Technical University of Crete
Electronic and Computer Engineering Department
Microprocessor & Hardware Laboratory

Diploma Thesis

Implementation of WebP algorithm on FPGA

Author:

Foivos Anastasopoulos

Committee:

Supervisor: Associate Professor
Ioannis Papaefstathiou

Professor
Dionisios Pnevmatikatos

Professor
Michalis Zervakis

Chania, 2014

Abstract

Image processing has become a crucial part of current technology status quo, being used in many fields such as computer vision, computer graphics and other . In modern sciences and technologies, images also gain much broader scopes due to the ever growing importance of scientific visualization . Complex scenarios of data processing in a wide variety of scientific fields indicate the necessity to visualize large data structures in a most effective way.

Considering the massive growth of internet , both as a scientific and industrial field, image processing is critical to the efforts of software developers as great part of information provided in the multimedia based computing industry contains some kind of image form. Think about all the images , videos, etc a person is brought upon in an average internet session every day, and you can estimate the vitality of image processing algorithms in the effort to make web faster and increase our productivity.

In this thesis WebP was studied , a new image format that provides lossless and lossy compression for images on the web , aiming to increase its performance. Focusing on the more frequently used lossy compression that VP8 encoder facilitates, the algorithm's hotspots were analyzed and the critical functions were implemented in hardware using Xilinx's Vivado HLS(High Level Synthesis) . This tool's convenience of use compared to manual hardware design made it possible to quickly implement various architectures and designs for the hardware accelerated modules and then compare them to choose the most optimal solution. Finally, design space exploration was performed to evaluate the resources used and assess the system's performance.

List of Figures

List of Tables

Contents

Acknowledgements

I would like to express my appreciation to my supervisor professor Mr. Ioannis Papaeustathiou for offering me the chance to be involved in this thesis intriguing subjects of study , as well as for his cooperation and advice until the completion of the thesis.

Also , I would like to thank the rest of the comitee, professor Mr. Dionusios Pnevmatikatos and Mr. Michail Zervakis for their time and effort to read the thesis and for taking part in the examination presentation.

Finally, special thanks to Post Doc Research Assistant Mr. Antonis Nikitakis for his guidance and assist through all the time needed to fulfill this thesis tasks.

Chapter 1

Introduction

In this chapter a brief introduction in Digital Image Processing and its related applications is attempted , then reconfigurable logic supported by FPGAs is introduced and finally ways to improve performance of Image Processing applications using FPGAs is presented.

1.1 Digital Image Processing

Digital image processing [1] is an expanding and dynamic area that impact our everyday life in various aspects with a large number of applications in fields such as computer graphics, computer vision , medicine, space exploration, surveillance, authentication, automated industry inspection and many more areas.

Advances in digital image processing allow use of increased complexity algorithms, and therefore, can offer both more sophisticated performance at simple tasks, and the implementation of methods which would be unlikely to be created by analog means.

Specifically, digital image processing is the only practical technology for:

- Classification
- Image Compression
- Feature Extraction
- Pattern Recognition
- Signal Analysis

Some techniques used in digital image processing include:

- Pixelation
- Linear Filtering
- Hidden Markov Models
- Anisotropic Defusion
- Partial Differential Equations
- Neural Networks
- Wavelets

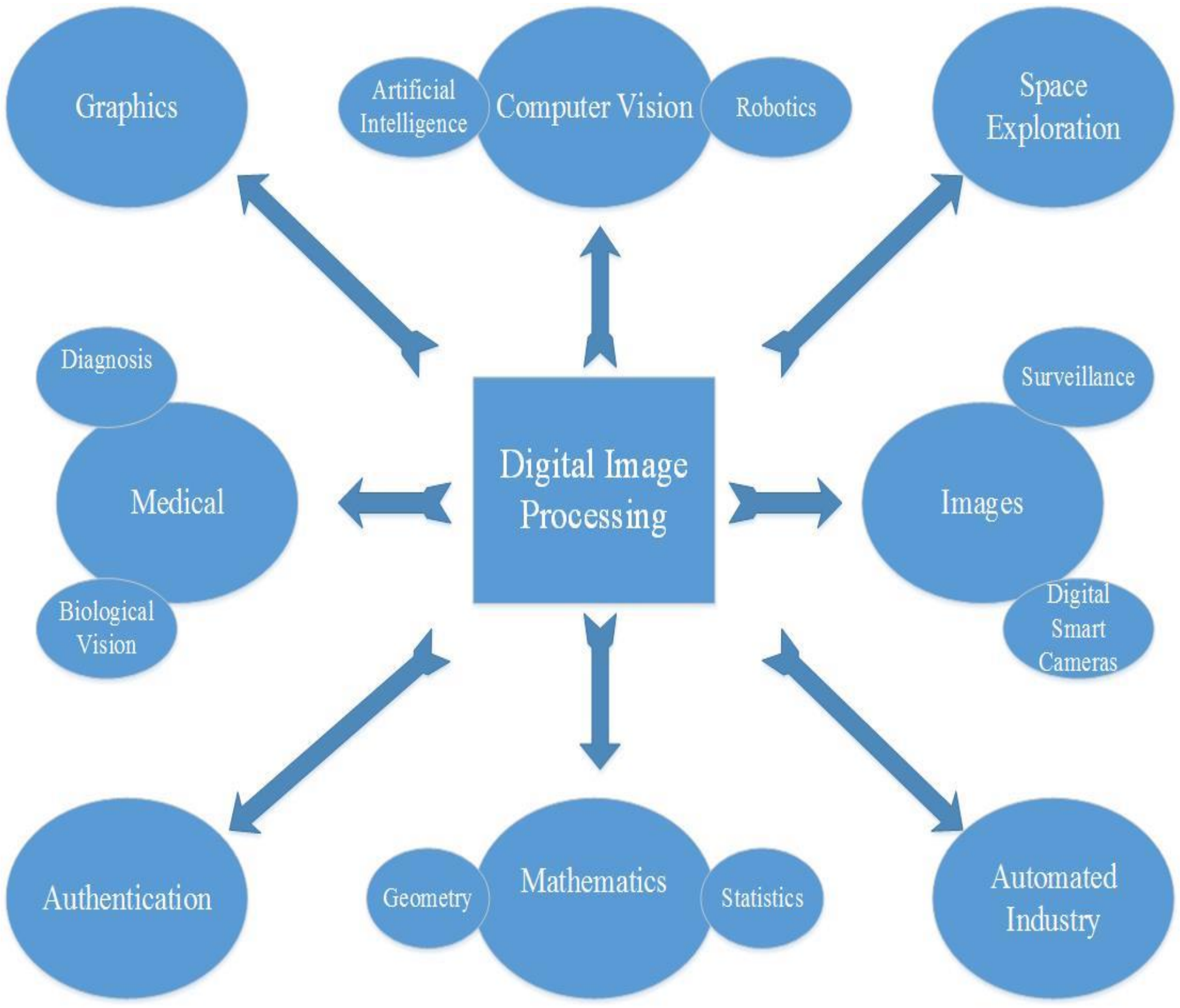


Figure 1.1 Image Processing Related Fields

Digital Images, specifically, play a significant role in present multimedia based computing industry.Being a popular mode of data representation, images have extensively been used in almost all sorts of digital device including the mobile phones, tablet and handheld computers. Although the most popularly used image algorithms are easy to be performed by the powerful processors, still the small devices of less capable processors suffer a lot from encoding or decoding procedures. This is due to some complex computations required by these algorithms . As the production and usage of tablet and handheld computing devices with less capable or low power-consuming processors increased, the necessity of producing more efficient as well as less time consuming tasks for the low capacity processors of these devices has been appeared .

