POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

RTL Implementation of the Adaptive Loop Filter for VVC encoder

Tesi di Laurea Magistrale in Telecommunication Engineering - Ingegneria delle Telecomunicazioni

Author: **Thomas Degand**

Student ID: 219156
Advisor: Prof. Paolo Bestagini
Academic Year: 2023-24

Abstract

This thesis explores the design and RTL implementation of the Adaptive Loop Filter (ALF) within the Versatile Video Coding (VVC) standard, developed by the Joint Video Experts Team and Fraunhofer HHI. The ALF is crucial for reducing visual artifacts in VVC-encoded video streams. The research was conducted during an internship at Allegro DVT in Grenoble, France.

The State of the Art section offers a comprehensive analysis of the ALF's technical specifications as outlined in the VVC standard. The filter employs a diamond-shaped pattern applied to 7x7 pixel blocks for luminance. It computes new pixel values based on neighboring pixel luminance, using coefficients influenced by parameters such as class index, filter set, and transposition index. This filtering process is optimized through gradient-based calculations on 4x4 pixel blocks, enabling the encoder to select the most suitable filter for each block to minimize distortion while balancing computational demands.

The thesis details the translation from the ALF specification and the software encoder into RTL implementation, including technical optimization choices. It covers fundamental RTL design concepts such as flip-flop processes, combinatorial logic, registers, pipelining, valid/ready interfaces, FIFO, FSM, and RAM, which are crucial for developing a thorough understanding of the mechanisms involved.

The implementation section describes the ALF's decomposition and block-by-block realization of its functions, emphasizing technical decisions and optimizations to meet design requirements and reduce synthesis area. A validation and testing method is also outlined.

In conclusion, the thesis confirms the successful RTL design of the ALF, highlights the personal and professional benefits gained from the internship, and reflects on the subject and context of the project.

Keywords: ALF, RTL, Implementation, VVC, Filter, Video, Diamond shape, ASIC

Abstract in lingua italiana

Questa tesi esplora la progettazione e l'implementazione RTL del Adaptive Loop Filter (ALF) all'interno dello standard Versatile Video Coding (VVC), sviluppato dal Joint Video Experts Team e dal Fraunhofer HHI. L'ALF è cruciale per ridurre gli artefatti visivi nei flussi video codificati in VVC. La ricerca è stata condotta durante uno stage presso Allegro DVT a Grenoble, in Francia.

La sezione sullo Stato dell'Arte offre un'analisi approfondita delle specifiche tecniche dell'ALF come descritto nello standard VVC. Il filtro utilizza un modello a forma di diamante applicato a blocchi di 7x7 pixel per la luminanza. Calcola nuovi valori di pixel in base alla luminanza dei pixel vicini, utilizzando coefficienti influenzati da parametri come l'indice di classe, il set di filtri e l'indice di trasposizione. Questo processo di filtraggio è ottimizzato tramite calcoli basati su gradienti su blocchi di 4x4 pixel, consentendo all'encoder di selezionare il filtro più adatto per ciascun blocco al fine di ridurre al minimo la distorsione, bilanciando al contempo le richieste computazionali.

La tesi descrive la traduzione dalla specifica dell'ALF e dell'encoder software all'implementazione RTL, comprese le scelte tecniche di ottimizzazione. Vengono trattati i concetti fondamentali della progettazione RTL come i processi flip-flop, la logica combinatoria, i registri, il pipelining, le interfacce valid/ready, FIFO, FSM e RAM, che sono cruciali per sviluppare una comprensione approfondita dei meccanismi coinvolti.

La sezione sull'implementazione descrive la decomposizione dell'ALF e la realizzazione blocco per blocco delle sue funzioni, sottolineando le decisioni tecniche e le ottimizzazioni per soddisfare i requisiti di progettazione e ridurre l'area di sintesi. Viene inoltre delineato un metodo di validazione e test.

In conclusione, la tesi conferma il successo della progettazione RTL dell'ALF, evidenzia i benefici personali e professionali acquisiti durante lo stage e riflette sul soggetto e il contesto del progetto.

Parole chiave: ALF, RTL, Implementazione, VVC, Filtro, Video, Quadri, ASIC

Contents

Foreword

The thesis focuses on the development of an Adaptive Loop Filter (ALF) for a state-of-the-art video encoder based on the Versatile Video Coding (VVC) standard. The goal of the ALF is to enhance the quality of compressed video by refining the image data through adaptive filtering techniques, which are critical for achieving high visual fidelity and efficient compression.

The ALF operates on Coding Units (CUs) of size 4×4 px within the video frame, employing a diamond-shaped filtering and cost calculation processes to optimize the encoding. The project encompasses several key stages:

1. **Design and Realization of the ALF Blocks**: This includes the development of blocks for classification of CUs, and pixel filtering.
2. **Cost Calculation**: The ALF performs cost estimation based on the distortion between filtered and original pixel values. This involves calculating the distortion for individual CUs and evaluating multiple filtering options to select the optimal filter set for groups of CUs.
3. **Integration and Validation**: The various blocks are integrated into a comprehensive ALF system developped in VHDL. And a detailed testbench is created to validate the functionality of the ALF. This includes using traces from a software encoder to compare results and ensure accuracy.
4. **Synthesis and Optimization**: The ALF design is synthesized to determine its physical footprint and performance characteristics on an ASIC. This phase includes evaluating different synthesis results to address critical path delays and optimize the design for efficient hardware implementation.

1 | Introduction

1.1. Company Presentation

1.1.1. The Company Allegro DVT

Created in 2003, Allegro DVT is a rapidly growing international company whose expertise is globally recognized in the field of digital video in the semiconductor market (RTL design of video encoders and decoders). Its R&D services are located in Montbonnot-Saint-Martin and Belfast. Its teams in France are composed of around forty engineers, all passionate about video processing. They also have commercial presence in the United States, China, and Finland. In total, four main teams make up the company: Compliant Stream, Software, Hardware, and Marketing.

In 2022, Allegro DVT acquired the company Labwise, originating from Finland, which offers large-scale testing services for digital television devices to TV broadcasters, network operators, and manufacturers.

In 2024, the company also integrated its former direct competitor in Compliance Stream, ViCueSoft, based in Armenia and Cyprus. It is a software technology company specializing in video quality analysis, transcoding, and machine learning. It should be noted that their software played a significant role in the realization of the internship.

Allegro DVT's video encoders and decoders are present in car cameras, video game streaming services, surveillance cameras, etc. Leaders in microelectronics, the automotive industry, and video surveillance are among its clients. The company is attractive in emerging markets such as AI, autonomous vehicles, robotics, and healthcare.

1.1.2. Products Sold

The main products sold by Allegro DVT are their video encoders and decoders in the form of RTL designs for ASIC, meaning a set of deliverables in VHDL and SystemVerilog that form a circuit synthesizable into a chip. Clients are responsible for the

chip manufacturing process themselves. The advantage of these designs is that they are customizable and adaptable for clients. An RTL design, due to its targeted action, is a much faster system compared to software encoders like reference ones or their variations, such as those found in ffmpeg.

They also provide an online tracking and debugging system through a ticket platform.

A separate team, originally the company's first, sells compliant stream tests, a means of testing the correct decoding of video streams. These tests are designed to account for all edge cases that could potentially cause problems for a decoder.

1.1.3. Overall Team Structure

Allegro DVT's goal is to expand into all video markets. To achieve this, the company requires a sales team, which is the Marketing team, along with research and development teams.

The first R&D team is the compliance stream team. They focus on building stress video streams, meaning files that push decoders to their limits. To achieve this, they need to have an in-depth understanding of all the specifications. During tenders, they even have the opportunity to propose their own specifications as a new official standard for a codec.

The second team is the Software team. They develop various encoder and decoder models, which are then used to assist the Hardware team in designing the RTL systems. Even though the product sold is a design, the Software team is also responsible for creating programs that will be introduced into the chips to manage, via a microcontroller, certain actions too complex to be developed in hardware. For clients, these programs are easier to use, and they can be updated without needing to reproduce new chips. Another team has emerged: the Video Algorithm team, which conducts research on new ways to implement AI at the software level, and later at the hardware level.

The final team is the Hardware team. Composed of about ten R&D engineers, it handles the RTL design of ASICs in SystemVerilog and VHDL for the video encoders and decoders sold by the company. It can be divided into two sub-teams: the design creation team, which develops the overall architecture for video processing as well as register and memory access (this is the team where the ALF internship was offered), and the delivery team, which assembles all hardware and software elements and generates them according to the client's requirements and constraints.

1.2. Internship Objectives

1.2.1. Introduction of the VVC Codec

Online video platforms have been booming for quite some time, as evidenced by the rise of YouTube, Twitch, and even Dailymotion. They were then followed by emerging streaming platforms like Netflix, Prime Video, and Disney+. More recently, short portrait-format videos have proliferated across all platforms, especially on social networks. And this is only part of the current video market. This market presents huge technological challenges. How can we store and transmit more content without exceeding hardware or economic limits? We must constantly push boundaries, look further, and see better. In fact, the word "video" comes from Latin, meaning "I see."

A video codec serves to see, to make video tangible. It is no longer raw; it is compressed and refined. The most well-known video codec is undoubtedly H.264, which is widely used in .mp4 files. This codec, also known as AVC, was announced in 2003 and is currently the most used worldwide. However, improvements didn't stop there. H.265, named HEVC, was announced in 2013 as its official successor. In 2020, H.266, known as VVC or Versatile Video Coding, was introduced as the latest replacement. VVC introduces a whole set of new features. It aims to reduce the video stream size by an average of 50

As a result, Allegro DVT began exploring this development by creating its own version of the software encoder and decoder and developing compliance tests.

1.2.2. Creation of an Internship on the Adaptive Loop Filter

In this context, the company found it interesting to offer an internship on one of the new features: the Adaptive Loop Filter (ALF). It is a visual enhancement filter integrated into video encoders. It belongs to a category of filters called In-Loop Filters, alongside previous filters such as Deblocking and Sample Adaptive Offset (SAO), whose use will be described later.

As mentioned earlier, this filter has already been specified and developed internally by Allegro in a software version. The internship consists of designing the RTL (Register Transfer Level) version of this filter. It was decided that this filter would be developed in VHDL and could be integrated into the hardware encoders previously created by the company. In reality, this filter is a parameterizable filter that combines several filters. The parameter choices justify the creation of a six-month internship. Therefore, the internship

will focus on the creation of the ALF in the VVC video encoder, not the decoder. This version implements more hardware realization methods, with parameter determination techniques. There is a complement to this filter called CCALF, but it will not be studied.

1.2.3. Documentation and Tools Provided

The provided documents include the VVC specification and the C++ source code of the software encoder developed by Allegro DVT.

The tools are installed on a Debian laptop. Notably, they include ViCueSoft's VQAnalyser and VQBench software for VVC video stream analysis, Synopsys' VCS software for simulating and analyzing signals of the implemented RTL components, Synopsys' Spyglass software, and an ASIC synthesis tool for verifying hardware design errors and ensuring compliance with various constraints, such as evaluating the potential chip area and adhering to clock frequency requirements.

1.2.4. Basic Operation of a Video Encoder

The primary goal of a video encoder is to generate a data stream, known as a bitstream, from source images. This bitstream allows a decoder to reconstruct the video as faithfully as possible to the original source. In this case, perfect quality during reconstruction is defined as the source itself. The difference between the reconstruction and the source is measured and referred to as distortion.

To generate this bitstream, each image in the video undergoes spatial and temporal transformations. A preliminary color conversion step can be added if necessary, defining pixel group colors within the YUV range. A pixel's color is separated into a luminance component (Y) and blue and red chrominance components (U and V), enabling better compression. Since the human eye is more sensitive to luminance variations than color variations, it is possible to reduce the amount of information for blue and red chrominance. This process reduces the data to be processed without significantly affecting perceived quality.

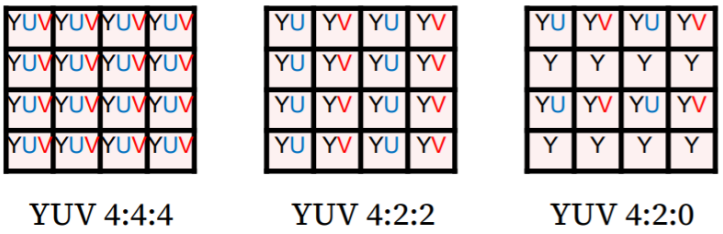

Figure 1.1: Distribution of the Y luminance and U and V color components in a 4x4px square.

The spatial and temporal transformation is then quantified, significantly reducing the size of the data by generating a series of much smaller values than those required to transmit a raw image. These resulting data are called residuals.

The residuals are then combined with header information to create an uncoded bitstream. This stream undergoes additional encoding aimed at further reducing its size by applying entropy approximation techniques. The encoded bitstream is then ready to be stored or transmitted to communicate the video to the decoder.

To ensure the transmission of optimal headers necessary for image reconstruction, the encoder, like the decoder, incorporates a residual reconstruction component. This section includes inverse quantization, inverse transformation, and filters. These filters, known as In-Loop Filters, aim to improve the quality of the reconstructed image by bringing it as close as possible to the source image.

In the case of the encoder, the primary objective of this step is to test different filtering options to minimize distortion and achieve the highest possible quality close to the original source. However, it is often necessary to strike a balance between minimal distortion and the cost of entropy encoding for bitstream production parameters. This trade-off is typically controlled by a variable, often denoted as Qp (for Quality Parameter), which serves as a quality factor.

In an encoder, the reconstruction part is referred to as "looped" because the encoder tests several filters with varying parameters to correct the defects induced by the residual creation method. The encoder generally works on reduced image sections, organized in squares of variable sizes called Coding Units (CU) for HEVC and VVC codecs. This process is often dictated by technical constraints related to available RAM and real-time processing capability. This division into blocks, known as Coding Tree Unit (CTU), can lead to undesirable effects such as distortion and shattering.

