 if sixPar==0
157
    ↪ b_1=bitshift(+(mv1_h(curcand)-mv0_h(curcand)),-(log2(w)-4),'int16')/16;
    ↪ %b_v_hw
158
    ↪ b_2=-bitshift((mv1_v(curcand)-mv0_v(curcand)),-(log2(w)-4),'int16')/16;
    ↪ %b_h_hw
159                 else
160
    ↪ b_1=bitshift(+(mv2_v(curcand)-mv0_v(curcand)),-(log2(h)-4),'int16')/16;
    ↪ %b_v_hw
161
    ↪ b_2=bitshift(+(mv2_h(curcand)-mv0_h(curcand)),-(log2(h)-4),'int16')/16;
    ↪ %b_h_hw
162                 end
163                 mvr_ex(curcand,currep,curbloc,1)=x*a_1 + y*b_1 +
    ↪ mv0_v(curcand); %mv_v exact_hw
```

```
164                 mvr_ex(curcand,currep,curbloc,2)=x*a_2 + y*b_2 +
    ↪ mv0_h(curcand); %mv_h exact_hw
165
    ↪ mvr(curcand,currep,curbloc,1)=round(mvr_ex(curcand,currep,curbloc,1)/16);
    ↪ %mv_v_hw
166
    ↪ mvr(curcand,currep,curbloc,2)=round(mvr_ex(curcand,currep,curbloc,2)/16);
    ↪ %mv_h_hw
167                 SAD(curcand)=SAD(curcand)+ sum(
    ↪ abs(Refframe((y0+1+y+mvr(curcand,currep,curbloc,1)):(y0+1+y+mvr(curcand,currep,curbloc,1)+3),
168                                 (x0+1+x+mvr(curcand,currep,curbloc,2)):
169                                 (x0+1+x+mvr(curcand,currep,curbloc,2)+3)) -
    ↪ CurCu(y+1:(y+4),x+1:(x+4))),'all');
170                 %Copy the original position of each block (graphical
171                 %reference)
172
    ↪ Refframe_AMC((y0+1+y):(y0+1+y+3),(x0+1+x):(x0+1+x+3),curcand)=CurCu(y+1:(y+4),x+1:(x+4));
173             end
174         end
175     end
176 end
177
178 %Plot the CurCu and the AMC connected by a red line
179 for curcand=1:cand_num
180     curbloc=0;
181     s1='AMC for candidate number:';
182     s2=num2str(curcand);
183     figure('Name',strcat(s1,s2))
184     colormap('gray')
185     image(Refframe_AMC(:,:,curcand))
186     title('TMC on the representative blocks')
187     xlabel('pixel x (horizontal) position')
188     ylabel('pixel y (vertical) position')
189     rectangle('Position',[(x0+0.5),(y0+0.5) w h],'EdgeColor','b')   %Plot the
    ↪ first 4x4 block in a representative 16x16 block
190     for j=0:floor((h-1)/16) %Vertical control
191         for i=0:floor((w-1)/16) %Horizontal control
192                         curbloc=curbloc+1;
193                         for currep=1:rep_num
194                                 offset_x=0;
195                 if mod(currep,2)==0
196                     offset_x=12;
197                 end
198                                 offset_y=0;
199                 if currep>2
200                     offset_y=12;
201                 end
202                                 %First representative 4x4 block in a 16x16
    ↪ block
```