1.2 Field Programmable Gate Arrays (FPGAs)

Field-programmable gate arrays (FPGAs) [2] succeeded in bringing a revolution in computer software and hardware industry by combining the advantages of both worlds. Significant performance ,power , cost and other gains of hardware designs can be maintained and additionally the strict nature of application-specific integrated circuits(ASICs) is surpassed as an FPGA-based system is reconfigurable and not bounded to the chip during manufacturing process. Despite the fact that FPGAs may not reach the highest performance levels achieved by ASICs and the systems designed on FPGAs are found to be larger regarding area , their flexibility and convenience of use can simplify the designing process as well as reduce the manufacturing time and cost .

In Figure 1.2 we can see the internal structure of an FPGA , containing logic blocks that are connected through a general routing structure . Inside logic blocks we can find processing elements as well as flip-flops for implementing combinational and sequential logic and the routing structure enables the connection of the logic elements used in the most preferable way. The facilitated flexibility of such devices allows the implementation of very complicated systems and the usage of special elements such as large memories , multipliers or even complete microprocessors, which are constructed into the silicon, can assist us to implement complete systems in a single device with increased speed and capacity .

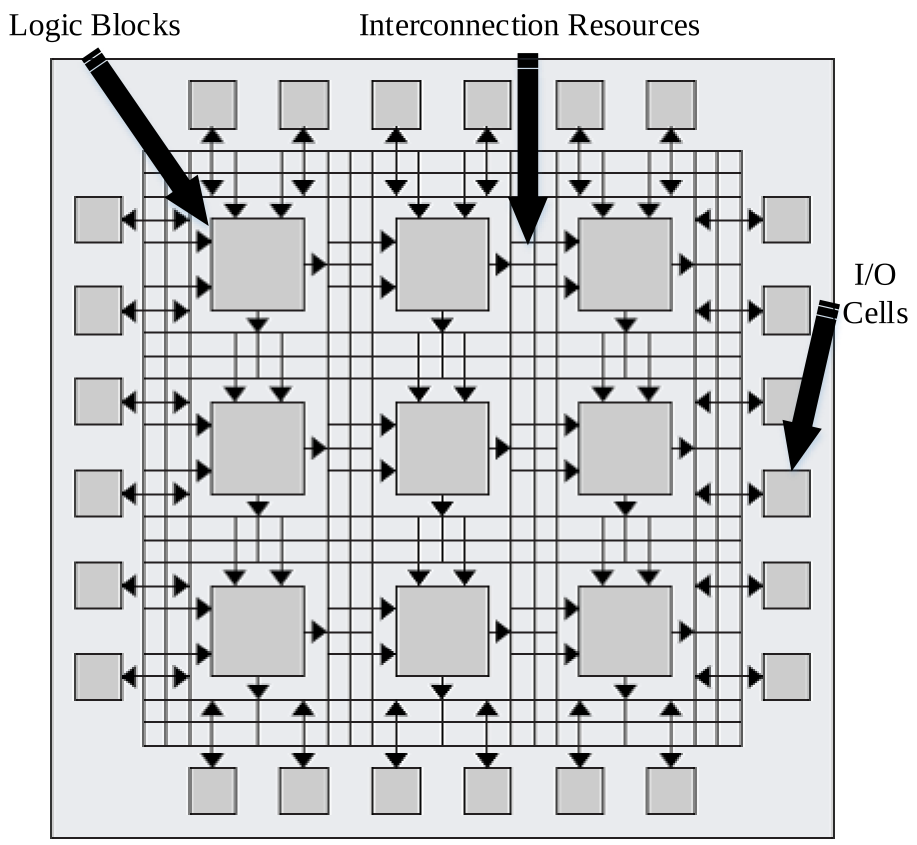


Figure 1.2 Abstract view of basic FPGA architecture, taken and modified from [2]

FPGAs have traditionally been configured by hardware engineers using a Hardware Design Language (HDL). The two principal languages used are Verilog HDL (Verilog) and Very High Speed Integrated Circuits (VHSIC) HDL (VHDL) which allows designers to design at various levels of abstraction.

Reconfigurable computing is the basic concept towards exploiting reconfigurable hardware devices. Even though FPGAs are not specifically optimized for reconfigurable purposes and , consequently, they lack in the architectural advantages that reconfigurable computing specific-devices can offer , reduced (time and financial) cost and power consumption make it up for the lost potential and allowed FPGAs to be widely used for a variety of hardware implemented applications.

The basic structure of an FPGA is composed of the following elements [3]:

• **Look-up table (LUT)**: Basic FPGA's building block for logic operations.
• **Flip-Flop (FF)**: Register element for LUT's result storage .
• **Wires**: Interconnections for elements communication .
• **Input/Output (I/O) pads**: Physically available ports for data transition in and out of the FPGA.

Contemporary FPGA architectures facilitate the above basic elements along with additional and more complicated computational and data storage blocks that can improve the computational density and efficiency of the device. These additional elements can be :

• Embedded memories for distributed data storage
• Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates
• High-speed serial transceivers
• Off-chip memory controllers
• Multiply-accumulate blocks

In Xilinx FPGAs there are also complex computational blocks such as DSP48 block, an arithmetic logic unit (ALU) embedded into the FPGA fabric , containing a series of 3 different connected blocks : add/subtract unit , multiplier , final add/subtract/accumulate engine. This chain of blocks enables a DSP48 block to implement functions of the following form:

$$p = a \times (b + d) + c \qquad (1.1)$$

$$p += a \times (b + d) \qquad (1.2)$$

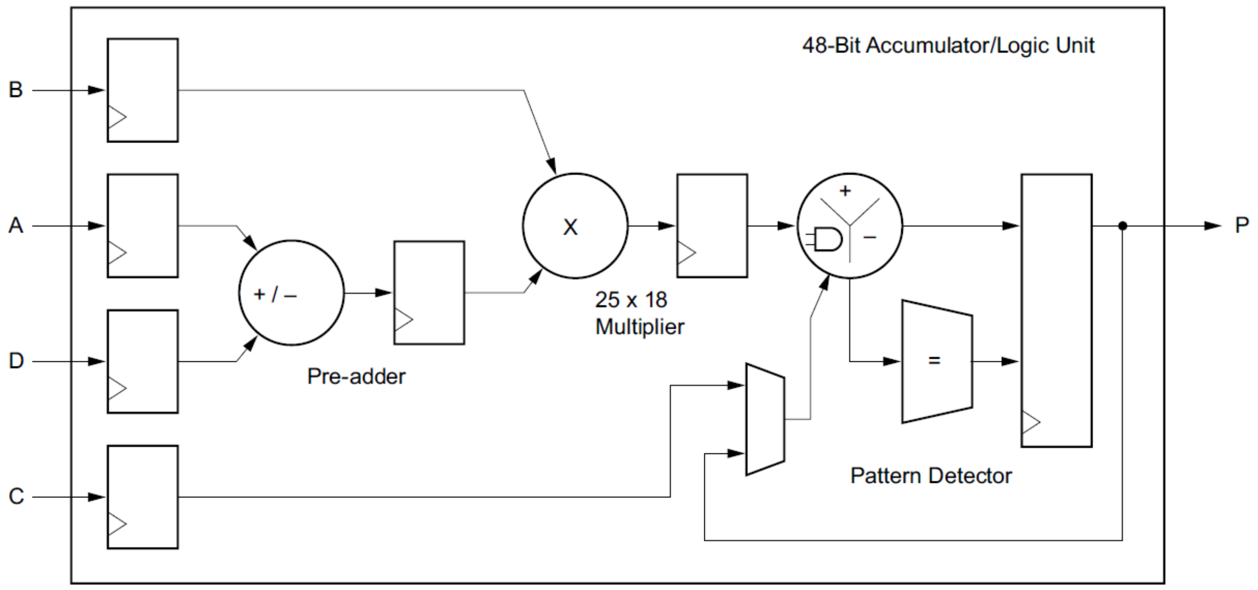


Figure 1.3 Structure of a DSP48 Block [3]

It is clear that modern FPGA technology can provide the designer with a large range of choices regarding computational and memory elements . With proper combination of those elements in the designing process along with communication optimization amongst them , even large systems with heavy computational load can be efficiently mapped into FPGA fabric.