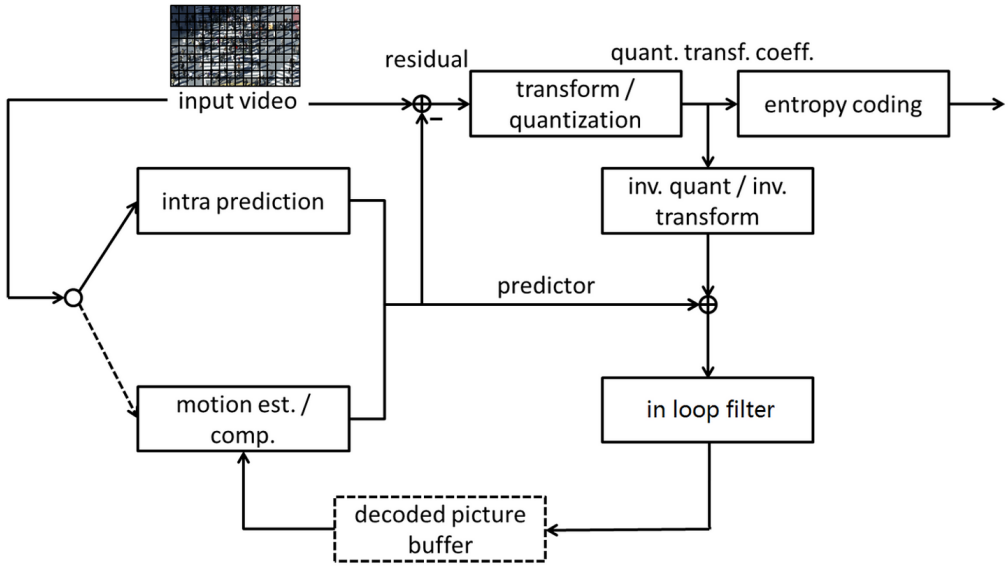

Figure 1.2: Block Diagram of a VVC Encoder.

1.2.5. Advantages of In-Loop Filters

To address these imperfections, the encoder employs specific filters, including Deblocking, Sample Adaptive Offset (SAO), and Adaptive Loop Filtering (ALF).

Figure 1.3: Apparent defects in a reconstructed image then filtered by different In-Loop Filters.

Deblocking is considered essential in AVC, HEVC, and VVC standards, while SAO and ALF are optional filters that can be applied or not. When analyzing decoded images using the VQAnalyser software, it is evident that while adding a filter can reduce imperfections, it can also introduce new ones. This partly explains the introduction of the ALF in VVC. ALF compensates for the shortcomings of the two previous filters. In other words, they work together harmoniously.

To measure the distortion reduction gains for a fixed Quality Parameter (Qp), the Peak-Signal-to-Noise-Ratio (PSNR) of a reconstructed image can be calculated using the formula below. The PSNR is geometrically proportional to the distortion.

$$PSNR = 10 \log_{10} \left(\frac{d^2}{EQM} \right) \tag{1.1}$$

Used Filters	**Deblocking**	**Deblocking + SAO**	**Deblocking + ALF**	**Deblocking + BOTH**
Relative PSNR Gain	—	-3.329%	-2.244%	-4.315%

Table 1.1: Table representing the average distortion reduction (PSNR) at fixed Qp for Allegro DVT's software encoder.

Thus, we can observe the average distortion reduction obtained by adding SAO compared to the distortion with only Deblocking, and the additional reduction after incorporating ALF. This measurement was carried out on about ten videos encoded by Allegro DVT's software encoder. To achieve these results, it is necessary to set up different configuration files for the encoders, list the videos to be measured from a company server, and run the measurements using the VQBench tool. It is important to note that even a small reduction in distortion at a fixed Qp represents a significant technical advancement in video encoding.

1.2.6. ALF Specifications

The ALF is thus an important element of the new VVC codec. Its use is justified, but it must be properly designed and ready for implementation. A specification document has been established to guide its development.

The ALF filter will only process the Y luminance component of the videos. It must be capable of filtering at a rate of 2 pixels per clock cycle. Additionally, the clock frequency must exceed 1 GHz, which should be feasible assuming its synthesis on a 7nm precision chip. The videos to be filtered should support resolutions up to 4K, in either portrait or landscape format, or a square size of 4096 x 4096 pixels.

2 | State of Art

2.1. ALF in VVC Specification

2.1.1. Filtering Guidelines

The filter is described in the specification as a diamond-shaped filter. For a luminance filter, it has a size of 7x7 pixels. It involves evaluating a new luminance value for a central pixel based on its neighboring pixels. The values of its 24 neighbors are added to the central pixel's value, each multiplied by 12 coefficients arranged symmetrically.

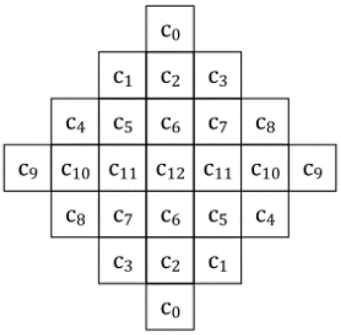

Figure 2.1: Arrangement of pixels used for a 7x7 diamond filtering.

The coefficients and their arrangement are determined by 3 parameters: a class index, a filter set index, and a transposition index. A table maps these indices to coefficients from among 64 precomputed coefficient sets by the team responsible for the specification.

Figure 2.2: Coefficient determination tables.

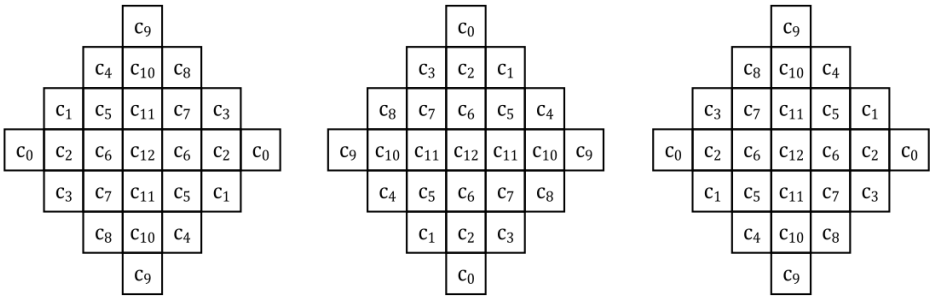

Figure 2.3: Arrangement of coefficients for transposition indices 1, 2, and 3.

Once the 12 coefficients and the 25 associated pixels are known, three calculation steps are applied. The first step is performed for each of the 24 neighboring pixels. It involves calculating the difference between the luminance value of the neighboring pixel and that of the central pixel (luminance is denoted as R). The second step, known as "clipping" (variables b_i), constrains the result of the difference between two bounds, negative and positive, of equal absolute value. This value typically corresponds to 2 raised to the number of bits used to define the luminance (8, 10, or 12 bits).

$$\begin{aligned} f_i = &\min(b_i, \max(-b_i, R(x + x_i, y + y_i) - R(x, y))) \\ &+ \min(b_i, \max(-b_i, R(x - x_i, y - y_i) - R(x, y))) \end{aligned} \tag{2.1}$$

These 24 values (variables f_i) are then multiplied by their associated coefficient. The results are summed, and, due to an approximation method involving the central pixel

in the differences, $64 = 2^7 - 1$ is added, then the sum is divided by $128 = 2^7$. This final result, whether negative or positive, is added to the original luminance value of the central pixel to obtain its filtered value.

$$\tilde{R}(x,y) = R(x,y) + \left[\left(\sum_{i=0}^{N-2} c_i f_i + 64 \right) \gg 7 \right] \tag{2.2}$$

It will be noted later that the introduction of image borders and edges will impose a slight variation in the calculation method.

2.1.2. Coefficient Determination Guidelines

The filtering process relies on 3 parameters applied at 2 levels of image partitioning.

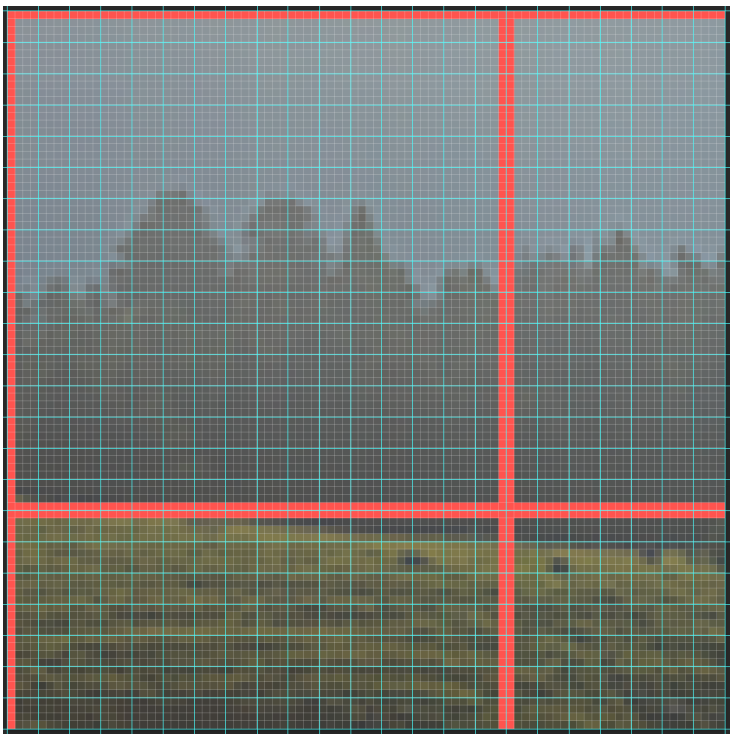

Figure 2.4: Partitioning of an image into 64x64 LCU and 4x4 pixel squares.

Each image is partitioned, as previously mentioned, into Coding Tree Units (CTUs). The choice of filter set or the absence of a filter is made at the largest possible section level, called the Largest Coding Unit (LCU). The image is therefore divided into large squares on which a choice of index set is applied. This partitioning concept is the only information transcribed in the specification; the method of choice is not presented, nor is the size of the LCU.

Another partitioning method divides the image into 4x4 pixel squares. This choice is due to the YUV color encoding, such as 4:2:0 and others, which make these pixels interdependent. At this level, the class index and the transposition index are determined. The specification precisely describes how to determine the indices for a given 4x4 pixel square. A 10x10 pixel square around the 4x4 pixel square to be filtered is required, as shown in the image below.

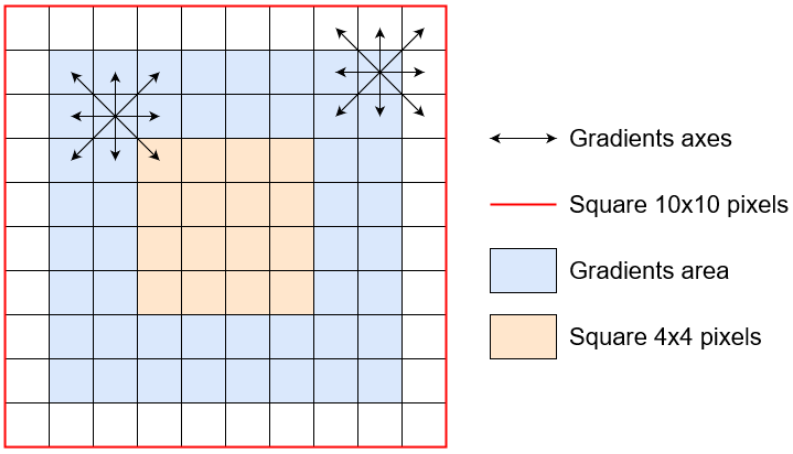

Figure 2.5: Representation of local gradients.

The specification describes an initial step called local gradient calculation. This step involves determining an activity orientation within 4x4 pixel blocks by considering their 8x8 pixel surroundings. To achieve this, gradients in 4 different directions are calculated, using a one-term Laplacian method.

$$\begin{aligned} H_{k,l} &= |2R(k,l) - R(k-1,l) - R(k+1,l)|, \\ V_{k,l} &= |2R(k,l) - R(k,l-1) - R(k,l+1)|, \\ D0_{k,l} &= |2R(k,l) - R(k-1,l-1) - R(k+1,l+1)|, \\ D1_{k,l} &= |2R(k,l) - R(k-1,l+1) - R(k+1,l-1)|. \end{aligned} \tag{2.3}$$

Once these gradients are obtained, they are summed to obtain 4 total gradients in the 4 directions: horizontal, vertical, right diagonal, and left diagonal.

$$
\begin{aligned}
g_h &= \sum_{k=i-2}^{i+5} \sum_{l=j-2}^{j+5} H_{k,l}, \\
g_v &= \sum_{k=i-2}^{i+5} \sum_{l=j-2}^{j+5} V_{k,l}, \\
g_d0 &= \sum_{k=i-2}^{i+5} \sum_{l=j-2}^{j+5} D0_{k,l}, \\
g_d1 &= \sum_{k=i-2}^{i+5} \sum_{l=j-2}^{j+5} D1_{k,l}.
\end{aligned}
\tag{2.4}
$$

To facilitate calculations during RTL design, the specification stipulates that the sum of gradients should only be computed every other time.

Initially, these 4 gradients will allow, through comparison, to determine the transposition index. The equivalence in coefficient arrangement is described in the previous section on filtering, 2.3.

Gradient Comparison	**Transformation**	**Transposition Index**
$g_{d1} < g_{d0}$ and $g_h < g_v$	No transformation	0
$g_{d1} < g_{d0}$ and $g_v \leq g_h$	Diagonal mirror	1
$g_{d0} \leq g_{d1}$ and $g_h < g_v$	Vertical mirror	2
$g_{d0} \leq g_{d1}$ and $g_v \leq g_h$	90° rotation	3

Table 2.1: Table of transposition index associations.

The class index calculation step is done by calculating 2 variables: D for Direction and A for Activity, using an algorithm with two threshold parameters denoted t1 and t2, which are not specified. The class index can range from 0 to 24, with the variable D being integer and varying from 0 to 3, while the variable A is quantized and varies from 0 to 4. We will have

$$
C = 5D + \hat{A} \tag{2.5}
$$

$$
A = \sum_{k=i-2}^{i+5} \sum_{l=j-2}^{j+5} (V_{k,l} + H_{k,l}) \tag{2.6}
$$

Algorithm 2.1 Determination of directivity D

1: $g_{h,v}^{min} = \min(g_h, g_v)$, $g_{h,v}^{max} = \max(g_h, g_v)$
2: $g_{d0,d1}^{min} = \min(g_{d0}, g_{d1})$, $g_{d0,d1}^{max} = \max(g_{d0}, g_{d1})$
3: **if** $g_{h,v}^{max} \leq t_1 \cdot g_{h,v}^{min}$ and $g_{d0,d1}^{max} \leq t_1 \cdot g_{d0,d1}^{min}$ **then**
4: $D \Leftarrow 0$
5: **end if**
6: **if** $g_{h,v}^{max}/g_{h,v}^{min} > g_{d0,d1}^{max}/g_{d0,d1}^{min}$ **then**
7: **if** $g_{h,v}^{max} > t_2 \cdot g_{h,v}^{min}$ **then**
8: $D \Leftarrow 2$
9: **else**
10: $D \Leftarrow 1$
11: **end if**
12: **else**
13: **if** $g_{d0,d1}^{min} > t_2 \cdot g_{d0,d1}^{max}$ **then**
14: $D \Leftarrow 4$
15: **else**
16: $D \Leftarrow 3$
17: **end if**
18: **end if**

2.1.3. Virtual Boundaries and Padding

From the beginning, we have introduced two related concepts: that of boundaries, due to the splitting of images into squares called LCUs and the splitting into 4x4 pixel squares, and that of neighbors, which can be those around the filtered pixels or the neighbors around the 4x4 squares during gradient calculation. As we will see later, these elements are of paramount importance in the RTL design realization. The specification therefore provides means to reduce their material and visual impact. To begin with, consider an LCU. On its last line of 4x4 pixel squares, a so-called virtual boundary is added. We will no longer consider the boundaries of LCUs, but only these long horizontal virtual boundaries traversing the image and the image edges.