```
203                                 first_edge_x=0.5+offset_x+16*i+x0;   %First (in
    ↪ alto a sx) edge of the Refframe's CurCu coordinates
204                                 first_edge_y=0.5+offset_y+16*j+y0;
205                                 first_edge_AMC_x=first_edge_x+mvr(curcand,currep,curbloc,2);
    ↪ %First (in alto a sx) edge of the Refframe's CurCu coordinates when AMC is
    ↪ applied
206                                 first_edge_AMC_y=first_edge_y+mvr(curcand,currep,curbloc,1);
207                                 rectangle('Position',[first_edge_x first_edge_y
    ↪ 4 4],'EdgeColor','r')
208                                 rectangle('Position',[first_edge_AMC_x
    ↪ first_edge_AMC_y 4 4],'EdgeColor','g')
209                                 line([first_edge_x+2,first_edge_AMC_x+2],
210                                 [first_edge_y+2,first_edge_AMC_y+2],'Color','red')
211                             end
212         end
213     end
214 end
215
216 %Results computation
217 [SAD_min,Best_candidate]=min(SAD(1:2));
218 SAD_max=max(SAD(1:2));
219 SAD_adv=(1-SAD_min/SAD_max)*100;
220 msgbox({'Best Candidate';num2str(Best_candidate);'SAD
    ↪ Advantage';strcat(num2str(SAD_adv,'%.2f'),'%')})
221
222 %Check if the VQ_best is different from the Matlab chosen one. If yes,
223 %perform the comparison
224 if isfield(table2struct(data),'VQ_best')
225     VQ_best=data.VQ_best(1);
226     if VQ_best~=Best_candidate
227         comp_offs_y(1)=data.comp_offs_y(Best_candidate);
228         comp_offs_x(1)=data.comp_offs_x(Best_candidate);
229         comp_offs_y(2)=data.comp_offs_y(VQ_best);
230         comp_offs_x(2)=data.comp_offs_x(VQ_best);
231         mv0_h_comp=[mv0_h(Best_candidate) mv0_h(VQ_best)];
232         mv0_v_comp=[mv0_v(Best_candidate) mv0_v(VQ_best)];
233         mv1_h_comp=[mv1_h(Best_candidate) mv1_h(VQ_best)];
234         mv1_v_comp=[mv1_v(Best_candidate) mv1_v(VQ_best)];
235         mv2_h_comp=[mv2_h(Best_candidate) mv2_h(VQ_best)];
236         mv2_v_comp=[mv2_v(Best_candidate) mv2_v(VQ_best)];
237         [coeffReq,nonZero,nonZero_gt10] =
    ↪ residual_compare(Refframe,CurCu,sixPar,mv0_h_comp, mv0_v_comp,mv1_h_comp,
    ↪ mv1_v_comp,mv2_h_comp, mv2_v_comp, comp_offs_x, comp_offs_y)
238         SAD_Ratio=SAD(Best_candidate)/SAD(VQ_best)
239     end
240 end
```

Appendix C

Estimating algorithm performance

C.1 Test sequences used

Ex. No.	Stream	POC	x_0	y_0	Par	CU_w	CU_h
3	RaceHorses_416x240_30.yuv	4	208	160	4	32	32
4	RaceHorses_416x240_30.yuv	4	256	64	4	16	32
5	VQ_sample_432x240.yuv	3	192	160	4	32	32
6	VQ_sample_432x240.yuv	28	192	96	4	16	32
7	RaceHorses_416x240_30.yuv	1	96	192	6	32	32
8	VQ_sample_432x240.yuv	12	160	128	6	32	16
9	VQ_sample_432x240.yuv	34	208	96	6	32	32
10	BasketballPass_416x240_50.yuv	49	224	104	6	16	16
11	BasketballPass_416x240_50.yuv	1	64	128	6	32	32
12	BasketballPass_416x240_50.yuv	1	64	96	6	32	32
13	BasketballPass_416x240_50.yuv	1	256	160	4	32	32
14	VQ_sample_432x240.yuv	1	256	64	6	64	32
15	VQ_sample_432x240.yuv	2	128	80	4	16	32
16	RaceHorses_416x240_30.yuv	1	256	176	4	32	16
17	RaceHorses_416x240_30.yuv	1	96	32	6	32	32
18	RaceHorses_416x240_30.yuv	2	128	160	4	32	32
19	RaceHorses_416x240_30.yuv	2	224	200	4	32	16
20	BasketballPass_416x240_50.yuv	5	64	200	4	64	16
21	BasketballPass_416x240_50.yuv	7	240	80	6	16	32
22	VQ_sample_432x240.yuv	3	224	128	6	32	32
23	VQ_sample_432x240.yuv	4	160	152	6	32	16
24	VQ_sample_432x240.yuv	4	160	128	4	32	16
25	BasketballPass_416x240_50.yuv	25	16	160	6	16	32
26	BasketballPass_416x240_50.yuv	29	32	144	4	16	16
27	RaceHorses_416x240_30.yuv	3	192	144	6	64	32
28	RaceHorses_416x240_30.yuv	5	136	168	4	16	16
29	BasketballPass_416x240_50.yuv	38	264	96	4	16	16
30	VQ_sample_432x240.yuv	25	192	128	6	64	64
31	BasketballPass_416x240_50.yuv	4	64	64	4	16	64
32	BasketballPass_416x240_50.yuv	13	64	0	6	32	64

Table C.1. Test cases used for the algorithm performance estimation (and later as test sequences for the hardware implementation logical verification).

C.2 Proposed Candidate construction algorithm performance

Example	VQ Comply	Any Constructed	Constr. Match	VQ C. w/o constr.
3	1	V	V	1
4	1	V	V	1
5	1	V	X	1
6	0	V	X	1
7	1	V	V	1
8	0	V	X	0
9	1	X	-	1
10	1	X		1
11	0	V	X	1
12	1	X	-	1
13	1	V	V	1
14	0	V	X	0
15	1	V	-	1
16	1	X	-	1
17	0	V	X	1
18	1	X	-	1
19	1	V	V	1
20	1	V	X	0
21	0	V	X	0
22	1	V	V	1
23	1	X	-	1
24	1	X	-	1
25	0	V	X	1
26	1	V	V	1
27	1	V	X	1
28	1	V	X	1
29	1	V	X	0
30	0	X	-	0
31	1	V	X	1
32	1	X	-	1
	Comply Ratio	Constructor usage	Cons. Match Ratio	Comply Ratio
	0,73	0,7	0,35	0,80

Table C.2. Some data about the Construction and proposed algorithm presented in section 2.3.1

C.3 Approximated AME algorithm performance

Report C.1. Matlab script for the evaluation of $E_{1,2}$ and $C_{1,2}$ (stored in variables *coeffReq* and *nonZero* respectively).

```
function [coeffReq,nonZero,nonZero_gt10] = residual_compare(Refframe,CurCu,
    sixPar, mv0_h, mv0_v, mv1_h, mv1_v, mv2_h, mv2_v, comp_offs_x, comp_offs_y)

%% Transformation of the Cu with both the candidates

%mvi_(v,h) sono i CPMV del
% 1: Miglior candidato secondo il mio algoritmo
% 2: Miglior candidato secondo il VTM

if sixPar==0
    mv2_h=zeros(2,1);
    mv2_v=zeros(2,1);
end

%Convert the CurCu into image in order to transform it
CurCu_imm=mat2gray(CurCu,[0 255]);
[CurCu_h,CurCu_w]=size(CurCu_imm);

for i=1:2
    movingpoints=[1 1; CurCu_w 1; 1 CurCu_h]; %Dove si trova la coda dei CPMV
    (ordine [x y])
    fixedpoints=[1+mv0_h(i)/16 1+mv0_v(i)/16; CurCu_w+mv1_h(i)/16
    1+mv1_v(i)/16; 1+mv2_h(i)/16 CurCu_h+mv2_v(i)/16]; %Dove si trova la punta
    dei CPMV

    tform = fitgeotrans(movingpoints,fixedpoints,"affine");

    CurCu_tran_imm=imwarp(CurCu_imm,tform);
    %Memorize separatedly the two CurCu tranformed. I cannot use
    %a 3-D matrix because the dimension of the two immages may be different
    if i==1
        CurCu_tran_1=CurCu_tran_imm*255;
    else
        CurCu_tran_2=CurCu_tran_imm*255;
    end
end

figure('Name','Transformation with Matlab Candidate')
colormap('gray')
image(CurCu_tran_1)

figure('Name','Transformation with VTM Candidate')
colormap('gray')
image(CurCu_tran_2)
```