1.3 Digital Image Processing on FPGA's

Digital image processing applications , such as these mentioned before , involve different processes –for instance image enhancement and object detection- which include highly demanding mathematical expressions and operations. Implementing such applications on a general purpose computer can be easier, however additional constraints on memory and other peripheral devices lead to inefficient results considering time.

Implementing digital image processing applications on FPGAs has become an attractive alternative for designers because of the increased flexibility and performance and the relatively low cost . Typical operations for image processing algorithms- for example image differencing, registration, recognition,etc - usually present inherent parallelizable nature, operating concurrently on multiple rows or columns of pixels. With recent advances in FPGA technology fast , parallel processing of image pixels can be achieved by mapping applications of the previously discussed nature into the silicon fabric to take advantage of the provided capabilities. Consequently, custom hardware implementation offers much greater performance and exceptional decrease in algorithm's execution times can be achieved compared to the software version of the same application. [4] [5]

Additionally to the fact that complex computation tasks can be accelerated by exploiting parallelism and pipelining , large software/hardware co-designs can be also implemented on a single device considering the microprocessors that are often embedded in them. The flexibility offered in the co-desing approach [6] nests in the fact that only the time-critical and heavy computational load handling functions are implemented as hardware-accelerated modules , allowing the rest of the software implemented algorithm to run on the processor and call the functions normally with the difference that they are now mapped into the FPGA . This way the time consuming procedure of designing the whole application in hardware description languages is eliminated, however the speed advantages are maintained. Considering the blend of embedded computing capabilities in FPGAs with the cost and performance production benefits that reside in them , FPGAs present a promising perspective for the programmable logic industry.

However, performance and flexibility are not the only benefit. Reduced time-to-market cost, fast prototyping of complex systems and simplified debugging and verification are –amongst other- the primary reasons for the expanding use of custom hardware designs. Considering the variety of modern technologies and scientific efforts that include image processing -related processes, hardware implementation of such algorithms provides great advantages to the developers depending on their motivation . In figure below , the benefits of FPGA prototyping are presented regarding the priorities the developers have claimed that urge them the most to use FPGAs for the prototype of their system. [7]

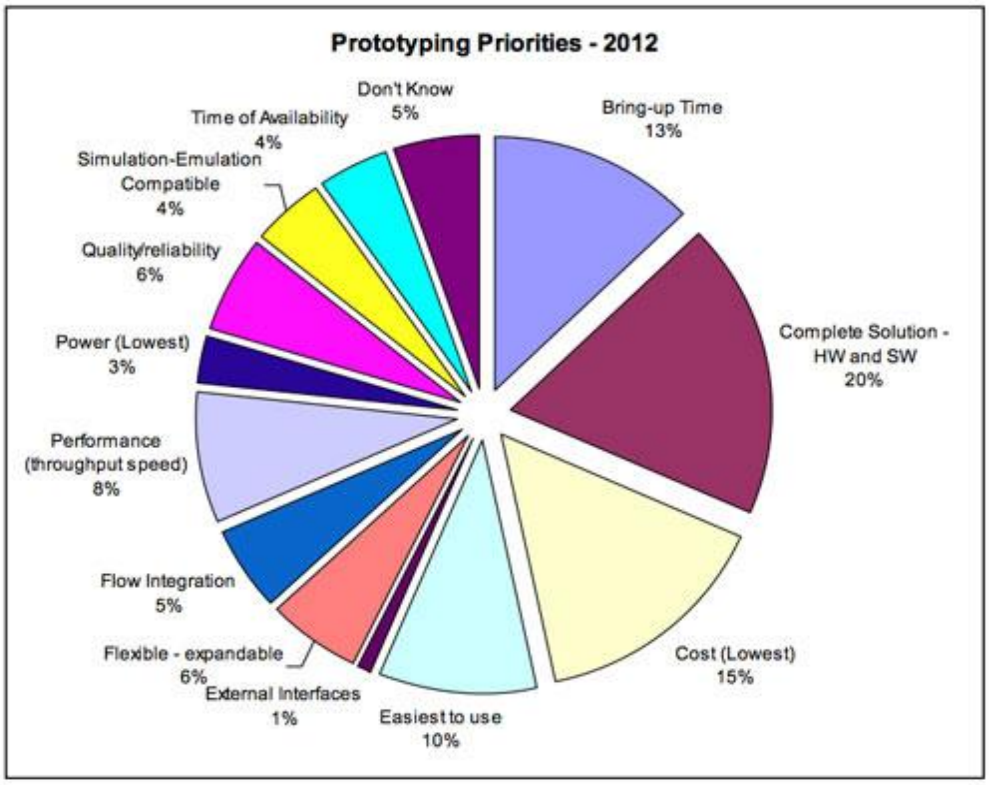


Figure 1.4 Prototyping priorities listed in the 2012 CDT survey [7]

Furthermore, the mobile nature of modern devices designed for a variety of uses , either educational and scientific or entertaining , indicates that power consumption facilitates a great factor towards hardware implementation . Considering that autonomy of those devices has rised as one of the most important parts when a consumer searches the market for a preferable device choice as modern way of living as well as upcoming trends in employment contain the need for remote access to e-mail or the internet generally anytime and anywhere ,we can understand that power consumption reduction has a major contribution to satisfy the more and more emerging necessity of mobile computing. Modern FPGA's can use power reduction techniques in order to facilitate mobile use , allowing designers to maintain a low-power profile across various implemented architectures on the reconfigurable fabric.

Chapter 2

WebP

In this chapter basic image compression techniques used by the most popular image formats are briefly preluded . Subsequently, an introduction to WebP image format is made, describing the concept behind the developing of this new format as well as the main differentiation points between WebP and older formats (for example JPEG). Finally, a closer look to the algorithm is presented, briefly analyzing the principals of its functionality.

2.1 Image Compression

Compression is a process that is used to reduce the physical size of an information block. In other words, the files are coded in such a form that their size is smaller than the original size before compression. Coding means a way of representing data when they are stored in a file, memory , etc. Each compression algorithm is designed so that it searches for and uses data compression for a given order in the stored data. This procedure can include the repeated character sequence, the frequency of occurrence of individual characters, the identification of large blocks of the same data and more.

The main parameters to compare the performance of compression algorithms are the compression ratio and the compression or decompression time. The compression ratio is usually expressed as the ratio between the size of the compressed and uncompressed data . The compression time is the time necessary to transform the original information in to the compressed form. The decompression time indicates the reverse process – it is the time needed to extract the compressed file to its original form.[8]

Image compression seperates into 2 main categories : *lossless* and *lossy* compression.[9]

Lossless compression is an error-free compression . The recovered image is numerically the same as the original image, on a pixel-by-pixel basis . Despite this excellent feature , only a small amount of compression is possible and hence , the obtained compression ratio is low . For applications that tolerate no loss in information , lossless compression is the only acceptable method of data reduction . These applications include : archival of medical images, because loss of any information may affect diagnosis ; archival of bussiness documents where omitting information is illegal etc.

Lossy (irreversible) image compression is based on compromising the accuracy of the recovered image in exchange for more compression. The reconstructed image contains distortion, which may or may not be visually apparent . Depending on the application , a significant compression can be achieved if the resulting degradation can be tolerated for that certain application. In contrast to lossless techniques , a very high compression could be accomplished.