Figure 2.6: Cutting an image into LCUs in red with virtual boundaries in blue.

The consideration of boundaries is realized through the use of what is called padding. If there is a need to account for pixels located relatively on the other side of a boundary compared to a current 4x4 square, they are assigned a repeated value of the nearest pixels. The best way to understand this is through an example in the lower left corner of an image, with a height aligned with an LCU.

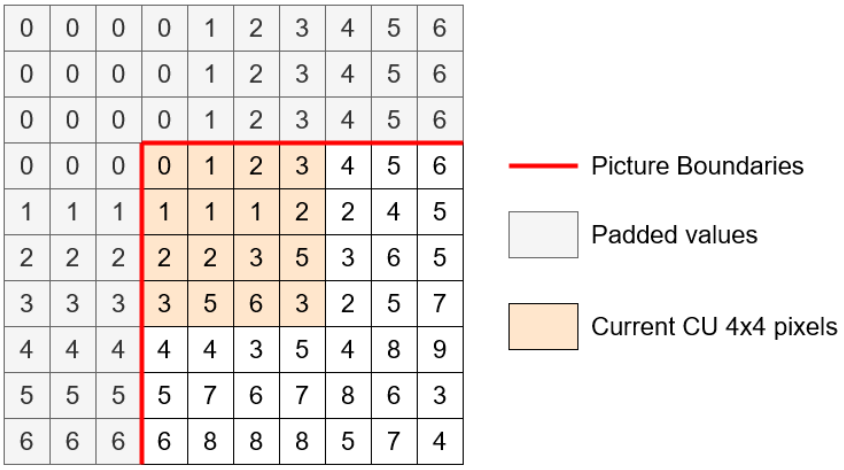

Figure 2.7: Representation of a 4x4 pixel square at the image edge with padding.

In the case of determining a class index or transposition index, gradients located outside of boundaries are simply not summed. However, the calculation of filtered pixels is more complex. If one is on the last or second-to-last line of an LCU split into 4x4 squares, it may happen that the filtering diamond exceeds a boundary. In this case, padding is performed as follows:

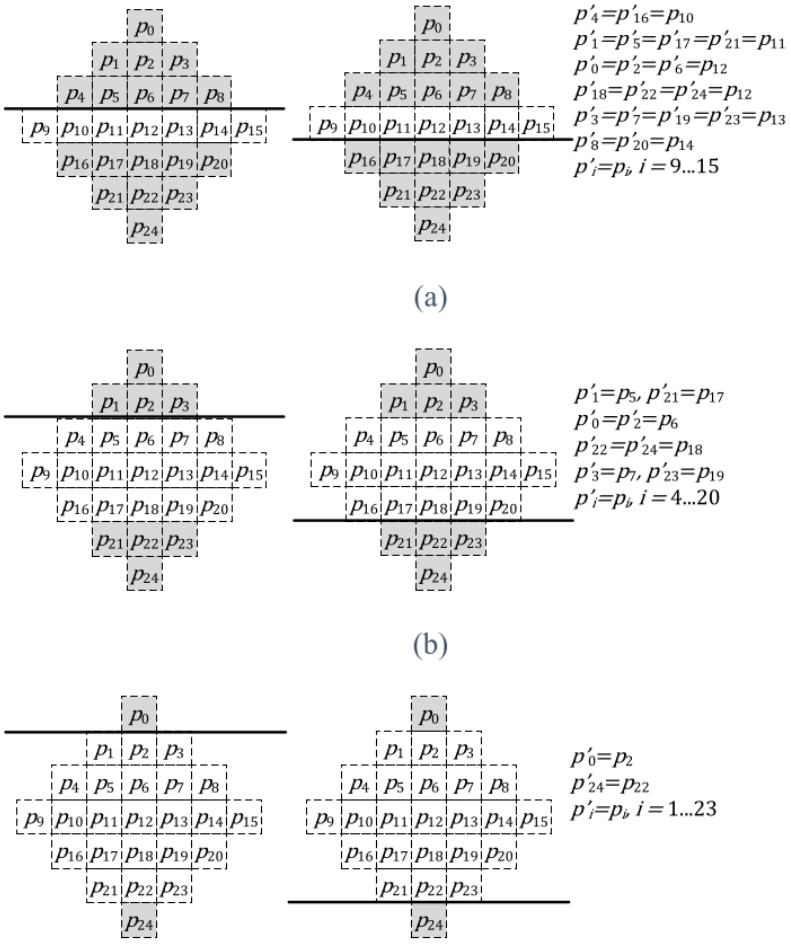

Figure 2.8: Padding for 7x7 diamond filtrations.

$$\tilde{R}(x,y) = R(x,y) + \left[\left(\sum_{i=0}^{N-2} c_i f_i + 512 \right) \gg 10 \right] \tag{2.7}$$

2.2. VVC Software Encoder by Allegro DVT

2.2.1. Specification Details

The Software team has made certain technical choices to guide the implementation of the ALF. The first choice is the use of buffers, which symbolize the inputs and outputs of the filters. A buffer consists of three 16-bit integer arrays, each corresponding to a line representing the three components of luminance Y, blue chrominance U, and red chrominance V, allowing for encoding images in 8, 10, and 12 bits. Additionally, to locate square images within these lines of values, values called Pitch have been assigned

to describe the image width. For example, for a 20x20 pixel image in YUV 4:4:4, we will have 3 arrays of 400 integers with associated Pitch values of 20.

Understanding how these buffers, written in C++, work is essential because, during testing periods, it will be necessary to simulate the ALF's input and output RAMs using a copy of these buffers in text file format, called traces.

The analysis of these buffers reveals that the ALF has been designed to operate by processing one LCU at a time, traversing the image from left to right and top to bottom. This reading method is called raster mode. The Software team has also chosen that the size of the LCUs used will be 64x64 pixels.

This method presents a problem concerning the right and bottom boundaries of the LCU being processed. As the neighboring values are still unknown, the decision has been made to apply padding. Another method will be described later.

The final choice of the team concerns the specification's vagueness regarding the use of thresholds in the class determination algorithm. These thresholds are set to t1 = 2 and t2 = 9.

2.2.2. Cost Calculation and Filter Selection

However, there is one important element not addressed in the specification: the method for choosing filter sets among the 16 available for an LCU. The team introduces 2 phases: a first search phase that calculates a cost for each LCU, and a second final phase that applies the best filter for the determined filter sets. The cost is defined by the distortion of the filtered image compared to the source image input into the encoder and the bit cost to encode the filtering indices. The cost of no filtering is defined as the distortion between the source image and the reconstructed image at the ALF input. This cost is not calculated over the entire LCU but over a section excluding the right and bottom boundaries of the LCUs. Moreover, the image input into the ALF during the search phase is not the image reconstructed by the previous filter, as specified, but the image directly reconstructed from the residues. This choice arises from the fact that the previous filters, which are the Deblocking and SAO, also have a search phase that the Software team decided to perform in parallel. This parallelism only exists once the software encoder is modeled as a hardware encoder. The two phases can be modeled by a diagram.

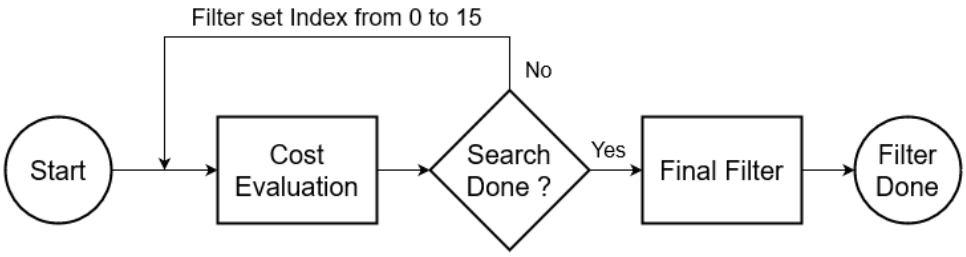

Figure 2.9: Block diagram of the ALF filtering phases.

A significant choice made by the Software team is to always recalculate the class and transposition indices for each filter set. This calculation is redundant and can be avoided. Another problematic choice is the concept of calculating neighboring pixels present in previous LCUs. The image is stored in its entirety in the software encoder at each filtering step. It is not practically acceptable to store everything in the hardware encoder. Therefore, an alternative method for describing the neighbors of processed pixels needs to be found. The diagram below represents a 128x128 pixel image split into 4 LCUs of 64x64 px each. Each LCU has a virtual boundary and areas to be reused when processing neighboring LCUs. A method to store these areas needs to be found. On the diagram below the data stored for lines is decreased thanks to the virtual boundary and padding processing.

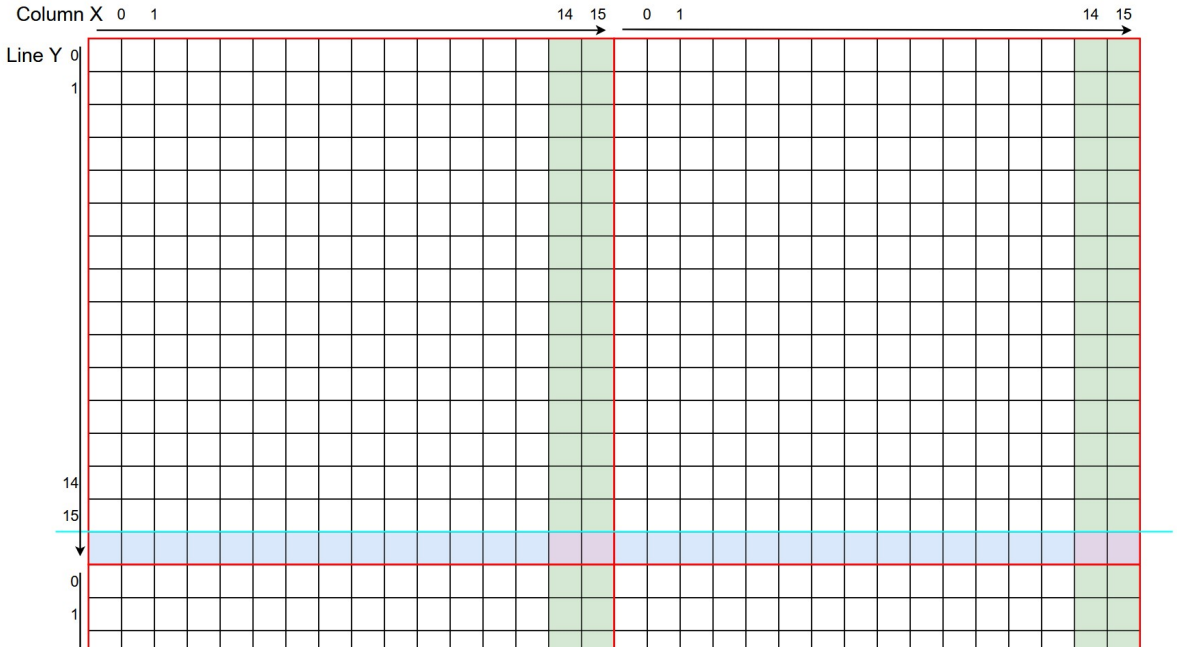

Figure 2.10: Neighboring LCUs and areas to be kept in memory.

2.3. Introduction to RTL Design

2.3.1. Transition from Software Development to Hardware

As previously mentioned, the work of RTL design from C++ source code requires a thorough analysis of rewriting methods. Currently, research into this rewriting, known as HLS for High-Level Synthesis, is struggling to advance in the field of ASIC synthesis (non-prefabricated or preprogrammable chips). Therefore, it is necessary to examine the tools available for correctly rewriting an encoder for hardware use.

2.3.2. Registers and Critical Paths

To more concretely describe what RTL design is, it involves the development of operation blocks with multiple inputs and outputs connected by wires. In the design, two mandatory inputs are added in the VHDL scripts that describe the architecture of these blocks: the reset signal and a clock signal. These two signals are directly connected to what are called registers, or flip-flop memories. The reset signal is a pulse that, once triggered, resets the memory of the registers. The clock signal is a periodic boolean signal that describes the duration for which a register will hold information in memory. Most registers retain the data in memory and block their inputs through a boolean input called enable. When implementing a flopped process in VHDL, an "if" condition will implicitly include the use of this enable signal. The idea is not to continuously write to memory. To avoid excessive power consumption of a chip, it is essential not to keep a register writing continuously. It should be able to "rest".

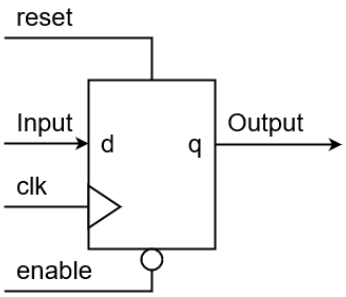

Figure 2.11: Diagram of a register.

A well-designed block consists of several registers, ideally at the input, output, and internally. They act as relays, intermediate points, or storage between a series of cables and logic gates known as combinatorial processes. Thus, an example of an RTL block can be described as follows.

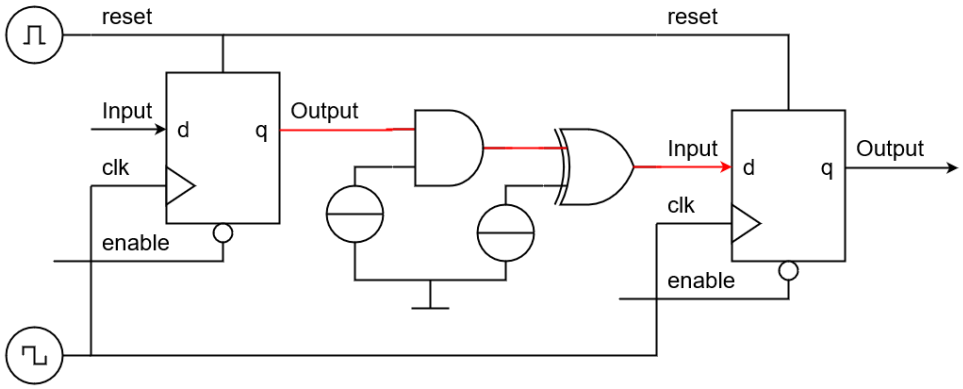

Figure 2.12: Diagram of a group of combinatorial operations, with the critical path in red.