```
41
42
43  %% Matlab candidate residual computation
44
45  %Fill the black squares with the values from the reference frame
46  [CurCu_AMC_1_h,CurCu_AMC_1_w]=size(CurCu_tran_1);
47  CurCu_AMC_1=CurCu_tran_1;
48  RefCu_1=zeros(CurCu_AMC_1_h,CurCu_AMC_1_w);
49
50  for j=1:(CurCu_AMC_1_h)%Scorri righe
51      for i=1:(CurCu_AMC_1_w)%Scorri colonne
52          if CurCu_AMC_1(j,i)==0
53              CurCu_AMC_1(j,i)=Refframe(j+comp_offs_y(1),i+comp_offs_x(1));
54          end
55              RefCu_1(j,i)=Refframe(j+comp_offs_y(1),i+comp_offs_x(1));
56      end
57  end
58
59  figure('Name','AMC with Matlab Candidate')
60  colormap('gray')
61  image(CurCu_AMC_1)
62
63  %Compute the residual
64  Residual_1=abs(CurCu_AMC_1-RefCu_1);
65  figure('Name','Residual with Matlab Candidate')
66  colormap('gray')
67  image(Residual_1)
68
69  %Compute what fraction of DCT coefficients contain 99% of the energy in the
    ↪  image
70  coeffReq(1)=dctCoeffNum(Residual_1,99);
71  %Number of nonzero elements
72  nonZero(1)=nnz(Residual_1);
73  nonZero_gt10(1)=nnz(round(Residual_1/10));
74
75  %% VTM candidate residual computation
76
77  %Fill the black squares with the values from the reference frame
78  [CurCu_AMC_2_h,CurCu_AMC_2_w]=size(CurCu_tran_2);
79  CurCu_AMC_2=CurCu_tran_2;
80  RefCu_2=zeros(CurCu_AMC_2_h,CurCu_AMC_2_w);
81
82  for j=1:(CurCu_AMC_2_h)%Scorri righe
83      for i=1:(CurCu_AMC_2_w)%Scorri colonne
84          if CurCu_AMC_2(j,i)==0
85              CurCu_AMC_2(j,i)=Refframe(j+comp_offs_y(2),i+comp_offs_x(2));
86          end
87              RefCu_2(j,i)=Refframe(j+comp_offs_y(2),i+comp_offs_x(2));
88      end
```

```
end

figure('Name','AMC with VTM Candidate')
colormap('gray')
image(CurCu_AMC_2)

%Compute the residual
Residual_2=abs(CurCu_AMC_2-RefCu_2);
figure('Name','Residual with VTM Candidate')
colormap('gray')
image(Residual_2)

%Compute what fraction of DCT coefficients contain 99% of the energy in the image
coeffReq(2)=dctCoeffNum(Residual_2,99);
nonZero(2)=nnz(Residual_2);
nonZero_gt10(2)=nnz(round(Residual_2/10));

%% CPMV comparison
%We compute the abs ratio and phase difference between the CPMVs. This
%gives us an idea of how much the extimated cpmvs are different from the
%VTM ones

[abs_mv_ratio(1),phase_mv_diff(1)] = mv_compare(mv0_h,mv0_v);
[abs_mv_ratio(2),phase_mv_diff(2)] = mv_compare(mv1_h,mv1_v);
if sixPar==1
    [abs_mv_ratio(3),phase_mv_diff(3)] = mv_compare(mv2_h,mv2_v);
    table({"Abs Ratio";"Phase Diff"},{abs_mv_ratio(1);phase_mv_diff(1)},{abs_mv_ratio(2);phase_mv_diff(2)},
        {abs_mv_ratio(3);phase_mv_diff(3)},'VariableNames',{'Var','MV0','MV1','MV2'})
else
    table({"Abs Ratio";"Phase Diff"},{abs_mv_ratio(1);phase_mv_diff(1)},{abs_mv_ratio(2);phase_mv_diff(2)},
        'VariableNames',{'Var','MV0','MV1'})
end
msgbox({'The model and VTM choices are different. Check the Command Window...'})
```

Report C.2. Function *dctCoeffNum(image,perc) called in script C.1* .

```
function coeffReq=dctCoeffNum(image,perc)

P=image;

%Compute the discrete cosine transform of the image data. Operate first along the rows and then along the columns.
```

```
Q = dct(P,[],1);
R = dct(Q,[],2);

%Find what fraction of DCT coefficients contain perc% of the energy in the
↪  image.
X = R(:);

[~,ind] = sort(abs(X),'descend');
coeffs = 1;
while norm(X(ind(1:coeffs)))/norm(X) < (perc/100)
   coeffs = coeffs + 1;
end