Compression methods regarding images usually adminstrate the trade-off between quality and size , trying to save storage size or bandwith but at the same time keep the error rate between decompressed data and the original picture at a reasonable level .

2.2 WebP Overview

WebP is an image format employing both lossy and lossless compression , developed by Google as part of an overall project designed to improve the performance of web pages and make the web faster by reducing image file sizes. Google estimates 65% of current internet traffic is image and photo data, so a significant decrease in the amount of data sent across the web would increase overall speed for all internet users. [10]

It is presented by the developer as a new standard for lossily compressed true-color graphics on the web, producing smaller files of comparable image quality to the other formats used until now. Specifically, Google claims that WebP lossless images are 26% smaller in size compared to PNGs and WebP lossy images are 25%-34% smaller in size compared to JPEG images at equivalent SSIM (Structural SIMilarity) index.

WebP supports lossless transparency (also known as alpha channel) with just 22% additional bytes. Transparency is also supported with lossy compression and typically provides 3x smaller file sizes compared to PNG when lossy compresion is acceptable for the red/green/blue color channels. After the first release of the format , developers published studies trying to prove the precision of their allegations about the advantages of WebP .

2.3 WebP Evaluation

Since the WebP image format is destined to replace existing image formats used across the web , it would be essential to verify that the announced advantages in theory have the expected practical impact . The studies demonstrated below assess WebP's performance in contrast to known formats facilitating both lossy and lossless compression, over quality as well as timing metrics during encoding and decoding process.

2.3.1 Compression Benefits (WebP vs JPEG)

In this case , the study [11] focuses in the additional compression achieved by WebP at the same quality level of JPEG . In particular, WebP images of same quality (as per SSIM index) as the JPEG images are generated and then the file sizes of WebP and JPEG images are compared.

Structure SIMilarity(SSIM) [12] defines the quality degradation as the product of luminance , contrast and structural errors affecting the image structure. The structural error is defined as the residual error in the image after its normalization with respect to luminance and contrast . The general form of the SSIM between signal *x* and *y* is defined as :

$$SSIM(x,y) = [l(x,y)]^{\alpha} \cdot [c(x,y)]^{\beta} \cdot [s(x,y)]^{\gamma} \tag{2.1}$$

where a, b and g are parameters that define there lative importance of the three components. Although its sensibility to relative translations, scaling and rotations of images, the SSIM index is quite simple and it performs well across a wide variety of image and distortion types. It is able to improve on the traditional PSNR by providing results which are more correlated with the image quality as perceived by the Human Visual System.

The following are the list of data sets used in the experiments by the researchers participated in this survey:

1.Lenna: widely used image Lenna (512 x 512 pixels).
2.Kodak: 24 images from the Kodak true color image suite .
3.Tecnick: 100 images from the collection available at Tecnick.com . The 100 original size RGB color images are used.
4.**Image_crawl:** A random sample of PNG images from Google Image Search web crawl database was collected. The majority of PNG images are icons, graphics, charts, scanned documents, etc. However most images in the standard test collections are like photographs, rather than computer generated images. To make this dataset of similar nature to the standard test suites, a face detection algorithm over these PNG images was run and considered only those images (approximately 11,000) for this experiment, that passed this detection test.
5.

	Lenna	*Kodak*	*Tecnik*	*Image_crawl*
WebP: Average File Size (Average SSIM)	26.7 KB (0.864)	46.5 KB (0.932)	139.0 KB (0.939)	9.9 KB (0.930)
JPEG: Average File Size (Average SSIM)	37.0 KB (0.863)	66.0 KB (0.931)	191.0 KB (0.938)	14.4 KB (0.929)
Ratio of WebP to JPEG file size	0.72	0.70	0.73	0.69

Table 2.1 Average file size for WebP /JPEG for the same SSIM index corresponding to JPEG Q=75 [11]

From the table above, we can observe that WebP gives additional 25%-34% compression gains compared to JPEG at equal or slightly better SSIM index.

In the second case, SSIM vs bits per pixel (bpp) plots for WebP and JPEG is analyzed. These plots show the rate-distortion trade off for WebP and JPEG. The source PNG image is taken, compressed to JPEG and WebP using all possible (0-100) quality values. Then for each quality value the SSIM and bpp achieved for JPEG and WebP is ploted. Following figures show such SSIM vs bpp plots for the 3 images chosen from the 3 public data sets used.

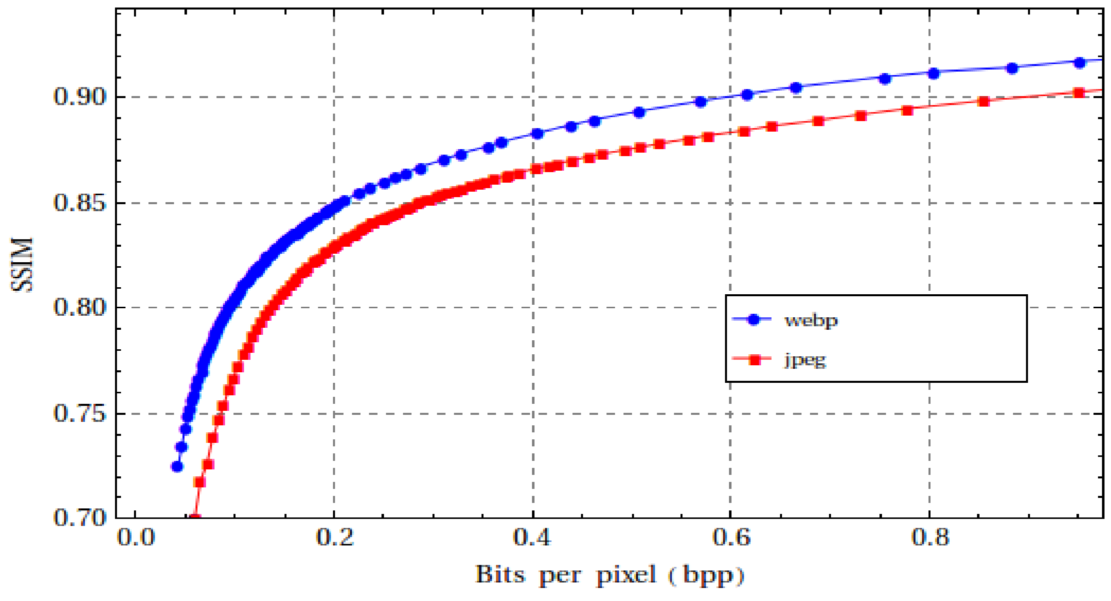


Figure 2. 2 SSIM vs. BPP for Lenna [11]