This concept is a basis that must be developed to obtain functional circuits, even under signal propagation delay constraints or circuits resistant to information leakage. The first deepening concept, and the most important to understand, is that of the critical path. Between two registers, a succession of wires and logic gates forms a path. The propagation time of a signal through each path is not the same. But, as in a race, there is a time limit for all signals to traverse their paths. The path that takes the longest time to traverse is called the critical path. It is a bottleneck because it defines the time that other signals will have to propagate and, consequently, the clock period indicating to a register to store its input data. The challenge in RTL design is to avoid any limiting paths. For a predefined clock period, all signals must have time to pass from the output of one register to the input of another register, or the same one. Here, the ALF must have a clock frequency higher than 1 GHz, so a signal should not take more than one nanosecond to move between two registers, or even less considering a safety margin.

2.3.3. Pipeline and Valid/Ready Interface

To transmit a data stream without loss, a method known as pipelining is introduced. It relies on the presence of the enable signal in the registers. Not all blocks take the same number of cycles to perform a series of operations; this number varies between blocks and within a single block. Therefore, the concept of waiting is introduced using two input boolean signals and two output boolean signals, named ready_in, valid_in, ready_out, and valid_out. If block B has no ongoing operations, it will notify block A, via the ready_in signal, that it is ready to receive data to process. Only when block A communicates to B, via the valid_in signal, that it has data to transmit will B be able to process this data and, temporarily, declare that it is not ready to receive new data. A will

then no longer communicate valid data until it has processed new data. The two signals between A and B are independent. A could provide a valid value regardless of whether B is ready or not. This process is repeated between B and C with the ready_out and valid_out signals. If C blocks data transmission, B will have to wait to send it new data and process it, which will block it. And so on for A. Therefore, there is neither data loss nor excessive power consumption.

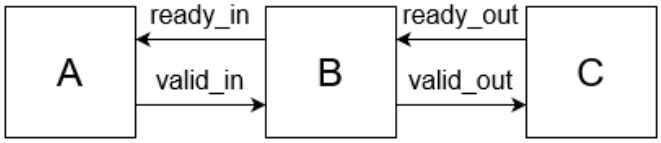

Figure 2.13: Diagram of the valid/ready interfaces between each sequential block.

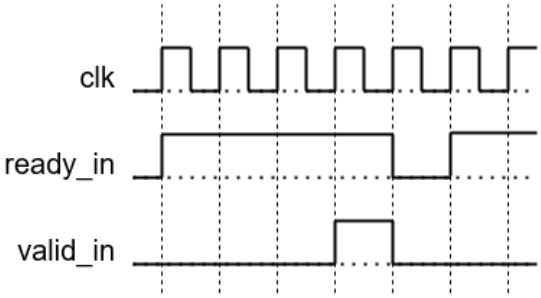

Figure 2.14: Timing diagram of B waiting for a valid data signal from A.

Internally within a block, the valid/ready description is always clock cycle by clock cycle. These are the only signals that are continuously updated in the registers, as they refresh the pipeline state. See A.1 for the basis of a VHDL script for a 2-cycle pipeline block.

2.3.4. FIFO Usage Concept

In general, the usage of registers to avoid overloading them involves placing them only at the output of blocks. In this case, the input data to a block can change even if the block has not had a chance to indicate that it wanted to keep that data. To address this, a register is placed at the input of the block, which takes any valid data and uses it for processing without announcing that the block is ready to receive new data. If the valid/ready interface of the previous block is correctly implemented, the valid input data will remain the same. Internally within the block, a new signal called "working" can be introduced to describe the duration during which the valid data is to be processed; the ready signal will then serve as an end indicator.

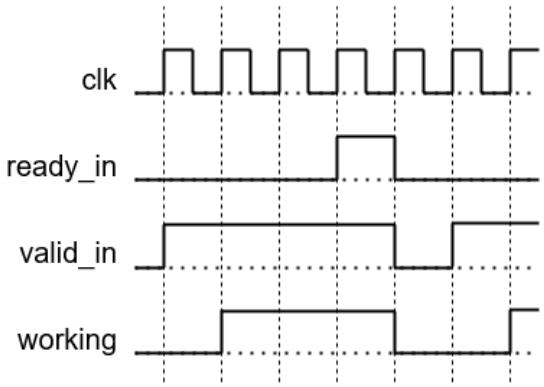

Figure 2.15: Timing diagram of a FIFO using the valid/ready interface.

2.3.5. RAM Usage Concept

So far, we have described the use of registers as practical and useful memories at each stage of the design. However, they are not suitable for holding a large amount of data. These registers take up space on the synthesized circuit. Therefore, it is necessary to introduce a new type of memory called RAM, which stands for Random Access Memory. RAM is characterized by a known number of words and a size for these words. For example, 40 words of 128 bits each. This could be 40 blocks of 4x4 pixels (16) with 8-bit luminance values.

This RAM is different from registers and thus has a separate design and a particular interface. In our case, we will only discuss basic RAM with single access. It is a memory with an input for data to be written and an output for data to be read. These two actions cannot be performed simultaneously and require the activation of boolean signals to indicate writing or reading. The RAM also has another signal that associates an address with the data being written or read. One should think of RAM as a huge cabinet with different drawers that must be located by an address.

It is noteworthy that this RAM also takes in the clock signal. This input justifies its operation. In the case of writing, during the first clock cycle, the data to be written, the address, and the write mode are specified. On the following clock cycle, the data will be written to the RAM. In the case of reading, during the first clock cycle, an output is specified, a register that will read the RAM, along with the address and the read mode. On the next clock cycle, the RAM will transmit the stored data to the output. However, it is on the following clock cycle that the output must be recorded in a register to use the data without causing critical paths. Accessing external RAM takes time, hence the importance of not directly using the output of a RAM.

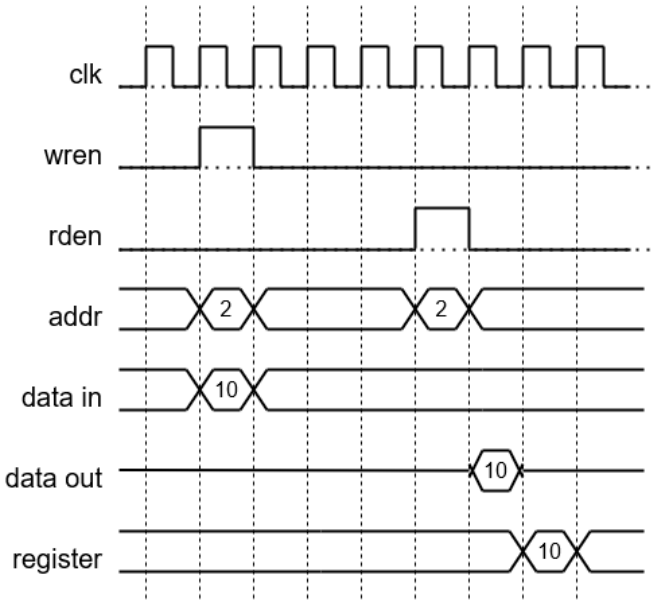

Figure 2.16: Timing diagram of RAM write and read operations.

2.3.6. FSM Usage Concept

Performing simple operations continuously, one after the other, is often not enough; sometimes more variability is needed. One might want to describe different states of a system so that it performs various actions intelligibly over a given period. Each state of a Finite State Machine (FSM) describes a different behavior depending on the state it is in. The state of the system depends on its state in the previous clock cycle and certain conditions. One should view the state of the machine as a named register, which, depending on the value of the register, selects different computations.

Most often, an FSM describes a system that waits for a request. It starts in its initial state, receives an action order, and then, once the action is completed, the system pauses, waiting for a new action order.

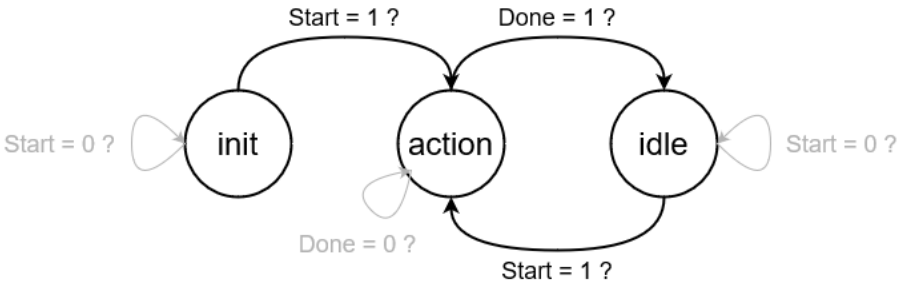

Figure 2.17: Block diagram of a basic Full State Machine.

2.3.7. Testbench and DPI Library

Finally, once the RTL design is completed, it is necessary to validate it by creating a Testbench. A Testbench acts as a wrapper or packaging. A wrapper instantiates all the final elements necessary to realize a specific design. It typically groups high-level blocks with RAMs via interfaces specified in the inputs and outputs. A Testbench does roughly the same thing, except it includes a notion of temporality. A Testbench can wait before providing values to wires and can generate a clock signal. A Testbench includes code that is not synthesizable. In practice, the clock is a component separate from the RTL design, even though its signal is connected to all blocks. Thanks to this notion of temporality, one can generate a clock as well as a reset signal that will be triggered at the start of a component under test.

Creating a Testbench in SystemVerilog is preferred. This language allows for two essential manipulations for verification. The first manipulation is reading and writing internal files during the simulation of the Testbench on software like QuestaSim or VCS. It is possible to retrieve the inputs of a block under test from a text file with specific functions (such as $fscanf). Similarly, it is possible to write the outputs of a block using other functions (such as $fdisplay). Thus, one can compare text files, called traces, generated by the software encoder with traces generated by the tested block. If the text files in both codes are structured correctly, it is enough to run the command "diff blocktested.txt codesoft.txt" to see errors directly and visually.

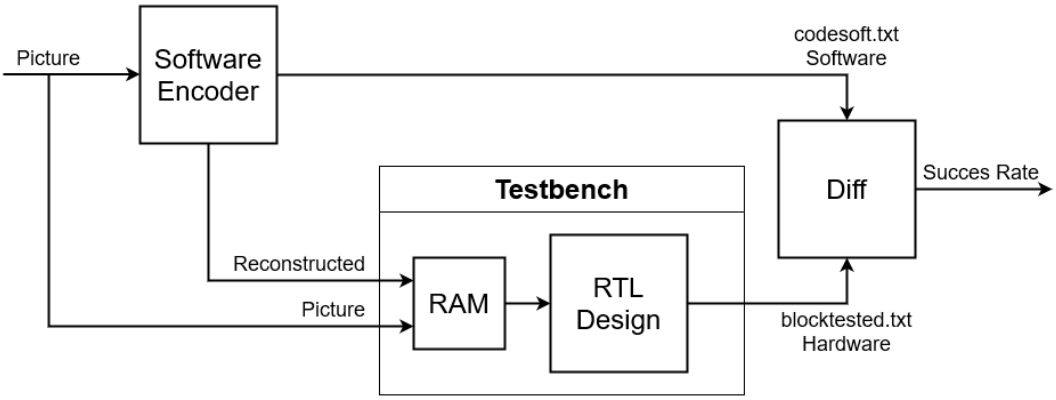

Figure 2.18: Block diagram of validation by traces.

This comparison concept is also found in the second manipulation. A library available in SystemVerilog, called DPI (Direct Programming Interface), allows the simulation of C++ functions directly within the hardware description code, as if it were a real block. Thus, this block is obviously not synthesizable, but one can compare its inputs and out-

puts directly in the Testbench, without using external files.

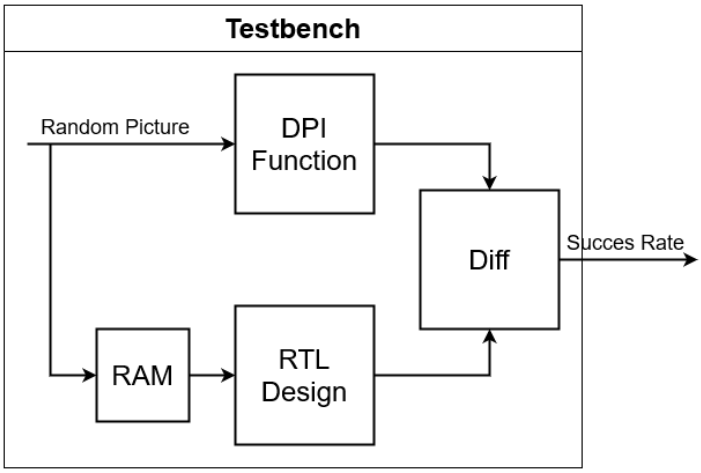

Figure 2.19: Block diagram of validation by DPI.

3 | Realization of ALF

3.1. Function Decomposition

3.1.1. Global Structure of the ALF

We now have all the tools to realize the ALF: the specifications, the technical choices of the Software team to follow, and the knowledge reminders to create a functional RTL design.

The first approach to take is to clarify the implementation steps, each step justifying the creation of an RTL block. Following the approach of the software encoder, we divide the filtering into two phases: an initial research phase and a final filtering phase.

To begin with, the first phase proceeds as follows. We traverse the 64x64 pixel LCU through the image in raster mode, from left to right and from top to bottom. We calculate an initial reference cost without ALF filtering by measuring the distortion between the LCU received at the ALF input and the original LCU at the encoder input. Then, we determine class and transposition indices for the 4x4 pixel blocks of the LCU (15x15 blocks) and filter them with the first set of filters (excluding the last column and last row of blocks to avoid unknown neighbors to the right and bottom). We calculate an initial cost between the filtered LCU and the source LCU. We repeat this process: determining indices, filtering, and calculating the cost for the 16 filter sets. We keep the best filter set giving the lowest cost. Finally, we compare the best filtering cost with the reference cost to decide whether to filter the LCU in the final phase. As mentioned previously, repeating the index determination step is not meaningful, so we will only do it once for simplicity. An essential but not described step is the transition from the LCU to 4x4 blocks with neighbors allowing index calculation and filtering. Therefore, we need to consider a step of cutting the 64x64 pixel LCU into 10x10 blocks around the 4x4 blocks. A realistic design cannot transport a complete LCU in registers. Ideally, this cutting block will be the conductor of the ALF, which we will describe later. In conclusion, we obtain a preliminary diagram of the first phase.