%The total number of coefficients in the dct
coeffNum=numel(R);
%The number of coefficients required to contain perc% of the energy in the
↪  image.
coeffReq=coeffs;
%fprintf('%d of %d coefficients are sufficient\n',coeffs,numel(R))
```

Example	VQ Comply	Parameters	E_r	C_r	No Compliance Reason
3	1	4	1,000	1,000	
4	1	4	1,000	1,000	
5	1	4	1,000	1,000	
6	0	4	0,718	0,936	Costructed Candidate Mismatch (CCM)
7	1	6	1,000	1,000	
8	0	6	1,009	0,987	CCM
9	1	6	1,000	1,000	
10	1	6	1,000	1,000	
11	0	6	1,044	1,000	Wrong choice (C instead of I)
12	1	6	1,000	1,000	
13	1	4	1,000	1,000	
14	0	6	1,083	0,982	CCM
15	1	4	1,000	1,000	
16	1	4	1,000	1,000	
17	0	6	1,022	0,994	Wrong choice (C instead of T)
18	1	4	1,000	1,000	
19	1	4	1,000	1,000	
20	1	4	1,000	1,000	
21	0	6	0,983	1,021	Wrong choice (T instead of C)
22	1	6	1,000	1,000	
23	1	6	1,000	1,000	
24	1	4	1,000	1,000	
25	0	6	1,245	1,026	CCM + Wr. Ch. (T instead of C)
26	1	4	1,000	1,000	
27	1	6	1,000	1,000	
28	1	4	1,000	1,000	
29	1	4	1,000	1,000	
30	0	6	1,406	0,985	Wrong choice (T2 instead of T1)
31	1	4	1,000	1,000	
32	1	6	1,000	1,000	
	Comply Ratio	"6" percentage	$E_{r,AVE}$	$C_{r,AVE}$	
	0,73	0,5	1,017	0,998	

Table C.3. Performance estimation of the proposed algorithm presented in section 2.3.2

Appendix D

Hardware Implementation

D.1 Constructor component

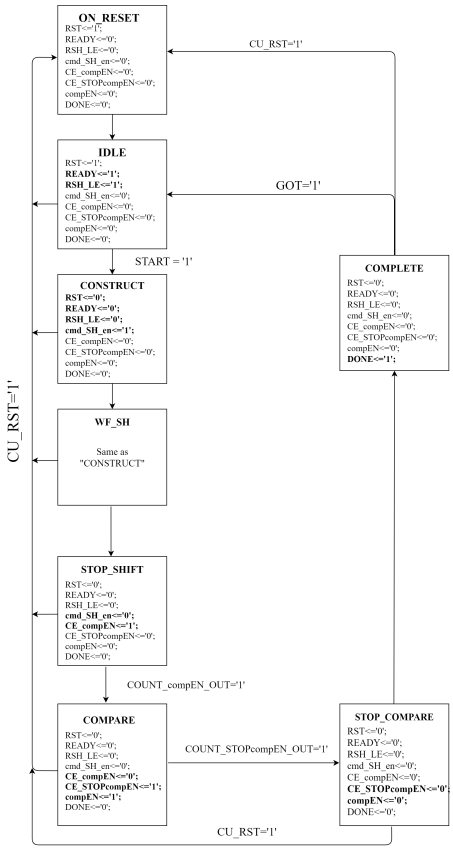

Figure D.1. Constructor's Control Unit state diagram.

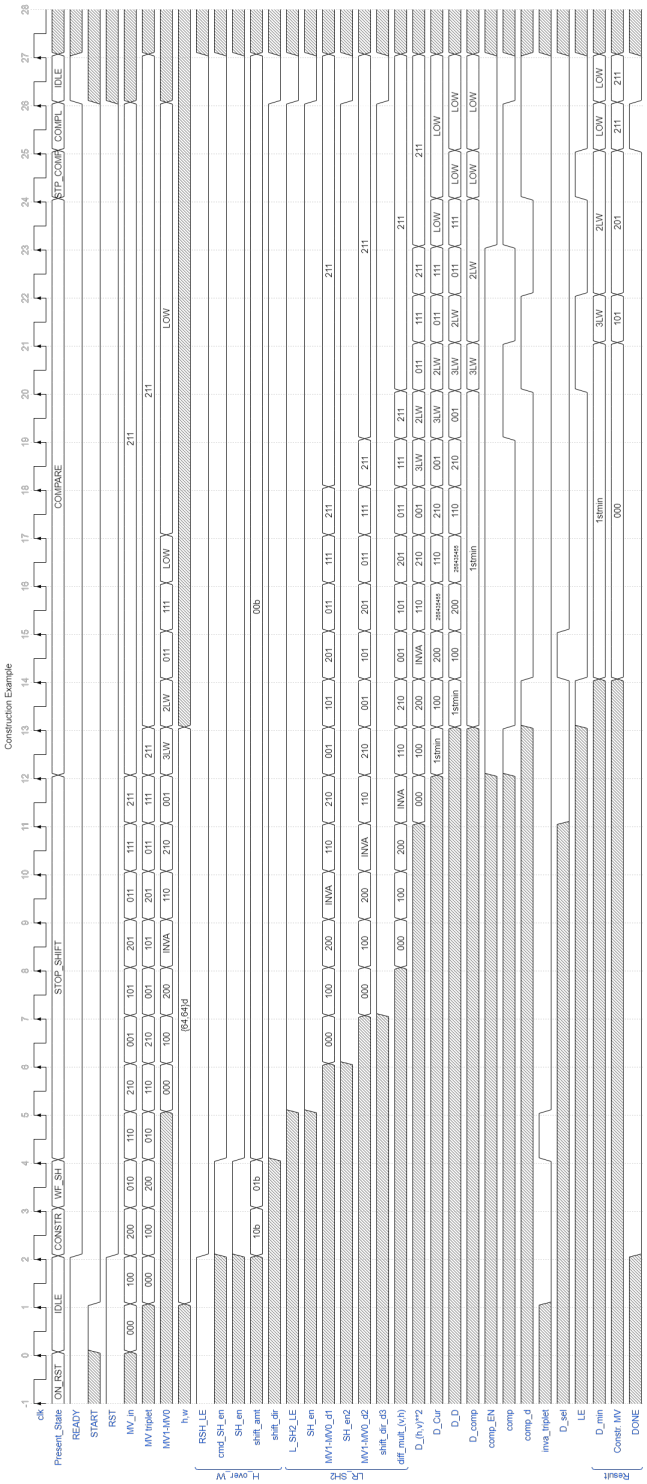

Figure D.2. Constructor Timing diagram.

D.2 Extimator component

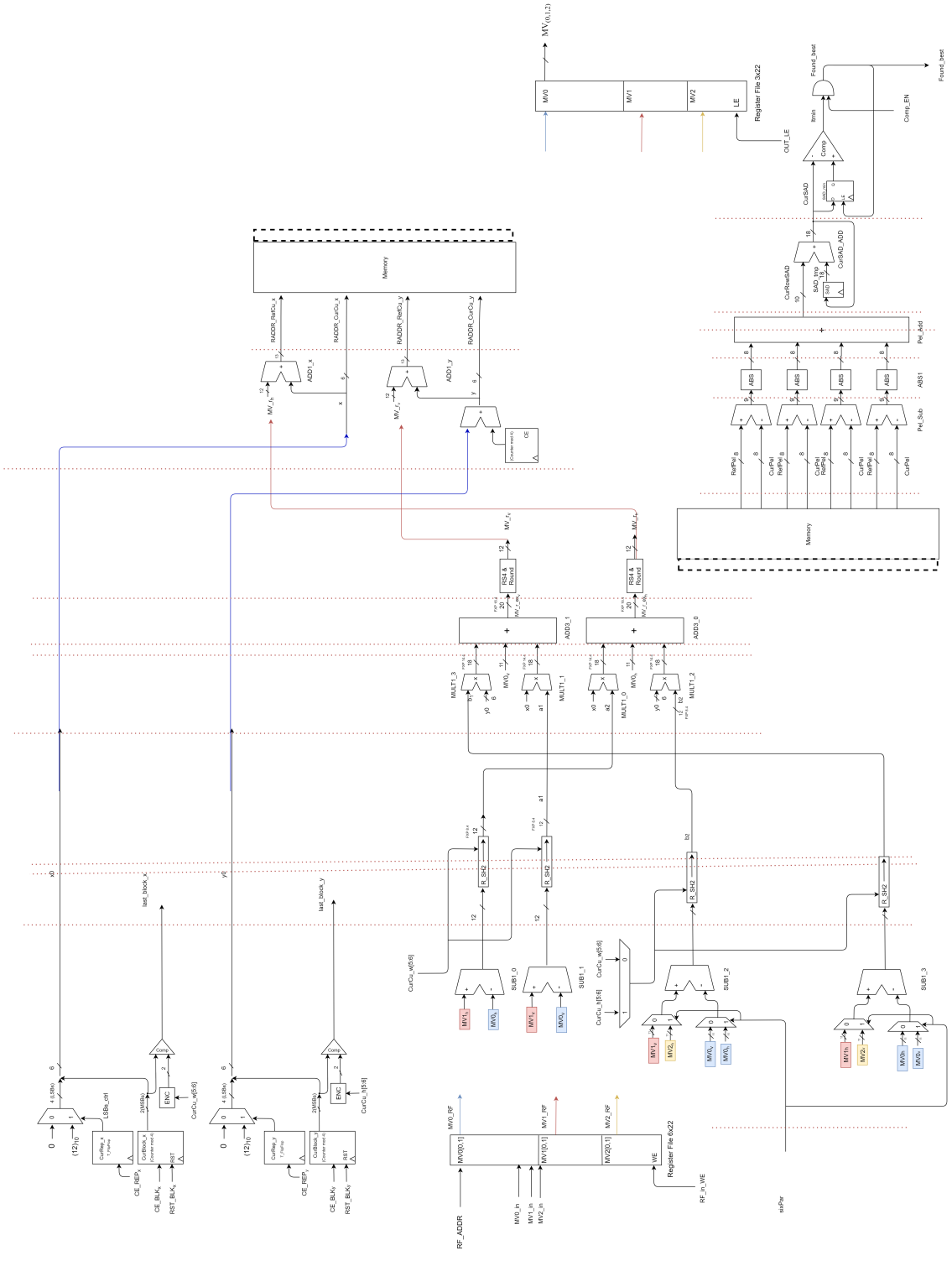

Figure D.3. Extimator Datapath. The dotted lines in red represent pipeline registers.

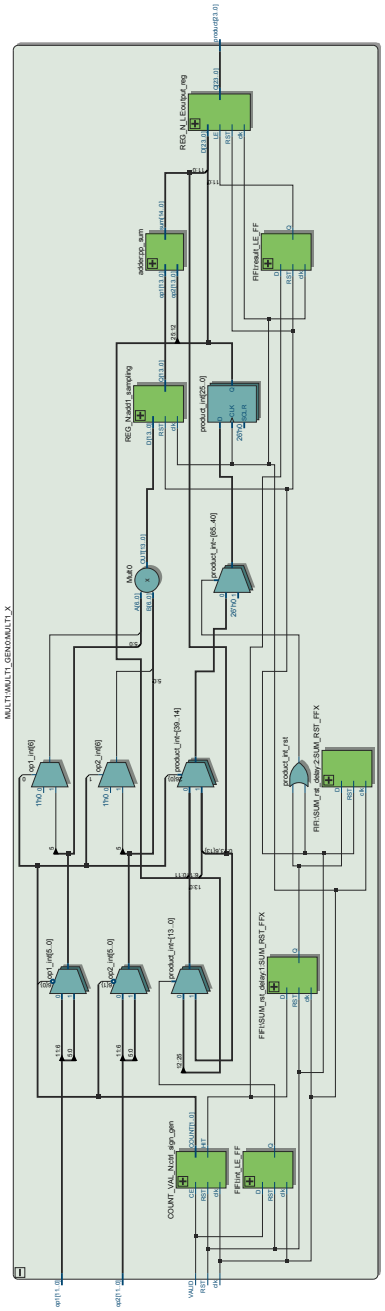

Figure D.4. The MULT1 component. RTL representation generated with *Quartus* design software [7].

Figure D.5. Extimator Timing diagram.

D.3 Verification and Synthesis

Report D.1. The Simulator script.