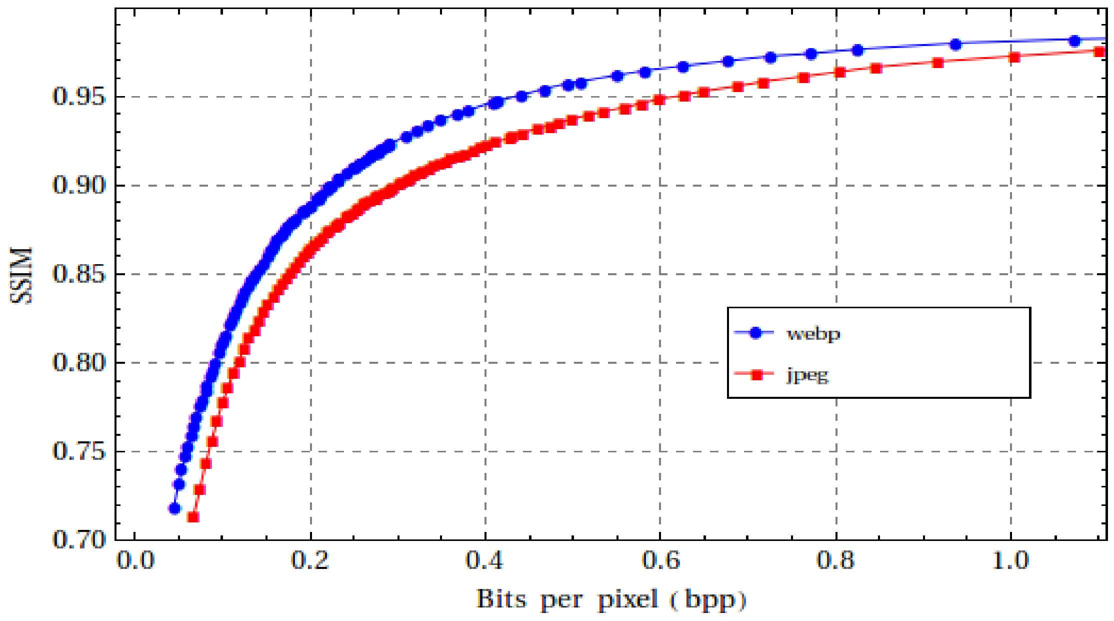


Figure 2.1 SSIM vs. BPP for kodim19.png from the Kodak dataset [11]

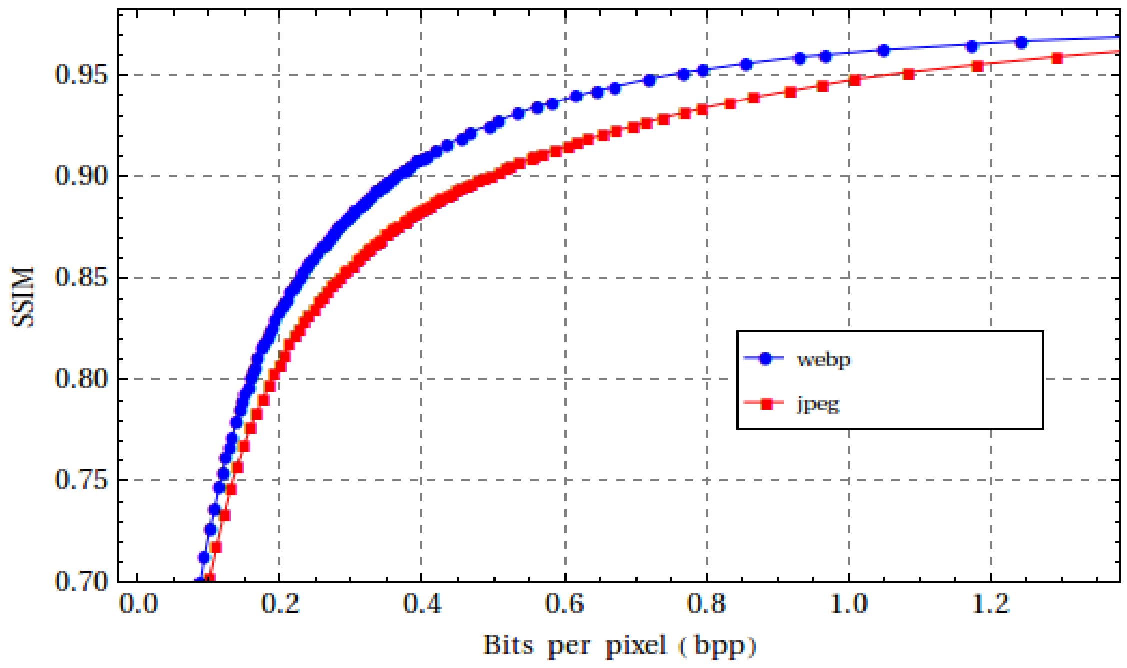


Figure 2.3 SSIM vs. BPP for RGB_OR_1200x1200_061.png from the Tecnick dataset[11]

Overall, from the above plots we can observe that WebP consistently requires less bits per pixel than JPEG to achieve the same SSIM index. These results indicate that WebP can provide significant compression improvements over JPEG .

Another study conducted to assess WebP's performance compared to image compression formats such as JPEG, JPEG 2000 and JPEG XR is set under a subjective quality evaluation perspective , putting a number of participating subjects through a multi-staged testing procedure . WebP's performance is noticed to be competitive towards JPEG 2000 and JPEG XR in the most cases except for limited occasions considering specific bit rate values and images. WebP consistently outperforms JPEG under all test conditions and experiments. More information about this study can be found in [13].

2.3.2 Compression Density and Encoding/Decoding Speed (WebP vs PNG)

In [14] , WebP's lossless and lossy modes performance is evaluated for images that are usually encoded as PNG images after the newly added alpha support for WebP. The study uses 3 image corpora , a photographic image, a graphical image with translucency and finally 1000 randomly collected PNG images with translucency crawled from the internet.

WebP is found to exceed compression density for both libpng(convert) and pngout , maintaining comparable encoding and decoding speeds to PNG as the tables below indicate.

Image Set	Convert-quality 95	pngout	WebP lossless (default settings)	WebP lossless -q 0 m-1	WebP lossy with alpha
photo	12.0	11.9	9.62	10.2	0.71
graphic	1.36	1.12	0.74	0.85	0.56
web	3.69	3.27	2.42	2.70	0.60

Table 2.2 Average bits-per-pixel for the three corpora using the different compression methods. *[14]*

Image Set	Convert-quality 95	pngout	WebP lossless (default settings)	WebP lossless -q 0 m-1	WebP lossy with alpha
photo	0.640 s	16.3 s	3.00 s	0.520 s	3.25 s
graphic	0.260 s	55.9 s	5.27 s	0.040 s	6.00 s
web	0.041 s	2.77 s	0.89 s	0.019 s	0.96 s

Table 2.3 Average encoding time for the compression corpora, and for different compression methods. *[14]*

Image Set	Convert-quality 95	pngout	WebP lossless (default settings)	WebP lossless -q 0 m-1	WebP lossy with alpha
photo	0.130 s	0.130 s	0.060 s	0.060 s	0.010 s
graphic	0.120 s	0.120 s	0.010 s	0.010 s	0.010 s
web	0.038 s	0.040 s	0.006 s	0.006 s	0.005 s

Table 2.4Average decoding time for the three corpora for image files that are compressed with different methods and settings. *[14]*

The survey reaches the conclusion that WebP is a simpler and more efficient replacement format for PNG images , especially with the lossy compression with alpha support that can contribute to speeding up image heavy websites.

2.4 WebP Algorithm Specifics

As declared in [15] ,WebP is an image format that uses either (i) the VP8 key frame encoding to compress image data in a lossy way, or (ii) the WebP lossless encoding. These encoding schemes should make it more efficient than currently used formats. It is optimized for fast image transfer over the network (e.g., for websites). The WebP format has feature equivalency (color profile, metadata, animation etc) with other formats as well.

The WebP container (i.e., RIFF container for WebP) allows feature support over and above the basic use case of WebP (i.e., a file containing a single image encoded as a VP8 key frame). The WebP container provides additional support for:

•**Lossless compression.** An image can be compressed without loss, using the WebP Lossless Format.
•**Metadata.** An image may have metadata stored in EXIF or XMP formats.
•**Transparency.** An image may have transparency, i.e., an alpha channel.
•**Color Profile.** An image may have an embedded ICC profile as described by the International Color Consortium.
•**Animation.** An image may have multiple frames with pauses between them, making it an animation.

Due to better compression of images ,preserving though same quality levels, and support for all these features, it can be an excellent replacement for most images: PNG, JPEG or GIF that most usually focus on either lossy or lossless compression techniques , whereas WebP tries to handle both . In this thesis we are focusing on the lossy part of WebP's algorithm as it is used in a wide range of applications and offers greater compression gains compared to lossless compression. On top of that, most images used on the web are compressed lossily as the exact reconstruction of the original image is not critical for the desired visual representation of the original picture . Also the majority of these images demonstrate low resolution and as a result small bitrate and size and WebP's compression techniques offer certain benefits over other widely used formats considering such low resolution images as presented in [16].