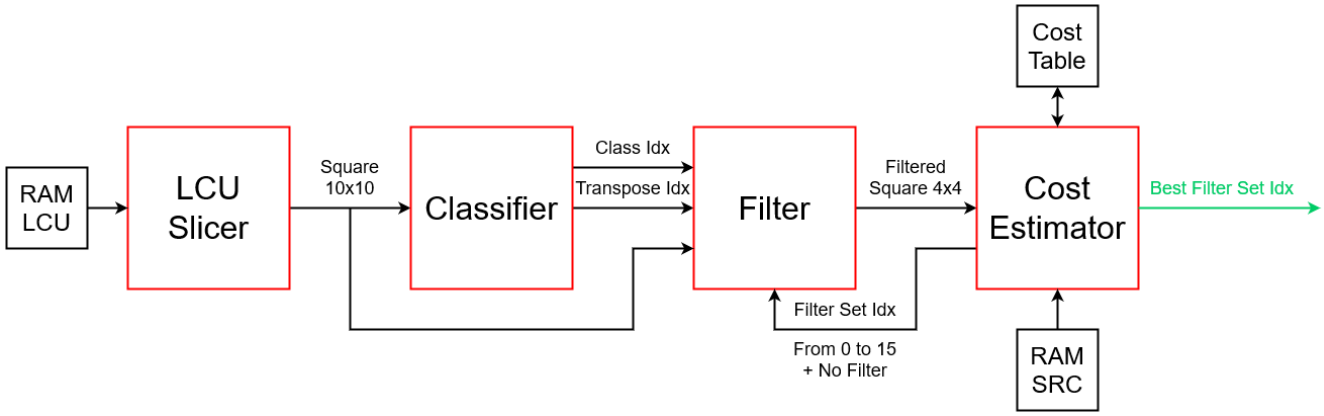

Figure 3.1: Block diagram of the ALF research phase.

3.1.2. Final Phase Filtering

The second phase, called the final phase, is much simpler. It retains the principle of cutting 10x10 pixel blocks from the LCU, determining class and transposition indices, and filtering. It is preferable to recompute these indices rather than occupying space in a RAM or registers to store all the indices calculated in the first phase or, worse, the complete filter results. The only index retained is the filter set, which is unique within an LCU. This results in an even simpler diagram.

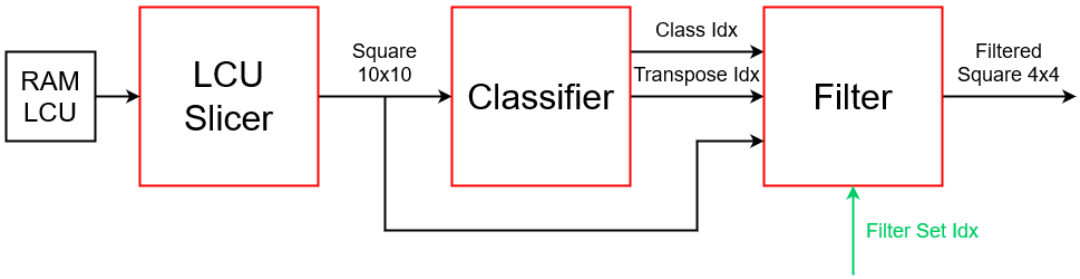

Figure 3.2: Block diagram of the ALF final filtering phase.

From now on, to avoid repeating the term "4x4 pixel block" everywhere, we will refer to it as a CU (Coding Unit). We will assume that the CU will always be 4x4 pixels and that the LCU will always be 64x64 pixels.

3.1.3. RAM Layout for Classified and Filtered Elements

To transmit the data necessary for filtering, a controller is needed to sequentially send 10x10 pixel blocks surrounding each CU in the LCU. A RAM is necessary to send

fixed-size words representing contiguous blocks that, when assembled, form the LCU. Therefore, the blocks must have a width and height that are multiples of 64 pixels to avoid using excessively large registers or RAM, while still processing these blocks in a reasonable time. We will use 256-word RAMs representing the luminance of 4x4 pixel blocks, with a parameter specifying the number of bits (8, 10, or 12). There are 256 4x4 pixel blocks, or CUs, in a 64x64 pixel LCU, which means 16 columns and 16 rows. By default, the values stored in a RAM are sorted in a Z-scan order.

272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289
256	0	1	4	5	16	17	20	21	64				80			85	
257	2	3	6	7	18	19	22	23								87	
258	8	9	12	13	24	25	28	29								93	
259	10	11	14	15	26	27	30	31								95	
260	32	33	36	37	48				96				112			117	
261	34	35	38	39												119	
262	40	41	44	45												125	
263	42	43	46	47												127	
264	128				144				192							213	
265																215	
266																221	
267																223	
268	160				176											245	
269																247	
270																253	
271	170	171	174	175	186	187	190	191	234	235	238	239	250	251	254	255	

Figure 3.3: Representation of Z-scan indices of an LCU.

This order aims to avoid calculations of the form 16*y + x to obtain a CU's linear position. Indeed, a RAM address is a single number in one dimension, so there is no notion of abscissas or ordinates. By knowing the X and Y coordinates of a CU, and converting these coordinates to binary, we can interleave the bits to obtain its Z-scan address. For example:

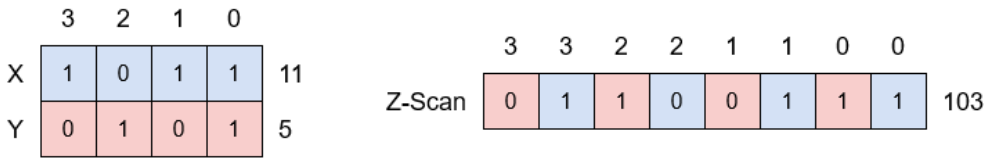

Figure 3.4: Representation of Z-scan indices for a CU (X = 11, Y = 5).

We now know the RAM model to use for storing the data of the current LCU. We can transition from one CU to another to produce 10x10 blocks. An optimal method involves traversing the LCU in a serpentine fashion. Starting from the corner with coordinates X=0, Y=0 of the first LCU, we will first create a 3x3 CU block, which requires 9 clock cycles to retrieve the values from the RAM.

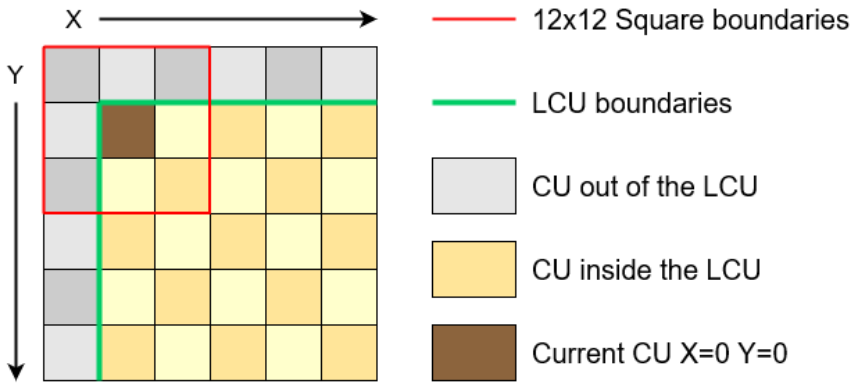

Figure 3.5: Generation of the first 10x10 block from CU(X = 0, Y = 0).

We will then obtain a 12x12 pixel block, which can be cropped into a 10x10 pixel block centered on the CU being processed. However, when considering the neighboring CU by reading from left to right, it will be observed that 6 of the CU blocks on the far left are already known. Therefore, only 3 new CUs need to be retrieved from the RAM to construct a 10x10 pixel block.

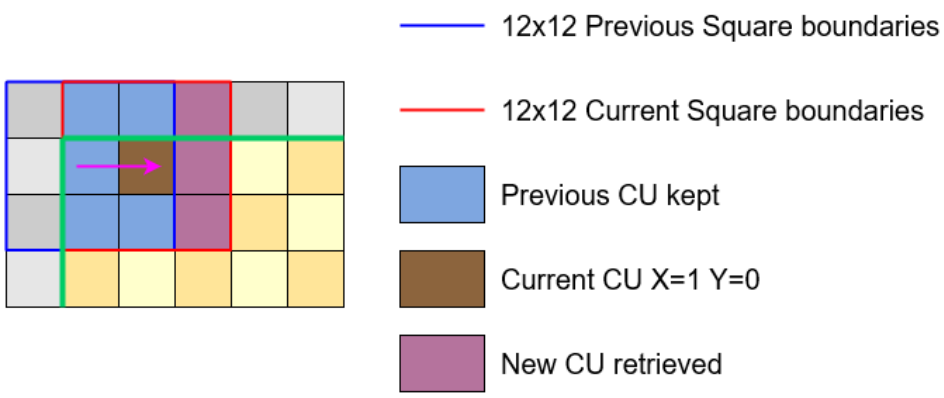

Figure 3.6: Generation of a second 10x10 block from CU(X = 1, Y = 0).

Upon reaching the end of a line, the only way to continue this method is not to return to the beginning of the second line on the left but to start from the end and construct the 10x10 pixel block with the 3 unknown CUs located below.

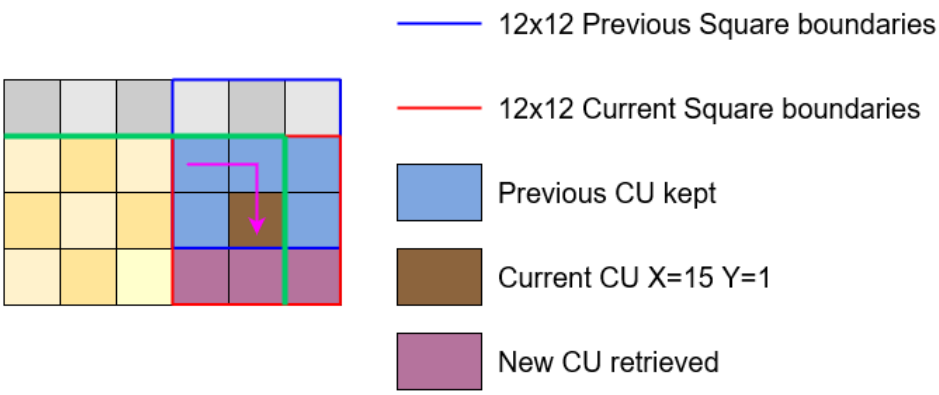

Figure 3.7: Generation of a 10x10 block at the end of the line from CU(X = 15, Y = 0).

Gradually, a pattern emerges that is well-described by a FSM (Finite State Machine). This pattern is serpentine in shape, and it can be managed by setting its starting point and the moments when it should turn. Before moving from one CU to another, a counter counts from 0 to 2. This counter accesses 3 CUs stored in the RAM, one after the other.

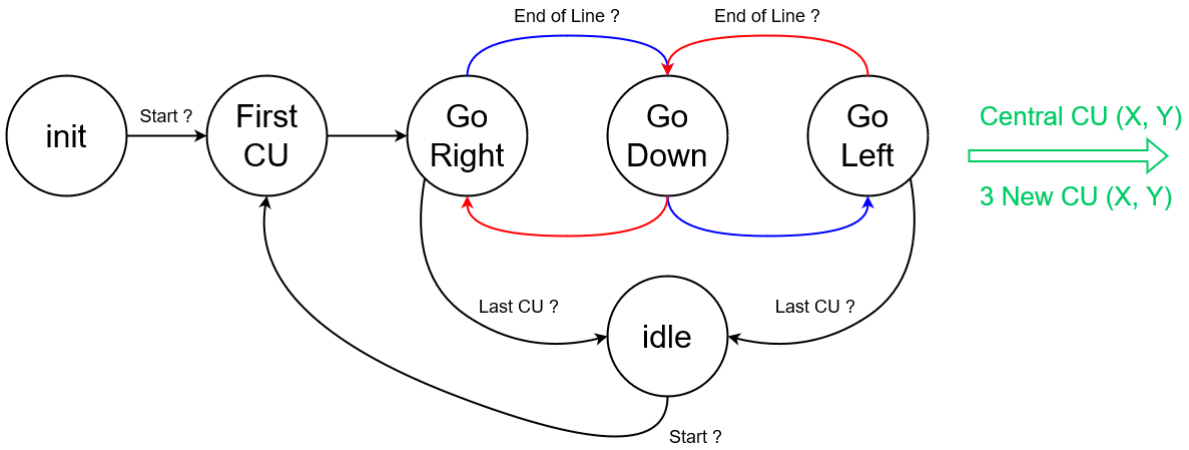

Figure 3.8: Finite State Machine for generating 10x10 pixel blocks.

However, this is not enough; it is necessary to define what happens when the current CU is at the edge of an LCU or the image. As explained earlier in the specification, in the case of an image edge or a virtual boundary, missing CUs will be replaced by padding.

However, in the case of an LCU boundary, it is necessary to know all the neighbors. Consider an LCU located in the second row and second column, with coordinates X = 1, Y = 1. Its first CU at position X = 0, Y = 0, in the top-left corner, will need to know some values from the LCU above, the LCU to the left, and the LCU in the top-left corner.

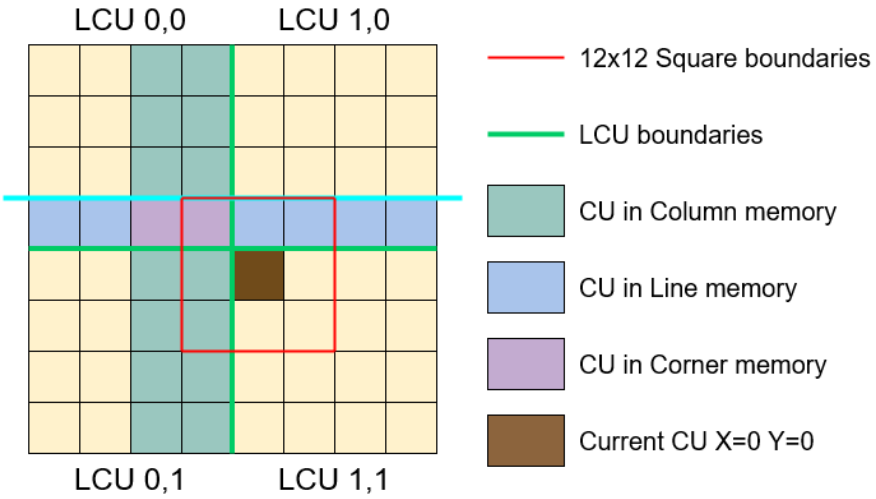

Figure 3.9: Generation of the first 10x10 block from CU(X = 0, Y = 0) considering neighbors.

If we now want to construct a 10x10 block for the CU at coordinates X = 15, Y = 0, it will be necessary to know the pixel values on the right that will appear in the next

LCU. Therefore, we will have to wait to be in that LCU to process the previous CUs. Thus, even more data needs to be retained. In the case of a CU at coordinates X = 10, Y = 15, the values of the pixels below will only arrive after passing a complete line of LCU. However, less information will need to be stored due to the padding of the virtual boundary.