```
#!/bin/bash
#This script takes as input argument a single example number and starts the simulation
#of that example in batch mode (the commands to QuestaSim are given automatically)

#acquire the example number
if [ $# -ne 1 ]
then
        echo "Usage: $0 example_number"
        exit
fi
exampleNum=$1

#Prepare files for the example
#text files
filesToBeRenamed=("constructor_out/constructor_out" "extimator_out/extimator_out" "memory_data/Curframe" "memory_data/Refframe" "VTM_inputs/VTM_inputs")
for curfileToBeRenamed in ${filesToBeRenamed[@]} ; do
        curFileName=$curfileToBeRenamed"_ex"$exampleNum".txt"
        if [ -f "../tb/$curFileName" ] ; then
                cp "../tb/$curFileName" "../tb/$curfileToBeRenamed.txt"
        else
                touch "../tb/$curfileToBeRenamed.txt"
        fi
done
#vhd files
filesToBeRenamed=("memory/DATA_MEMORY")
for curfileToBeRenamed in ${filesToBeRenamed[@]} ; do
        curFileName=$curfileToBeRenamed"_ex"$exampleNum".vhd"
        if [ -f "../tb/$curFileName" ] ; then
                cp "../tb/$curFileName" "../tb/$curfileToBeRenamed.vhd"
        else
                touch "../tb/$curfileToBeRenamed.txt" #I leave it as txt so that compiler will generate an error.
                #This is made on purpose beacuse if the memory file is not present there actually IS a problem
                #and the compiler needs to be stopped
        fi
done

#open the compilation command file
exec 3< ./modelsim_ex/tb_AME_Architecture.txt

```

```
40 #Search and store the Log filename
41 while read LINE <&3 ; do
42         if [ "$LINE" == "--LOG_FILENAME" ] ; then
43                 read LINE <&3
44                 break
45         fi
46 done
47 LogFilename=$LINE
48
49 #Move in the simulation folder and delete the old log
50 SimPath="/home/thesis/costantino.taranto/git/AME_Architecture/sim"
51 cd $SimPath
52 if [ -f "$LogFilename" ] ; then
53         rm "$LogFilename"
54 fi
55 #Prepare the simulation tool
56 source /eda/scripts/init_questa
57 if [ -d "work" ]
58 then
59         rm -r work
60 fi
61 vlib work
62
63
64 #Compile the source files
65 while read LINE <&3 ; do
66         if [ "$LINE" == "--SOURCE" ] ; then
67                 break
68         fi
69 done
70
71 while read LINE <&3 ; do
72         if [ "$LINE" == "--TESTBENCH" ] ; then
73                 read LINE <&3
74                 #Note that there is a testbench
75                 isTb=1
76                 eval "$LINE>>$LogFilename"
77                 break
78         else
79                 eval "$LINE>>$LogFilename"
80         fi
81 done
82
83 #launch the simulation
84 if grep -Fq "** Error" $SimPath/$LogFilename
85 then
86         echo "Compliation error. See \"/sim/$LogFilename\" file for more."
87 else
88         if [[ $isTb -eq 1 ]] ; then
```