2.4.1 VP8 Encoder

WebP's lossy compression [17] uses the same methodology as VP8 for predicting (video) frames. *Figure 2.7* provides an overview for the VP8 encoding process [18].   VP8 is based on block prediction and as any block-based codec VP8 divides the frame into smaller segments called macroblocks. Within each macroblock, the encoder can predict redundant motion and color information based on previously processed blocks. The image frame is 'key' in the sense that it only uses the pixels already decoded in the immediate spatial neighborhood of each of the macroblocks, and tries to fill the unknown part of them. This is called predictive coding.

The algorithm adjusts the predicted blocks (as wells as synthesize the unpredicted blocks) using a discrete cosine transform (DCT). In one special case, though, VP8 uses a "Walsh-Hadamard"

(WHT) transform instead of a DCT.

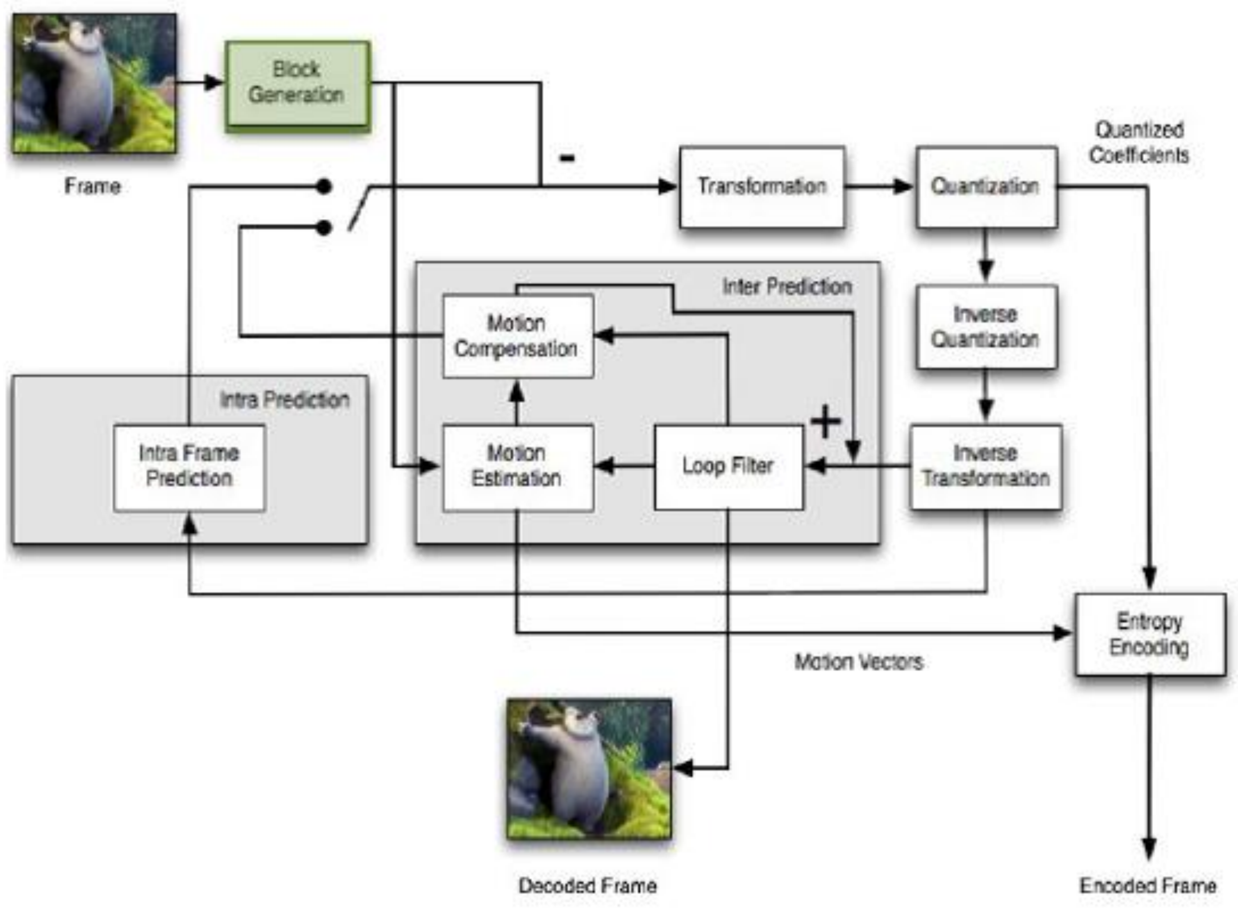


***Figure 2.4 Overview of VP8 Encoding Process** [18]*

WebP algorithm as any similar compression system [19], reduce data rate by exploiting the temporal and spatial coherence of most video signals. The frequency segregation provided by DCT and WHT facilitate the exploitation of both spatial coherence in the original signal and the tolerance of the human visual system to moderate losses of fidelity in the reconstructed signal. VP8 augments these basic concepts with, amongst other, sophisticated use of contextual probabilities. The result is a significant reduction in data rate at a given quality.

Unlike some similar schemes (MPEG formats), VP8 specifies exact values for the reconstructed pixels. Specifically, the specification for the DCT and WHT portions for the reconstruction does not allow for any "drift" caused by truncation of fractions. The algorithm is specified using fixed-precision integer operations exclusively. This facilitates the verification of the correctness of an encoder/decoder implementation as well as avoiding difficult to predict visual incongruities between such implementations.

VP8 holds exclusively an 8-bit YUV 4:2:0 image formats. Each 8-bit pixel in the two chroma planes (U and V) corresponds positionally to a 2x2 block of 8-bit luma pixels in the Y plane; coordinates of the upper left corner of the Y block are of course exactly twice the coordinates of the corresponding chroma pixels.

As usually, pixels are simply a large array of bytes stored in rows from top to bottom , each row being stored from left to right. This raster-scan order is reflected in the layout of the compressed data as well.

Internally, VP8 decomposes each frame into an array of macroblocks , square arrays of pixels whose Y dimensions are 16x16 and U and V dimensions are 8x8. Macroblock-level data in a compressed frame occurs and must be processed in a raster order similar to that of pixels comprising the frame.

Macroblocks are further decomposed into 4x4 subblocks . There are 16 Y sublocks, 4 U sublocks and 4 V sublocks in every macroblock. Again, sublock-level data occurs and are processed in raster order within the containing macroblock.