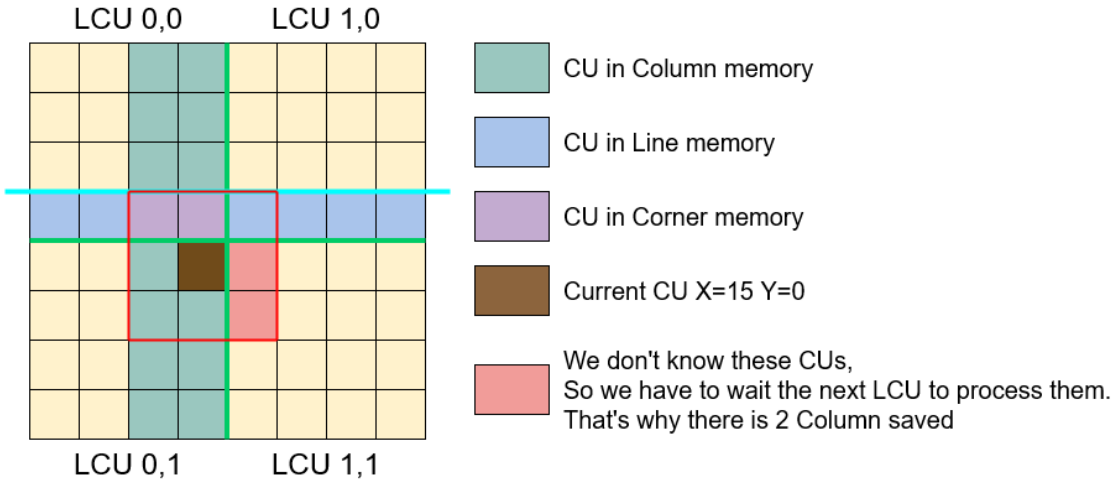

Figure 3.10: Generation of a 10x10 block at the end of the line considering neighbors.

In summary, it is necessary to instantiate a RAM for neighbors, which will store all the last lines of CUs from a full image line, 2 columns of CUs in an LCU, and 2 CUs at the corners of an LCU. Given that a RAM is fixed, it should be sized for the case of a 4K width image, i.e., 64 LCUs. The following RAM model will be used:

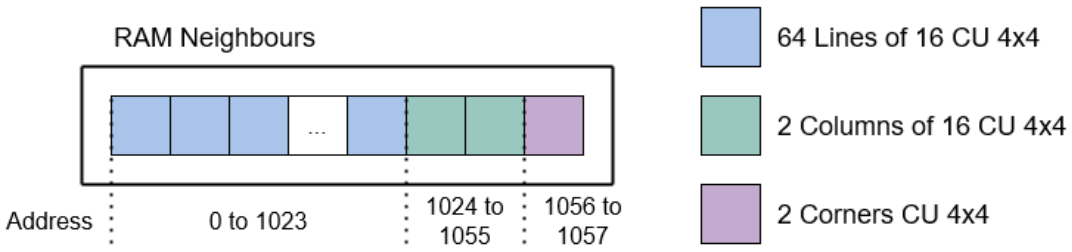

Figure 3.11: Memory address layout of the neighbor RAM.

The FSM remains unchanged. If the coordinates of a CU are negative and outside the current LCU, it will be sufficient to fetch from the neighbor RAM; otherwise, data will continue to be fetched from the current LCU RAM. However, an additional step of writing to the neighbor RAM will be required.

The final RTL block implementation involves adding wait states using a valid-ready interface and integrating additional states into the FSM. It will also require slicing the 12x12 block into 10x10 pixel blocks, not forgetting the potential padding. This results in the conductor of the ALF. In the search phase, the serpentine pattern will propagate from the CU at coordinates X=0, Y=0 to the CU at coordinates X=14, Y=14, following the software encoder's guidelines. In the final filtering phase, the coordinates for starting and ending each CU for a given LCU can be described in a table, based on its position in the image.

LCU Locations	Departure point	Arrival point	Min Col	Max Col	Min Row	Max Row
X=0, Y=0	0, 0	14, 14	0	14	0	14
X=max, Y=max	-1, -1	15, 15	-1	15	-1	15
X=0, Y=max	0, -1	14, 15	0	14	-1	15
X=max, Y=0	-1, 0	15, 14	-1	15	0	14
X=0	0, -1	0, 14	0	14	-1	14
Y=0	-1, 0	14, 14	-1	14	0	14
others	-1, -1	-1, 14	-1	14	-1	14

Table 3.1: Table of relative coordinates for a 10x10 block generated from the current LCU.

3.1.4. Classification of a 4x4 Coding Unit

Once the 10x10 pixel block is generated with the coordinates of the block to be processed, all the calculation steps are provided as specified. The gradient formula for a pixel located in an 8x8 pixel block is calculated every two cycles. Thus, there are a total of 32 local gradients for each of the 4 directions. These values are then summed up. In calculating a local gradient, the central pixel of the gradient is multiplied by 2. This multiplication does not affect the length of the paths between registers, as it is simply a concatenation with a 0 at the end of the binary representation of the number. There is an addition of a bit but no logic gate. The 2 subtractions and the absolute value function require 3 addition operations. To optimize, it is advisable not to exceed 4 additions in one cycle to avoid a critical path. It is noted that to sum 32 values, an adder tree needs to be created, where the longest addition path traverses 5 adders (taking the log2 of the number of additions). An important note during design is that the sum of 2 eight-bit

numbers can yield a result up to 9 bits. Therefore, an additional bit must be provided at each adder stage.

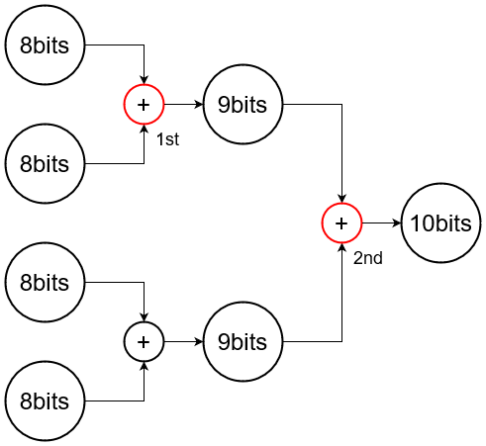

Figure 3.12: Example of an adder tree for summing 4 eight-bit integers.

To avoid a limiting critical path and reduce the size of the registers used, during the first cycle of the block pipeline, the local gradients and the first stage of the summation will be calculated. In a second cycle, the remaining 4x16 addition operations will be performed. This results in the initial block for gradient determination.

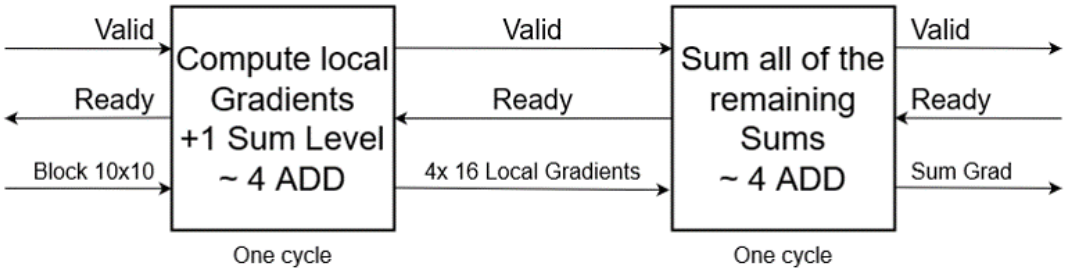

Figure 3.13: Block diagram of the gradient calculation phase.

After designing a block that processes a CU in 2 cycles to obtain its 4 gradients, a block is created to model the algorithm. This algorithm was rewritten by the Software team to include the values of the 2 thresholds and to avoid the divisions proposed by the specification. Ultimately, the algorithm is divided into 2 steps: a parallel calculation of D and A, followed by a calculation of C. Each of these 2 steps involves a multiplication. Multiplying by a number that is not a power of 2 is a significant logic element. Therefore, it is preferable to divide these 2 parts of the algorithm into 2 cycles. This results in a final block that determines the class and transposition indices of a CU based on the previously generated 10x10 pixel block. This block is constructed using a valid/ready interface that

forms a pipeline. Even though it takes 4 latency cycles to determine the indices, it can determine new indices in each subsequent cycle after a simple delay.

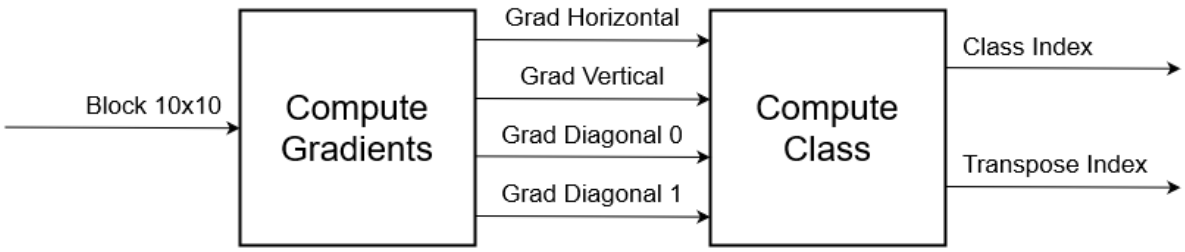

Figure 3.14: Final block diagram for determining class and transposition indices.

The block is complete but not yet validated. To validate it, the function is described more or less in the source code of the software encoder. The difficulty lies in the fact that the software classifies a complete LCU all at once, producing an index array, while the RTL design only determines indices for one CU at a time. Therefore, a testbench will be created to generate a random 64x64 pixel LCU and apply the classification block to a single CU. By instantiating the C++ function via the DPI library, we will be able to determine the classes for the LCU and retain only a single pair of indices for the targeted CU. The testbench will compare the results for the same CU with random values, whether the CU is in the middle of an LCU or on the edge, between the DPI function results and those of the RTL block. After testing 100,000 combinations, everything works correctly, and the block is thus validated.

3.1.5. Filtering a 4x4 Coding Unit

Just like with the classification block, the filtering block receives the generated 10x10 pixel block and its coordinates. The design will be such that the filtering block receives the previously determined indices along with the delayed pixels. With an additional input for the filter set, we get the initial description level. This level includes the coefficient table and the filter for a complete CU.

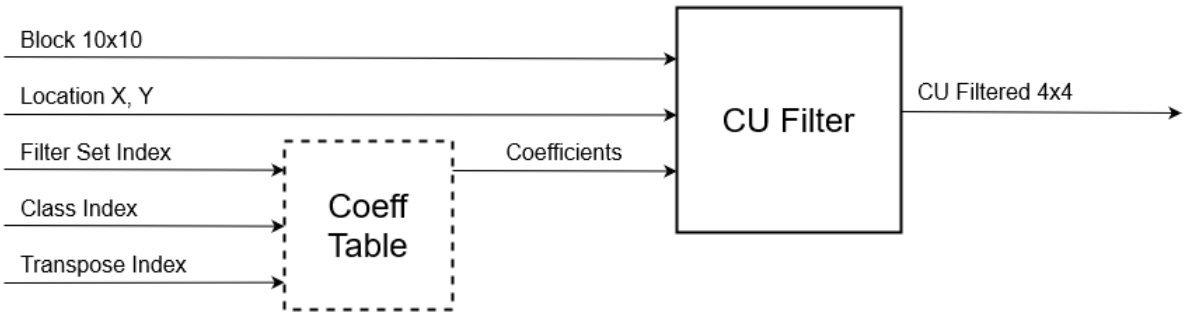

Figure 3.15: First level of description of the filter block.

Before addressing the filter for a complete CU, we start with the filter for a single pixel. We follow the formulas from the specification. We need the coordinates of a pixel in a CU, the group of 25 pixels in the filtering diamond, the 12 coefficients in transposition order, and information on whether the pixel is close to a virtual boundary. This boundary information will help choose between using the parameter 2^7 or 2^{10}, as specified in the documentation. In the diagram below, 'side0' and 'side1' represent the 2 opposite pixels in the diamond, which are multiplied by the same coefficient. There are then 12 weights represented by the pixels, multiplied by 12 coefficients, and added. The addition varies depending on the boundary condition.

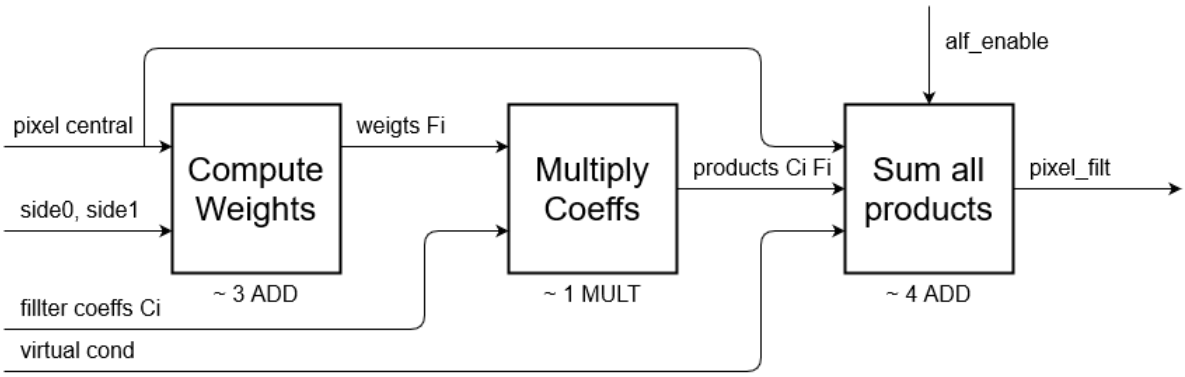

Figure 3.16: Block diagram of a single pixel filter.

At this point, we need to describe the parallelism and sequencing for filtering the 16 pixels of a CU. We choose to split the process into two parts to reduce the number of elements without extending the processing time. In 4 cycles, representing the 4 columns of a CU, 4 pixel filters will process the entire CU. In 4 cycles, not of latency but of pipeline waiting using a FIFO, we approach the time required to generate a 10x10 pixel block, which is therefore not too restrictive.

We will keep the coefficients and pixel inputs to the filter for 4 cycles to filter each column one after the other. The advantage of splitting into columns rather than rows is to simplify the padding step. For the same CU, the padding conditions applied near a virtual boundary will be the same for each pixel filter during the 4-column period. In short, the CU filter is a parallelization of 4 pixel filters, with the arrangement of received pixels depending on a counter ranging from 0 to 3 in a loop.