```
89                  read LINE <&3
90                  if [ "$LINE" != "--TB_NAME" ] ; then
91                          echo "Please provide a Testbench unit name."
92                  else
93                          read LINE <&3
94                          eval "vsim -do ../sim/batch_simulate.cmd work.$LINE"
95                          #eval "vsim work.$LINE"
96                  fi
97          else
98                  echo "Compilation completed succesfully. No Testbench has been
                    ↪  provided."
99          fi
100 fi
101
102 #store the results filename with the correct name
103 mv "../tb/results/results.txt" "../tb/results/results_ex$exampleNum.txt"
```

Report D.2. The logical verification simulation results over the 30 test seqences.

```
Starting the simulations from example 3 to 32
You can run vsim
Reading pref.tcl
Simulation for example 3 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 4 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 5 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 6 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 7 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 8 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 9 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 10 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 11 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 12 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 13 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 14 terminated successfully.
You can run vsim
Reading pref.tcl
```

```
Simulation for example 15 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 16 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 17 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 18 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 19 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 20 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 21 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 22 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 23 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 24 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 25 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 26 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 27 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 28 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 29 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 30 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 31 terminated successfully.
You can run vsim
Reading pref.tcl
Simulation for example 32 terminated successfully.
All the simulations completed successfully.
```

Report D.3. The `synopsys_commands_custom.tcl` Tcl script for the synthesis automation.

```
#*******Reading VHDL source files***************
#-------ANALYSIS PHASE
#compile all files provided by the script "compilation_list_gen.sh"
```

```
4  uplevel #0 source
   ↪  /home/thesis/costantino.taranto/git/AME_Architecture/scripts/syn_scripts/syn_toCompile.txt
   ↪  > ./synopsys_results/analyze_out.log
5  #Set one parameter to preserve rtl names in the netlist to ease the procedure
   ↪  for power consumption estimation.
6  set power_preserve_rtl_hier_names true
7
8  #-------ELABORATION PHASE
9  #Launch elaborate command to load the components
10 #elaborate <top entity name> -arch <architecture name> -lib WORK > log_fileName
11 elaborate AME_Architecture_expanded -arch structural -lib WORK >
   ↪  ./synopsys_results/elaborate_out.log
12 #uniquify #optional command to addres to only 1 specific architecture
13 link
14
15 #*******  Applying constraints   ***************
16 #create clock (period in ns)
17 create_clock -name MY_CLK -period 2.93 clk
18 #since the clock is a "special" signal in the design, we set the "don't touch"
   ↪  property
19 set_dont_touch_network MY_CLK
20
21 #jitter simulation
22 set_clock_uncertainty 0.07 [get_clocks MY_CLK]
23
24 #input/output delay
25 set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all_inputs]
   ↪  CLK]
26 set_output_delay 0.5 -max -clock MY_CLK [all_outputs]
27
28 #set output load (buffer x4 used)
29 set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
30 set_load $OLOAD [all_outputs]
31
32 #*********    Start the syntesis    *************
33 compile >  ./synopsys_results/compilation_results.txt
34
35 #*********    Save the results      *************
36 report_timing  >  ./synopsys_results/timing_results.txt
37 report_area    >  ./synopsys_results/area_results.txt
38
39 #Finally, we can save the data required to complete the design and to perform
   ↪  switchingactivity-based power estimation.
40 #First, we ungroup the cells to flatten the hierarchy as follows:
41 ungroup -all -flatten
42 #Then, we have to export the netlist in verilog. So that we impose verilog
   ↪  rules for the names of the internal signals. This is obtained with
43 change_names -hierarchy -rules verilog
44 #We also save a file describing the delay of the netlist:
```