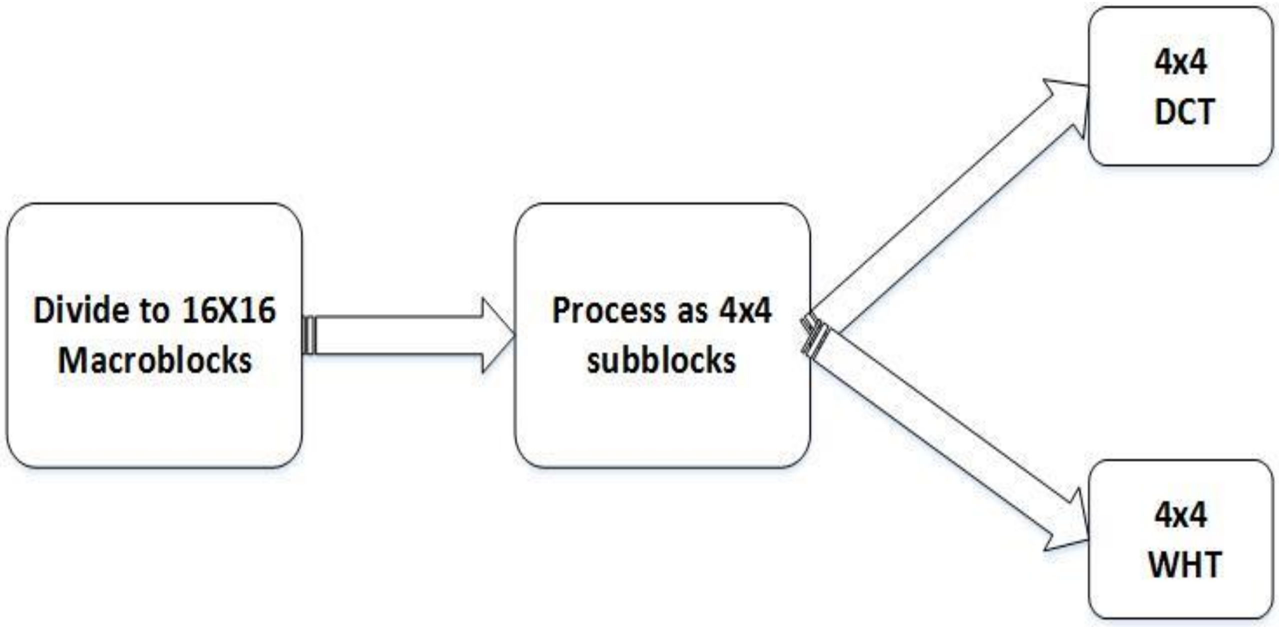


Figure 2.5 VP8 Macroblock Coding

Pixels are always treated, at a minimum, at the level of subblocks, which could be parallelised as the “atoms” of the VP8 algorithm. Particularly, the 2x2 chroma blocks corresponding to 4x4 Y subblocks are never treated explicitly in the data format or in the algorithm specification. DCT and WHT always operate at a 4x4 resolution.

The redundant data can be subtracted from the block, which results in a more efficient compression. We are only left with a small difference called residual, to transmit in a compressed form. After being subject to a mathematically invertible transform (DCT), the residuals typically contain many

zero values, which can be compressed much more effectively. The result is then quantized and entropy-coded. In fact, the quantization step is the only one where bits are discarded in a lossy way. All other steps are invertible and lossless.
The following diagram shows the steps involved in WebP lossy compression. The differentiating features compared to JPEG are circled in red.

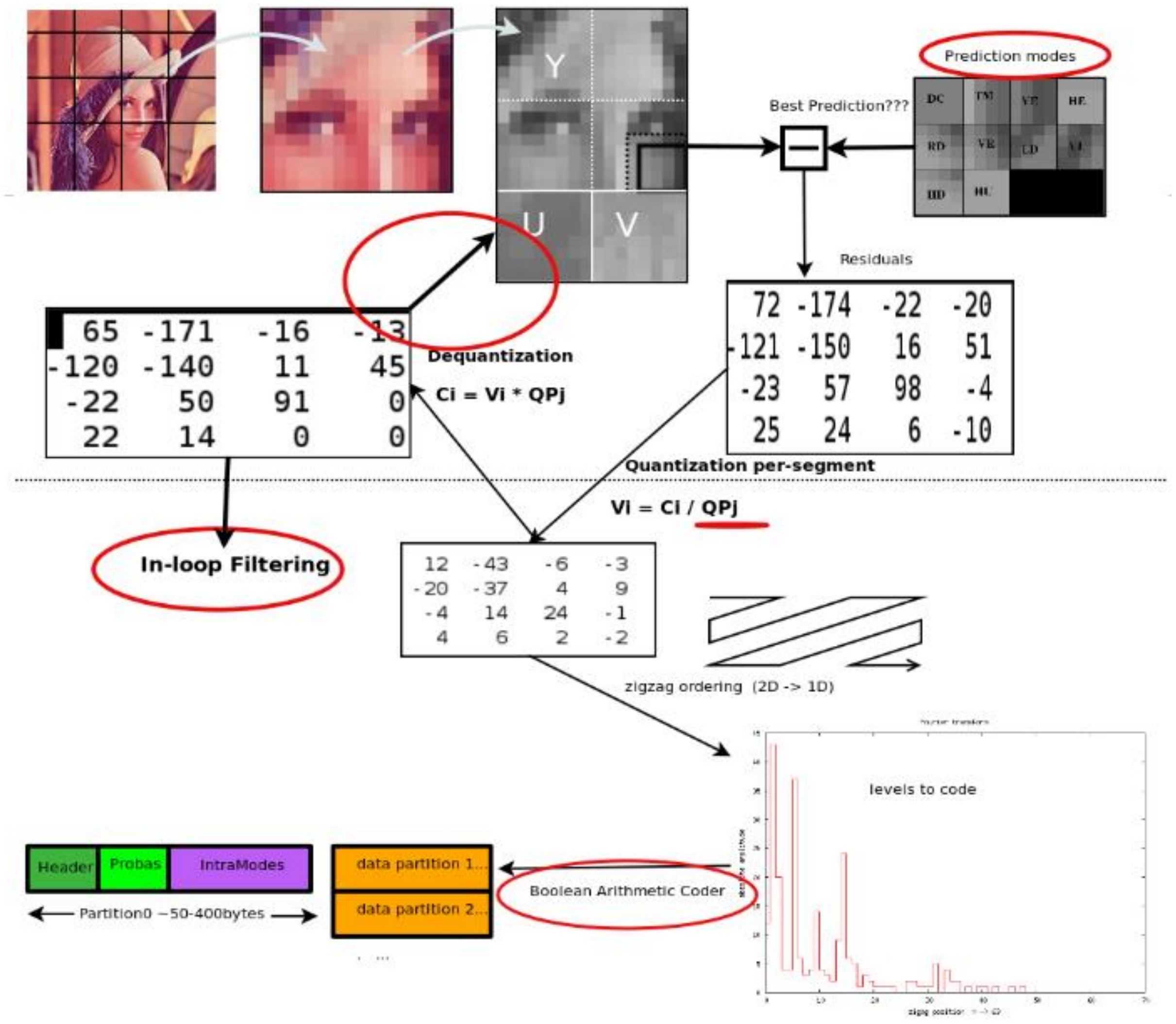


Figure 2.6 WebP's lossy compression basic stages [17]

WebP uses block quantization and distribute bits adaptively across different image segments: fewer bits for low entropy segments and higher bits for higher entropy segments. WebP uses arithmetic entropy encoding achieving better compression compared to the Huffman encoding used in JPEG encoding.

A VP8 encoder [20], [21] uses two classes of prediction:

- *Intra prediction* uses data within a single video frame
- *Inter prediction* uses data from previously encoded frames

The residual signal data is then encoded using other techniques, such as transform coding, as we already discussed. When it comes to image compression ,only intra prediction is used as obviously a single frame is encoded/decoded.

VP8 Intra-prediction Modes

VP8 intra-prediction modes are used with three types of macroblocks:
•4x4 luma
•16x16 luma
•8x8 chroma

Four common intra-prediction modes are shared by these macroblocks:

•**H_PRED** (horizontal prediction). Fills each column of the block with a copy of the left column, L.
•**V_PRED** (vertical prediction). Fills each row of the block with a copy of the above row, A.
•**DC_PRED** (DC prediction). Fills the block with a single value using the average of the pixels in the row above A and the column to the left of L.
•**TM_PRED** (TrueMotion prediction). A mode that gets its name from a compression technique developed by On2 Technologies. In addition to the row A and column L, TM_PRED uses the pixel P above and to the left of the block. Horizontal differences between pixels in A (starting from P) are propagated using the pixels from L to start each row.

The diagram below illustrates the different prediction modes used in WebP lossy compression.