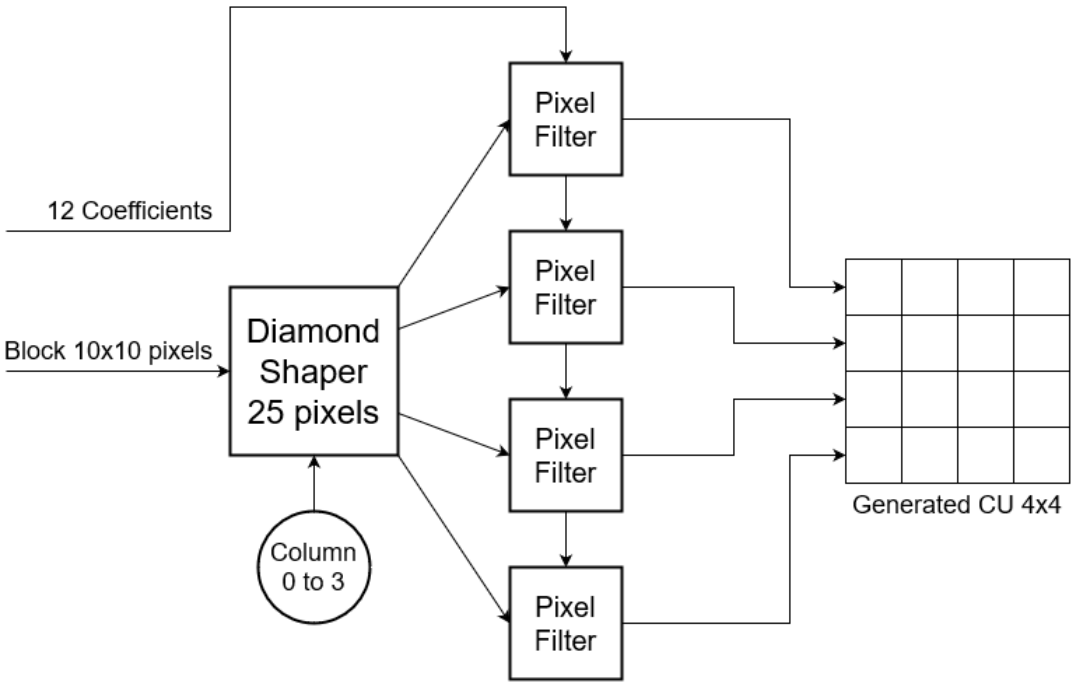

Figure 3.17: Block diagram of a 4x4 pixel CU filter.

The validation phase is the same as for the classification block. Filtering is performed on a complete LCU in the software encoder. Thus, we need to select a single CU from the filtering process to compare it with the output of the RTL block. Similarly, we perform several thousand combinations before considering the block validated.

3.1.6. Cost Calculation of Filtering

The cost block is divided into two levels: a pure cost calculation block, which we define as distortion, symbolized by the sum of squared differences of the 16 pixels in a CU, and a block that connects everything, makes decisions, and adds up all the costs for the complete LCU. In the context of the project, we decided not to implement the part related to encoding the filtering indices.

First, the distortion calculation block can be described in 2 cycles. The first cycle

computes the difference and square of each pair of pixels compared between filtered and source images, and the second cycle sums up the 16 local costs. Note that this is a pipelined block, so it can provide a new result every cycle.

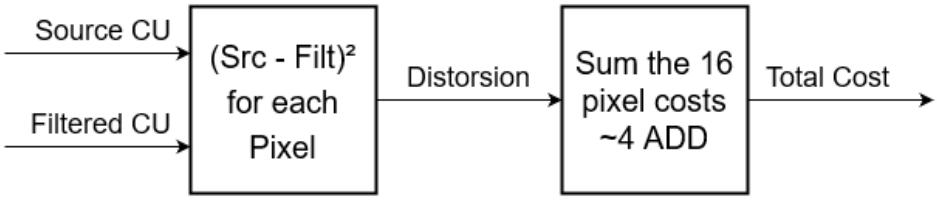

Figure 3.18: Block diagram of the cost/distortion estimator.

The cost comparator block must progressively calculate all costs for each CU in an LCU and for each set of filters. We have the option to test the 16 sets of filters one after the other, using a counter, similar to the design of the 4-column filter. We then store the results in a 17-value register: the reference cost without filtering and the cost for each of the 16 filter sets. As we go along, we compare the costs to determine the best filter set, as the software encoder would.

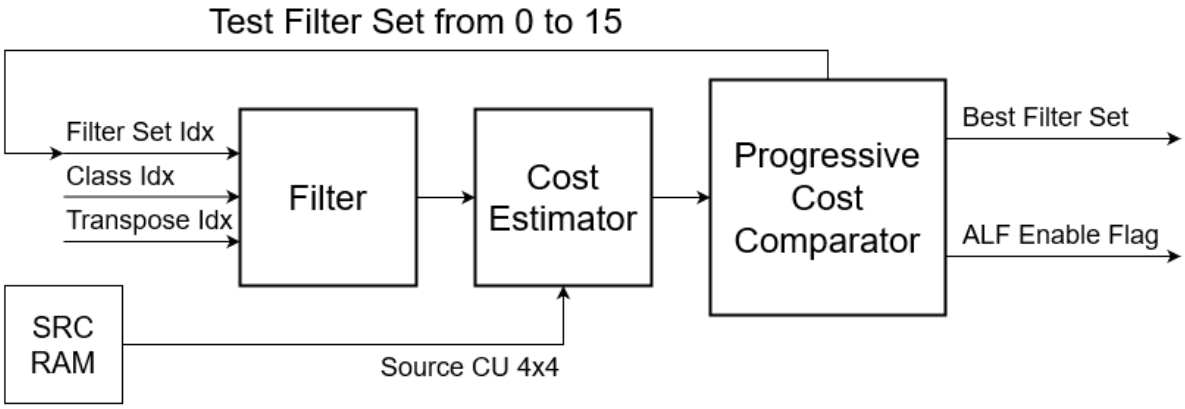

Figure 3.19: Block diagram of the sequential cost comparator.

In total, 17 filterings + 1 final 4-cycle filtering are required to obtain a correctly filtered 16-pixel CU. Thus, approximately 16 pixels can be filtered in 4x18 cycles, yielding a rate of 0.22 pixels filtered per cycle, which is well below the 2 pixels per cycle required by the specifications.

The only solution to meet the requirements while retaining what has been accomplished so far is to test the 16 filter sets in parallel, along with calculating the reference cost without filtering. This approach will involve 2 filtering stages: one to find the best

filter set and one for the final result. This way, we can filter approximately 16 pixels of a CU in 4x2 cycles, thus meeting the specification of 2 pixels filtered per clock cycle.

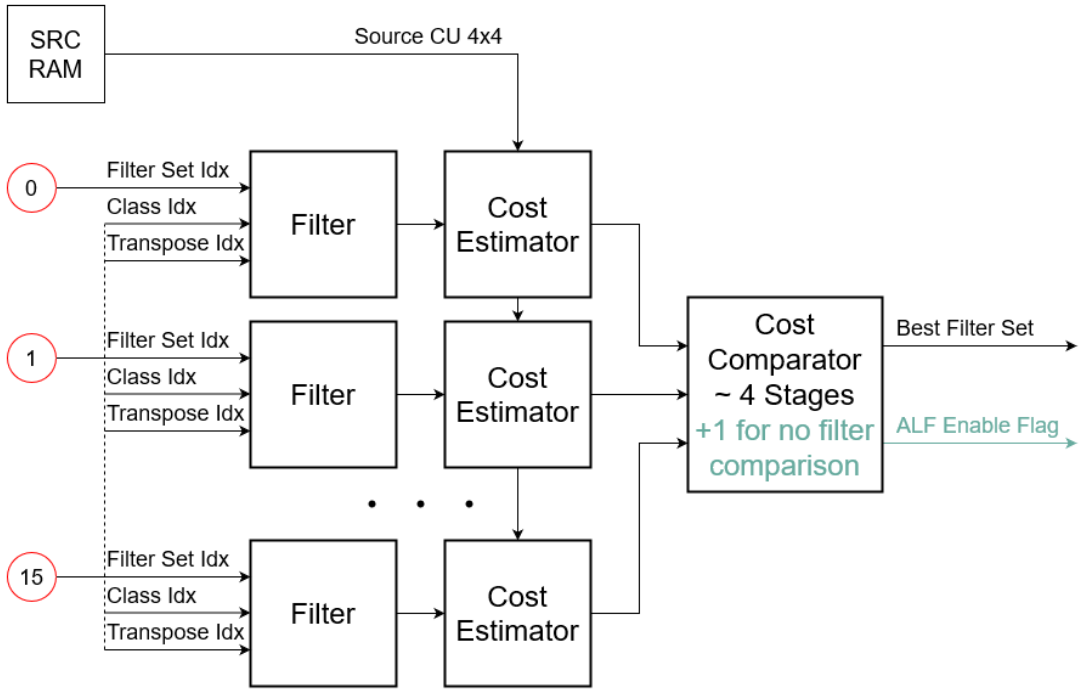

Figure 3.20: Block diagram of the parallel cost comparator.

3.2. Assembly of the ALF and Synthesis

3.2.1. Assembly of the Blocks

To assemble all the blocks and perform two phases—search and filtering—the orchestrator needs to be adjusted. Recall that the orchestrator is the block (FSM) responsible for slicing the 10x10 pixel squares. We introduce a new boolean signal for the cost evaluation phase. When this signal is active, the serpentine will propagate over a limited section of the LCU, as described in its section. Furthermore, we will not enter the writing states of the neighbor RAM during this phase, as this RAM should only be rewritten once per new LCU, at the end of all processing. After the best filter set has been evaluated, the block will send a signal to indicate the end of the search phase. Only after restarting in the final filtering phase will the cost evaluation signal be inactive, and we will write to the neighbor RAM as planned.

The output of the block determining class indices and transposition indices will be connected to the cost comparator during the search phase and then to a conventional filter during the final filtering phase.

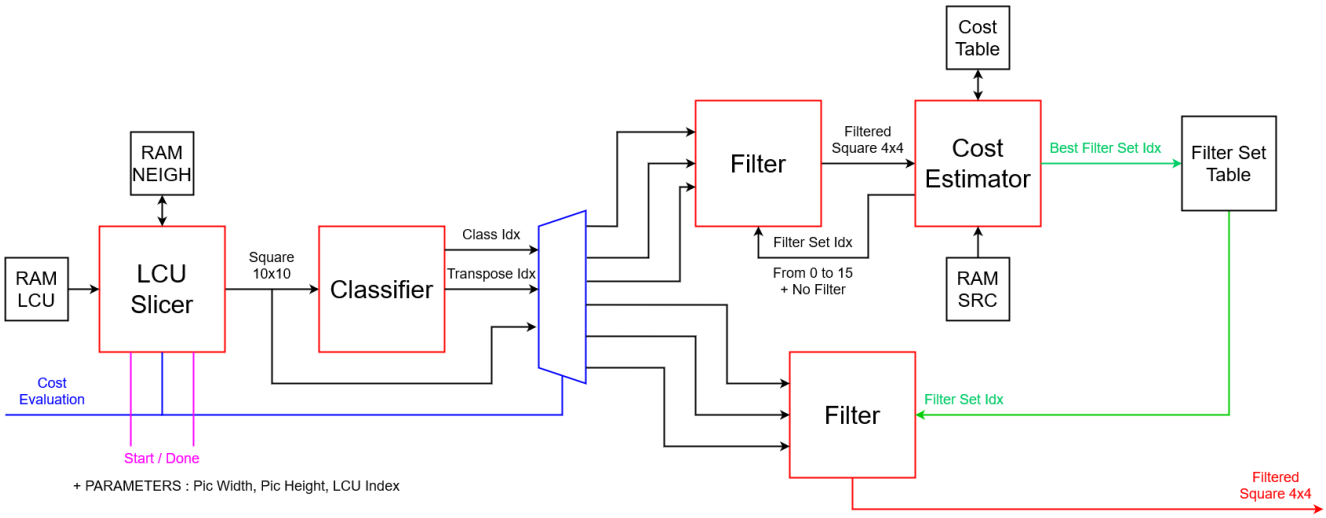

Figure 3.21: Final block diagram of the ALF.

3.2.2. Creation of the Final Testbench

Once the complete ALF is designed, a final testbench is necessary to validate it. This will be done using traces of the ALF's inputs and outputs generated by the software encoder. The testbench is similar to the one used for comparing filter sets. The traces will be simulated as RAM inputs and outputs of the RTL design of the ALF. An additional step is included in the testbench to verify each CU internally, to avoid running a differentiation command outside the script. Using a Python script, the system that generates traces is generalized and creates a new testbench for a given image. The only parameters to modify in the testbench configuration and in the software encoder configuration are the image dimensions and the link to the image.

With the generation parameters, the testbench calculates the number of LCUs to process. It will then activate the FSM for each new LCU it processes, in cost evaluation mode followed by final filtering mode. During the final filtering phase of each LCU, the LCU will send a signal to validate a newly filtered CU. The testbench will then perform a verification and provide a success rate. The ALF will be validated for a 100% success rate with stress images, common video images, images of sizes not aligned with the 64x64 LCUs, and finally, maximum image sizes of 4096x4096 pixels.

3.2.3. Synthesis Results and Expectations

Once the ALF is validated, it will be synthesized to estimate its size when implemented on a real circuit. The synthesis will also indicate if the design has any critical

paths that exceed the time allocated by the clock frequency. An assumed clock frequency of 1.4 GHz will be considered.

In total, three main syntheses are performed:

- **Sequential Cost Calculation Synthesis**: This will determine the minimal achievable size of the design.
- **Parallel Cost Calculation Synthesis**: This will check for critical paths while adhering to the specifications.
- **Final Parallel Search Phase Synthesis**: This is considered final and addresses timing issues caused by critical paths.

The results are displayed by the company's internal interface.

Gated Reg	Timing	Area (um²)
98.93%	0.09	10640

Figure 3.22: ASIC synthesis summary of the ALF with sequential search phase.

Gated Reg	Timing	Area (um²)
99.02%	0.24	53810

Figure 3.23: ASIC synthesis summary of the ALF with parallel search phase.

Gated Reg	Timing	Area (um²)
99.02%	0.01	59225

Figure 3.24: ASIC synthesis summary of the ALF with critical path resolution.

One piece of data, 'gated reg', indicates the proportion of registers that are not constantly used, which does not consider permanent power issues. Another data point

shows the maximum time exceeded in µs for a signal propagating through a critical path. Details include the starting register, ending register, and the logic gates used, see Appendix 2. The final interesting data is the size of the RTL design once it is synthesized. This size should be compared to other In-Loop Filters to gauge its scope.