```
45 write_sdf ../netlist/AME_Architecture_expanded.sdf
46 #We can now save the netlist in verilog:
47 write -f verilog -hierarchy -output ../netlist/AME_Architecture_expanded.v
48 #and the constraints to the input and output ports in a standard format:
49 write_sdc ../netlist/AME_Architecture_expanded.sdc
50
51 #******* close Design compiler    ****************
52 quit
```

Bibliography

[1] Mathias Wien and Benjamin Bross. Versatile video coding - algorithms and specification. ieee international conference on multimedia and expo (icme), 2020.

[2] Maurizio Masera. Transform algorithms and architectures for video coding. *Doctoral Program in Electrical, Electronic and Telecommunication Engineering (30th cycle)*, 2018.

[3] ViCueSoft. Vq analyzer - video analyzer and quality tuning. URL `https://vicuesoft.com/vq-analyzer/`.

[4] Young-Ju Choi, Dong-San Jun, Won-Sik Cheong, and Byung-Gyu Kim. Design of efficient perspective affine motion estimation/compensation for versatile video coding (vvc) standard. *Electronics*, 8(9), 2019. ISSN 2079-9292. doi: 10.3390/electronics8090993. URL `https://www.mdpi.com/2079-9292/8/9/993`.

[5] Khalid Sayood. *Introduction to Data Compression (Fifth Edition)*. Morgan Kaufmann, 2018. URL `https://www.sciencedirect.com/science/article/pii/B978012809474700001X`.

[6] Sankar Shanmuganathan and Nagarajan Ls. Zzrd and zzsw: Novel hybrid scanning paths for squared blocks. *International Journal of Applied Engineering Research*, 9: 10567–10583, 10 2014.

[7] Intel. Intel® quartus® prime lite edition design software. URL `https://www.intel.com/content/www/us/en/collections/products/fpga/software/downloads.html`. Version 21.1.

[8] Cisco Systems. Vni complete forecast highlights, 2018. `https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_2021_Forecast_Highlights.pdf`.

[9] Benjamin Bross, Ye-Kui Wang, Yan Ye, Shan Liu, Jianle Chen, Gary J. Sullivan, and Jens-Rainer Ohm. Overview of the versatile video coding (vvc) standard and its applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 31 (10):3736–3764, 2021. doi: 10.1109/TCSVT.2021.3101953.

[10] Frank Bossen, Karsten Sühring, Adam Wieckowski, and Shan Liu. Vvc complexity and software implementation analysis. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(10):3765–3778, 2021. doi: 10.1109/TCSVT.2021.3072204.

[11] Fraunhofer Heinrich-Hertz-Institut. Versatile video coding (vvc). URL `https://jvet.hhi.fraunhofer.de/`.

[12] Jill Boyce, Karsten Suehring, Xiang Li, and Vadim Seregin. Jvet-j1010: Jvet common test conditions and software reference configurations. 07 2018.

[13] Farhad Pakdaman, Mohammad Ali Adelimanesh, Moncef Gabbouj, and Mahmoud Reza Hashemi. Complexity analysis of next-generation vvc encoding and decoding. pages 3134–3138, 2020. doi: 10.1109/ICIP40778.2020.9190983.

[14] Wei-Jung Chien, Li Zhang, Martin Winken, Xiang Li, Ru-Ling Liao, Han Gao, Chih-Wei Hsu, Hongbin Liu, and Chun-Chi Chen. Motion vector coding and block merging in the versatile video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(10):3848–3861, 2021. doi: 10.1109/TCSVT.2021.3101212.

[15] Han Gao, Xu Chen, Semih Esenlik, Jianle Chen, and Eckehard Steinbach. Decoder-side motion vector refinement in vvc: Algorithm and hardware implementation considerations. *IEEE Transactions on Circuits and Systems for Video Technology*, 31 (8):3197–3211, 2021. doi: 10.1109/TCSVT.2020.3037024.

[16] Hasan Azgin, Ercan Kalali, and Ilker Hamzaoglu. An approximate versatile video coding fractional interpolation hardware. In *2020 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–4, 2020. doi: 10.1109/ICCE46568.2020.9042986.

[17] Matías J. Garrido, Fernando Pescador, M. Chavarrías, P. J. Lobo, and Cesar Sanz. A 2-d multiple transform processor for the versatile video coding standard. *IEEE Transactions on Consumer Electronics*, 65(3):274–283, 2019. doi: 10.1109/TCE.2019.2913327.

[18] Yibo Fan, Yixuan Zeng, Heming Sun, Jiro Katto, and Xiaoyang Zeng. A pipelined 2d transform architecture supporting mixed block sizes for the vvc standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(9):3289–3295, 2020. doi: 10.1109/TCSVT.2019.2934752.

[19] I. Farhat, W. Hamidouche, A. Grill, D. Menard, and O. Déforges. Lightweight hardware implementation of vvc transform block for asic decoder. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1663–1667, 2020. doi: 10.1109/ICASSP40776.2020.9054281.

[20] Werda Imen, Belghith Fatma, Maraoui Amna, and Nouri Masmoudi. Dct-ii transform hardware-based acceleration for vvc standard. In *2021 IEEE International Conference on Design and Test of Integrated Micro and Nano-Systems (DTS)*, pages 1–5, 2021. doi: 10.1109/DTS52014.2021.9498196.

[21] Jurgen Kello, Massimo Ruo Roch, Guido Masera, and Maurizio Martina. Low-complexity reconfigurable dct-v architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(12):3417–3421, 2020. doi: 10.1109/TCSII.2020.2998604.

[22] Matías J. Garrido, Fernando Pescador, Miguel Chavarrías, Pedro J. Lobo, César Sanz, and Pedro Paz. An fpga-based architecture for the versatile video coding multiple transform selection core. *IEEE Access*, 8:81887–81903, 2020. doi: 10.1109/ACCESS.2020.2991299.

[23] Bouthaina Abdallah, Fatma Belghith, and Nouri Masmoud. Low-complexity transform algorithm for versatile video coding. In *2019 IEEE International Conference on Design and Test of Integrated Micro and Nano-Systems (DTS)*, pages 1–3, 2019. doi: 10.1109/DTSS.2019.8915188.

[24] Heiko Schwarz, Tung Nguyen, Detlev Marpe, Thomas Wiegand, Marta Karczewicz, Muhammed Coban, and Jie Dong. Improved quantization and transform coefficient coding for the emerging versatile video coding (vvc) standard. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 1183–1187, 2019. doi: 10.1109/ICIP.2019.8803768.

[25] Zhaobin Zhang, Xin Zhao, Xiang Li, Li Li, Yi Luo, Shan Liu, and Zhu Li. Fast dst-vii/dct-viii with dual implementation support for versatile video coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(1):355–371, 2021. doi: 10.1109/TCSVT.2020.2977118.

[26] Keke Ding, Dong Jiang, Feiyang Zeng, Jucai Lin, and Jun Yin. A fast transform algorithm based on vvc. In *2020 4th International Conference on Computer Science and Artificial Intelligence*, CSAI 2020, page 80–85, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388436. doi: 10.1145/3445815.3445829. URL `https://doi.org/10.1145/3445815.3445829`.

[27] Marta Karczewicz, Nan Hu, Jonathan Taquet, Ching-Yeh Chen, Kiran Misra, Kenneth Andersson, Peng Yin, Taoran Lu, Edouard François, and Jie Chen. Vvc in-loop filters. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(10):3907–3925, 2021. doi: 10.1109/TCSVT.2021.3072297.