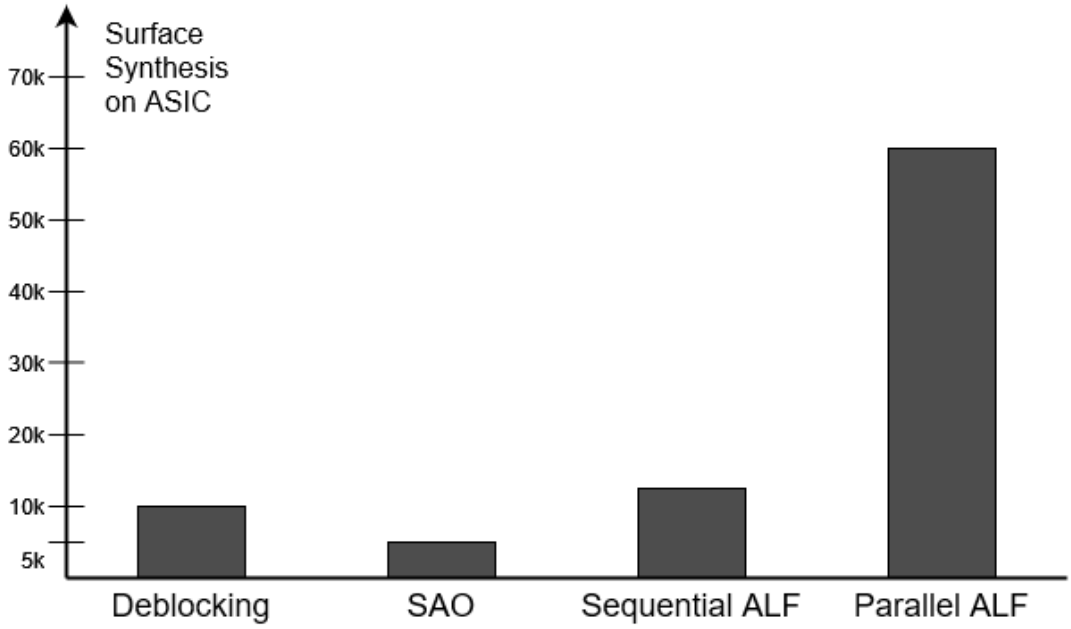

Figure 3.25: Surface areas of VVC encoder In-Loop Filters compared to the ALF.

Even though there are no specific size requirements for the final ALF, it is clear that its size significantly exceeds that of other filters. Future considerations will include optimizations. Visually, it quickly becomes apparent that optimization will involve determining what should be parallelized and what should be sequentialized.

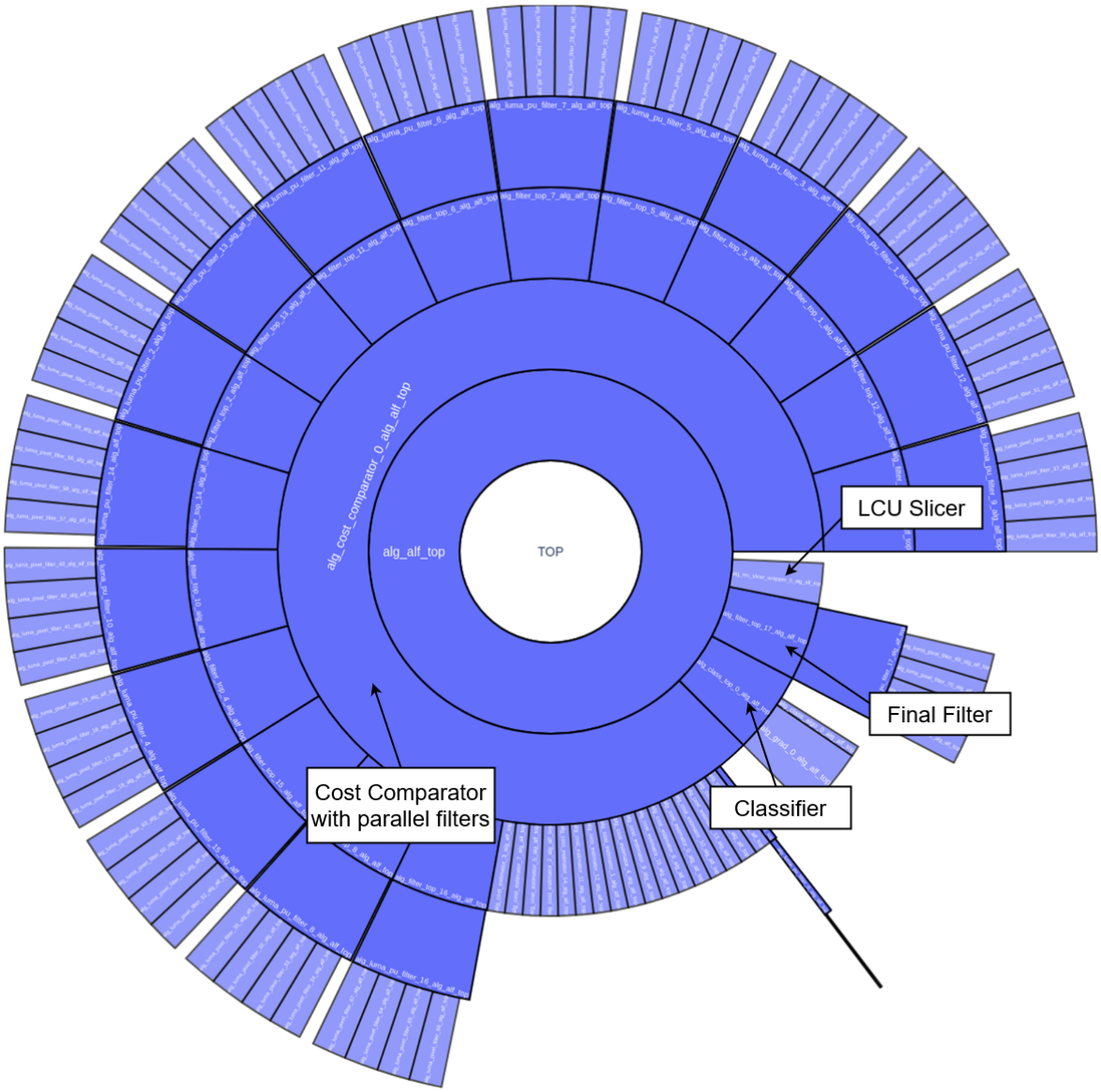

Figure 3.26: Area distribution of the ALF with parallel search phase.

3.2.4. Future Implementation Tasks to Consider

Given the constraints of the internship, there is limited time to develop the remaining implementation steps for the video encoder or to perfectly optimize the design size without interfering with the functionality of surrounding blocks.

One potential solution to reduce the size of the ALF is to find a balance between sequential and parallel cost calculations. To achieve this, it would be beneficial to reduce the CU filtering duration to a single clock cycle instead of four, by parallelizing 16 pixel filters. This would require generating a new 10x10 pixel square every clock cycle, rather

than every three cycles. Consequently, it would be necessary to partition the ALF input RAM and the neighboring RAM into multiple sections. Essentially, instead of storing 4x4 pixel squares in a single RAM, they would be stored across 4 RAMs, creating a sort of checkerboard pattern.

However, such optimizations do not account for implementation constraints. A single-access RAM presents limitations. If the SAO (Sample Adaptive Offset) is writing to one RAM while the ALF is attempting to read a CU from the same RAM, there could be conflicts. Each filter would have to wait for the other, which may lead to potential issues that need to be managed in the future.

Additionally, the use of control signals for the ALF needs to be considered. It must be determined whether the start and end processing signals for an LCU can be effectively utilized. This might necessitate modifications to the blocks already implemented for the encoder.

Conclusions

The implemented ALF is functional and meets the specifications; it can filter 2 pixels per clock cycle at a frequency of 1.4 GHz. The ALF has been verified using a testbench that compares its filtering results with those obtained from the software encoder, across a wide range of images. Since the encoder itself was validated beforehand, the ALF is therefore validated. However, there are two issues remaining: the synthesis area taken up by the ALF is too large, and the current implementation is not feasible.

The challenges encountered during the internship mainly involved understanding the source code of the software encoder. The encoder uses numerous C++ classes and objects that are not described. It was necessary to spend time manipulating them and examining other scripts to understand their functionality and their place in the code hierarchy. The concept of DPI was also new to me, and I had to get acquainted with it. All of this contributed to my improvement in C++ coding as well as my research and analytical skills.

During the internship, I was able to deepen my knowledge of video encoding and compression, as well as RTL design implementation concepts. This internship is comprehensive and provides solid practical experience. To accomplish the ALF, I had to demonstrate autonomy as well as curiosity. I questioned my colleagues within the company and learned from them how to optimize my code and visualize its implications.

In conclusion, this internship provided me with valuable work experience that will benefit me throughout my engineering career. One should not view this final internship as merely an exercise to be completed, but as an opportunity to learn to work independently, in good conditions, and to develop professionally in the field, here in RTL design.

Bibliography

[1] F. HHI. Versatile video coding (vvc) software, 2020. URL https://vcgit.hhi.fraunhofer.de/jvet/VVCSoftware_VTM.

[2] F. HHI. Vvenc – open source encoder for vvc, 2020. URL https://github.com/fraunhoferhhi/vvenc.

[3] J. V. E. T. (JVET). Versatile video coding (vvc): Working draft 10 of vvc, 2020. URL https://jvet-experts.org/doc_end_user/current_document.php?id=10541.

[4] J. V. E. T. (JVET). Overview of the versatile video coding (vvc) standard and its applications, 2020. URL https://www.researchgate.net/publication/355032975_Overview_of_the_Versatile_Video_Coding_VVC_Standard_and_its_Applications.

[5] Vicuesoft. Coding tree unit. URL https://vicuesoft.com/glossary/term/ctu_cu_pu_lcu.

[5] [3] [1] [2] [4]

A | Appendix

```
architecture rtl of example is
  signal i_ready_1      : std_logic;
  signal i_ready_2      : std_logic;
  signal i_valid_1_reg  : std_logic;
  signal i_valid_2_reg  : std_logic;
  signal i_data_1_reg   : std_logic_vector(7 downto 0);
  signal i_data_2_reg   : std_logic_vector(7 downto 0);
begin
  process(rstn, clk)
  begin
    if rstn = '0' then
      i_valid_1_reg <= '0';
      i_valid_2_reg <= '0';
      i_data_1_reg  <= (others => '0');
      i_data_2_reg  <= (others => '0');
    elsif rising_edge(clk) then
      if i_ready_1 = '1' then
        i_valid_1_reg <= valid_in;
        if valid_in = '1' then
          i_data_1_reg <= func1(data_in);
        end if;
      end if;
      if i_ready_2 = '1' then
        i_valid_2_reg <= i_valid_1_reg;
        if i_valid_1_reg = '1' then
          i_data_2_reg <= func2(i_data_1_reg);
        end if;
      end if;
    end if;
  end process;
  valid_out <= i_valid_2_reg;
  data_out  <= i_data_2_reg;
end architecture;
```

Figure A.1: Basic RTL design script performing 2 operations over 2 cycles.

```
Startpoint: rec_slicer_wrapper/rec_slicer/i_pu_pos_x_reg_2_
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: neigh_addr[0]
          (output port clocked by clk)
Path Group: REGOUT
Path Type: max

Point                                                                                    Incr       Path
-------------------------------------------------------------------------------------------------------------
clock clk (rise edge)                                                                    0.00       0.00
clock network delay (ideal)                                                              0.00       0.00
rec_slicer_wrapper/rec_slicer/i_pu_pos_x_reg_2_/CK (DFFRPQA_X2N_AH240TL_C8)              0.00       0.00 r
rec_slicer_wrapper/rec_slicer/i_pu_pos_x_reg_2_/Q (DFFRPQA_X2N_AH240TL_C8)               0.05       0.05 r
rec_slicer_wrapper/rec_slicer/U1215/Y (XOR2_X1TN_AH240TL_C8)                            0.02 *     0.07 r
rec_slicer_wrapper/rec_slicer/U24/Y (NAND2_X2R_AH240TL_C8)                              0.02 *     0.09 f
rec_slicer_wrapper/rec_slicer/U1030/Y (AOI21_X4N_AH240TL_C8)                            0.02 *     0.11 r
rec_slicer_wrapper/rec_slicer/U89/Y (OAI21_X3N_AH240TL_C8)                              0.01 *     0.12 f
rec_slicer_wrapper/rec_slicer/U1078/Y (XOR2_X2TN_AH240TL_C8)                            0.02 *     0.14 f
rec_slicer_wrapper/rec_slicer/U99/Y (NAND2_X4R_AH240TL_C8)                              0.01 *     0.15 r
rec_slicer_wrapper/rec_slicer/U138/Y (NAND2_X4R_AH240TL_C8)                             0.02 *     0.17 f
rec_slicer_wrapper/rec_slicer/U1043/Y (NAND2_X2R_AH240TL_C8)                            0.01 *     0.18 r
rec_slicer_wrapper/rec_slicer/U1034/Y (OAI31_X2N_AH240TL_C8)                            0.01 *     0.20 f
rec_slicer_wrapper/rec_slicer/U1065/Y (NOR2_X2F_AH240TL_C8)                             0.01 *     0.21 r
rec_slicer_wrapper/rec_slicer/U231/Y (OAI31_X2N_AH240TL_C8)                             0.01 *     0.22 f
rec_slicer_wrapper/rec_slicer/neigh_addr_0_ (alg_rec_slicer_0_alg_alf_top)               0.00       0.22 f
rec_slicer_wrapper/neigh_addr_0_ (alg_rec_slicer_wrapper_0_alg_alf_top)                  0.00       0.22 f
neigh_addr[0] (out)                                                                      0.00 *     0.22 f
data arrival time                                                                                   0.22

clock clk (rise edge)                                                                    0.71       0.71
clock network delay (ideal)                                                              0.00       0.71
clock uncertainty                                                                       -0.09       0.63
output external delay                                                                   -0.50       0.13
data required time                                                                                  0.13
-------------------------------------------------------------------------------------------------------------
data required time                                                                                  0.13
data arrival time                                                                                  -0.22
-------------------------------------------------------------------------------------------------------------
slack (VIOLATED)                                                                                   -0.09
```

Figure A.2: Summary of a time-dependent critical path produced by the synthesis software.

List of Figures

List of Tables

Acknowledgements

My first thanks go to MANIGLIER Stéphane, my tutor, who enabled me to join the hardware design team and complete my internship in the best possible conditions.

I would also like to thank CHOLLET Erik, who acted as my second internship tutor. He was always on hand to advise me and point me in the right direction, enabling me to make the most of this end-of-studies project experience.

My final thanks go to all the Allegro DVT staff, who have been so supportive of my integration into the company.