[28] Sijia Chen, Zhenzhong Chen, Yingbin Wang, and Shan Liu. In-loop filter with dense residual convolutional neural network for vvc. In *2020 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, pages 149–152, 2020. doi: 10.1109/MIPR49039.2020.00038.

[29] Zhijie Huang, Jun Sun, Xiaopeng Guo, and Mingyu Shang. One-for-all: An efficient variable convolution neural network for in-loop filter of vvc. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(4):2342–2355, 2022. doi: 10.1109/TCSVT.2021.3089498.

[30] Xuewei Meng, Jiaqi Zhang, Chuanmin Jia, Zhang Xinfeng, Wang Shanshe, and Ma Siwei. Optimized adaptive loop filter in versatile video coding. In *2021 Data Compression Conference (DCC)*, pages 359–359, 2021. doi: 10.1109/DCC50243.2021.00082.

[31] Heiko Schwarz, Muhammed Coban, Marta Karczewicz, Tzu-Der Chuang, Frank Bossen, Alexander Alshin, Jani Lainema, Christian R. Helmrich, and Thomas Wiegand. Quantization and entropy coding in the versatile video coding (vvc) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(10):3891–3906, 2021. doi: 10.1109/TCSVT.2021.3072202.

[32] Heiko Schwarz, Tung Nguyen, Detlev Marpe, Thomas Wiegand, Marta Karczewicz, Muhammed Coban, and Jie Dong. Improved quantization and transform coefficient coding for the emerging versatile video coding (vvc) standard. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 1183–1187, 2019. doi: 10.1109/ICIP.2019.8803768.

[33] Meng Wang, Shiqi Wang, Junru Li, Li Zhang, Yue Wang, Siwei Ma, and Sam Kwong. Low complexity trellis-coded quantization in versatile video coding. *IEEE Transactions on Image Processing*, 30:2378–2393, 2021. doi: 10.1109/TIP.2021.3051460.

[34] Kai Zhang, Yi-Wen Chen, Li Zhang, Wei-Jung Chien, and Marta Karczewicz. An improved framework of affine motion compensation in video coding. *IEEE Transactions on Image Processing*, 28(3):1456–1469, 2019. doi: 10.1109/TIP.2018.2877355.

[35] Joint Video Experts Team (JVET). Test model 8 for versatile video coding (vtm 8), 2020-01-17. URL `https://mpeg.chiariglione.org/standards/mpeg-i/versatile-video-coding/test-model-8-versatile-video-coding-vtm-8`.

[36] Matheus F. Stigger, Victor H. S. Lima, Leonardo B. Soares, Claudio M. Diniz, and Sergio Bampi. Approximate satd hardware accelerator using the 8×8 hadamard transform. In *2020 IEEE 11th Latin American Symposium on Circuits and Systems (LASCAS)*, pages 1–4, 2020. doi: 10.1109/LASCAS45839.2020.9068987.

[37] Sang-Hyo Park and Je-Won Kang. Fast affine motion estimation for versatile video coding (vvc) encoding. *IEEE Access*, 7:158075–158084, 2019. doi: 10.1109/ACCESS.2019.2950388.

[38] Weizheng Ren, Wei He, and Yansong Cui. An improved fast affine motion estimation based on edge detection algorithm for vvc. *Symmetry*, 12(7), 2020. ISSN 2073-8994. doi: 10.3390/sym12071143. URL `https://www.mdpi.com/2073-8994/12/7/1143`.

[39] G. Bjontegaard. Calculation of average psnr differences between rd-curves. *ITU-T VCEG-M33*, 2001. URL `https://cir.nii.ac.jp/crid/1572543025125831168`.

[40] Ahmet CanMert, Ercan Kalali, and Ilker Hamzaoglu. A low power versatile video coding (vvc) fractional interpolation hardware. In *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 43–47, 2018. doi: 10.1109/DASIP.2018.8597040.

[41] Xin Wang, Heming Sun, Jiro Katto, and Yibo Fan. A hardware architecture for adaptive loop filter in vvc decoder. In *2021 IEEE 14th International Conference on ASIC (ASICON)*, pages 1–4, 2021. doi: 10.1109/ASICON52560.2021.9620332.

[42] Television History The First 75 Years. America's first electronic television set. URL `http://www.tvhistory.tv/1939%20Du%20Mont%20Brochure.htm`.

[43] Mary Bellis. The history of color television, 2019. URL `https://www.thoughtco.com/color-television-history-4070934`.

[44] Public Subsidiary of Hubbard Broadcasting Inc. History of u.s. satellite broadcasting company inc., 2018. URL `http://www.fundinguniverse.com/company-histories/u-s-satellite-broadcasting-company-inc-history/`.

[45] Margaret Livingstone. The first stages of processing color and luminance: Where and what, 2002.

[46] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, 2012. doi: 10.1109/TCSVT.2012.2221191.

Acknowledgements

This thesis work represents the conclusion of a study cycle that started a long time ago. Along this path, I have met wonderful people for which I am grateful, but first of all, I want to show my love and appreciation to some people who were always there, from the beginning: my parents. Without them, I would never be able to live the life I'm living and get there to produce these beautiful results. Thank you from the deepest part of my heart.

Another important person which I owe this work to and I really want to thank is my dear supervisor, professor Maurizio Martina. During these months I discovered him as a strongly enthusiast, sympathetic and encouraging professor, and a joyful and pleasant person. Thank you for the interest you showed in my work and the support you gave me through all this time.

During this last year in Politecnico, I did never stay in a classroom, since I had already attended all my courses. My house has been the VLSI Lab, which I want to thank for hosting me.
I am glad I had the chance to meet in person a teacher who changed my life during the last academic year: Prof. Mariagrazia Graziano. I am never tired to tell her how much I'm grateful for her teachings, and I am doing so here too.
Thank you also to the great teachers and beautiful persons I have met during this course of study at Politecnico: Prof. Maurizio Zamboni, Prof. Gianluca Piccinini, Prof. Guido Masera, Prof. Massimo Ruo Roch, and all the others who have increased my love and enthusiasm for this marvelous and curious subject that is Electronic Engineering.

I want to show my love and gratitude to my fantastic colleagues and friends which whom I collaborated during these years. Fore example, the *Operazione San Silvestro*'s improbable group: Martino D'Alessandro, Francesco Trinca e Pasquale Santoro. Thank you for all the fun and the time spent together. Not to forget my friends Gianluca Goti, Matilde Cerbai, and Laura Chisciotti, I'm glad we had the luck to meet in person and to stay close during the hard times. Thank you for your beautiful company and support.

A huge thank you to all my friends from Calabria, whom I will not list here since there are too many names. You are my second family and I would not be here without your company and emotional support. Thank you for all the laughter, the fun, and the precious moments we lived together.

To my family: my loved brother, my cousins, my uncles and aunts, and my two grandmas. I am grateful for your love throughout all these years. You were always there since I moved my first steps into this world, and I hope you will always be by my side.