Wu

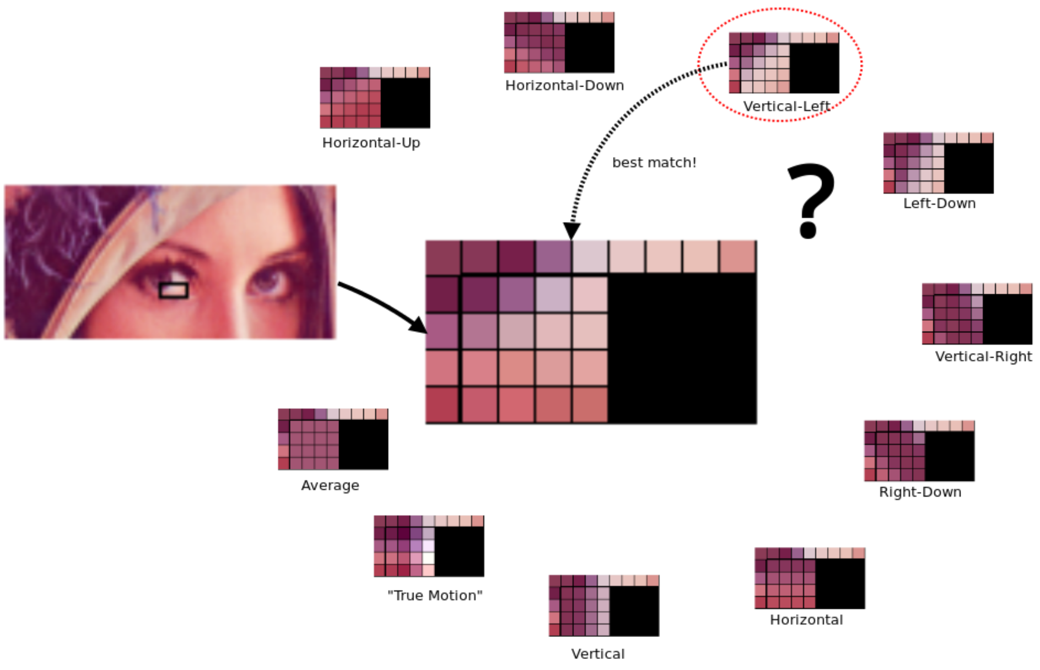


Figure 2.7WebP lossy compression prediction modes [17]

For 4x4 luma blocks, there are six additional intra modes similar to V_PRED and H_PRED, but correspond to predicting pixels in different direction

Chapter 3

Related Work

This chapter contains a synopsis of studies related to this thesis subject. There is a variety of hardware designs implementing versions of DCT/Inverse DCT and quantization process used in popular codecs such as JPEG and H.264 , however it was not possible to find an implementation that handles exactly the same scope with the hardware accelerated modules that will be presented in the following chapters of this thesis.

In [22] , the authors propose two different architectures for the hardware acceleration of DCT and blocks quantization of the H.264 compression standard on FPGA fabric . The first architecture focuses in optimized area results maintaining the natural sequential execution of the algorithms, whereas the second one aims to achieve high throughput and fast processing by facilitating a parallel architecture . The reason for these 2 different solutions is to create appropriate designs for both low power or high performance devices. The achieved FPGA throughput is estimated to be 11M and 32M pixels/sec for DCT and Quant area optimized implementations and 1719M and 1551M pixels/sec for the speed optimized architecture respectively.

Additionally for H.264 hardware implementations, in [23] a novel hardware architecture containing intra-prediction, integer transform, quantization, inverse integer transform, inverse quantization and mode decision processing blocks is proposed for the H.264 macroblock engine. The main improvement in the specified design is a method to reduce cycle overhead for intra16 prediction modes by pre-computing the quantized values of DC coefficients, resulting in reduced latency. The modules are implemented using Verilog Hardware Description language and they run at 54 MHZ using Hynix 0.35 µm Triple Layer Metal library, whereas all types of macroblocks can be processed in 927 clock cycles.

Authors in [24] present an efficient hardware solution for H.264 4x4 forward and inverse transform coding and quantization/rescaling blocks with reduced complexity as the rescaling stages are merged into the quantizer and as a result the number of necessary multiplications for the processing is decreased. The modules are synthesized with TSMC 0.35 µm technology and the implemented encoder can achieve 256 M samples/sec at 32 MHZ.

A high performance architecture for the hardware implementation of simplified 8x8 transfromation and quantization facilitated in H.264 standard is developed in [25]. The concept has to do with pipelined operations in the design to decrease accesses in memory that cost resources as well as time and increase throughput. The system is mapped into XC2V4000 device of Xilinx’s Virtex 2 family and the implemented architecture satisfies the real-time constraints for even high resolution video formats.

Regarding JPEG hardware implementations, a low cost JPEG Encoder hardware module is demonstrated in [26] that processes an image as a stream of 8x8 blocks. The necessary divisions in quantization stage are replaced with a combination of multiplications and shifts with the appropriate usage of quantization tables and DCT step is structured in a way that the usual need of a zigzag unit is eliminated. The JPEG Encoder is implemented on Xilinx Spartan-3 XC3S200 and the reduced complexity leads in minimal usage of FPGA resources, setting the specific proposal ideal for low-cost FPGAs.

Researchers of [27] present a 2D-DCT hardware accelerator design for a FPGA-based SoC using a single 1D-DCT pipeline apparted by 7 stages and special memories , resulting in a 80 clock cycle design running at 107 MHZ that imlements a complete 8x8 2D DCT. JPEG algorithm is found to be significantly accelerated when implemented in HW/SW co design on Microblaze soft core processor and the XC2VP30 board compared to the complete software system running on the same processor.

In next study [28], implementations of DCT and Inverse DCT used in many compression standards –for instance JPEG, MPEG and H.26X- are targeted on Memec Virtex II Pro Development Kit so as to optimize the processing time of the system by implementing a SW/HW co-design based on the embedded processor cooperation with the customized hardware accelerators. The DCT core is presented to compute a 8x8 block in about 0,7 μsec running at 100 MHZ , whereas the 2-D DCT calculation of a 32x32 pixels gray level image can be completed in around 12 μsec.

Considering VP8 Encoder hardware implementations , [19] proposes a cost effective VP8 hardware encoder by reusing H.264 hardware IP already developed in industry. Feature parity between VP8 and H.264 allows the adaptation of most of H.264 hardware implementation in the encoder's architecture, resulting in comparable quality to the reference VP8 Encoder in relatively low HW cost and increased flexibility and effectiveness.

Chapter 4

WebP Profile Analysis

In the following chapter a profile analysis for the WebP algorithm is presented , identifying the most time-consuming functions that will be implemented in hardware along with a description of their functionality. Furthermore , the partitioning of the original program is demonstrated, specifying the software and hardware accelerated stages of the implementation, and the expected speed-up is calculated.

4.1 General Profiling Information

Profiling took place in a *64 bit AMD triple core processor* , running at *3 GHZ* . The operating system was *Ubuntu 12.04 (64 bit)* and version *0.2.1* of WebP was studied , released in August 2012. For the profiling purposes of this thesis *gprof* was used, a performance analysis tool for UNIX applications, and the generated flat profile and call graph was unified in a diagram using *gprof2dot* python script.

To increase the reliability of the profiling we experimented with various pictures of different size and resolution and their profiling results. More specifically, images from 400x400 pixels (Lenna) up to 4096x2034 pixels were converted to WebP format using 3 different quality factors 50,75 and 100 (75 considered the default quality factor). WebP algorithm presents similar behaviour for all images and quality factors with minor variations in the time used by the critical functions. A representative diagram of the algorithm's function calls is the following in *Figure 4.1*, showing the profile analysis of WebP algorithm regarding conversion of Lenna.jpg image to WebP image format with a quality factor of 75 :

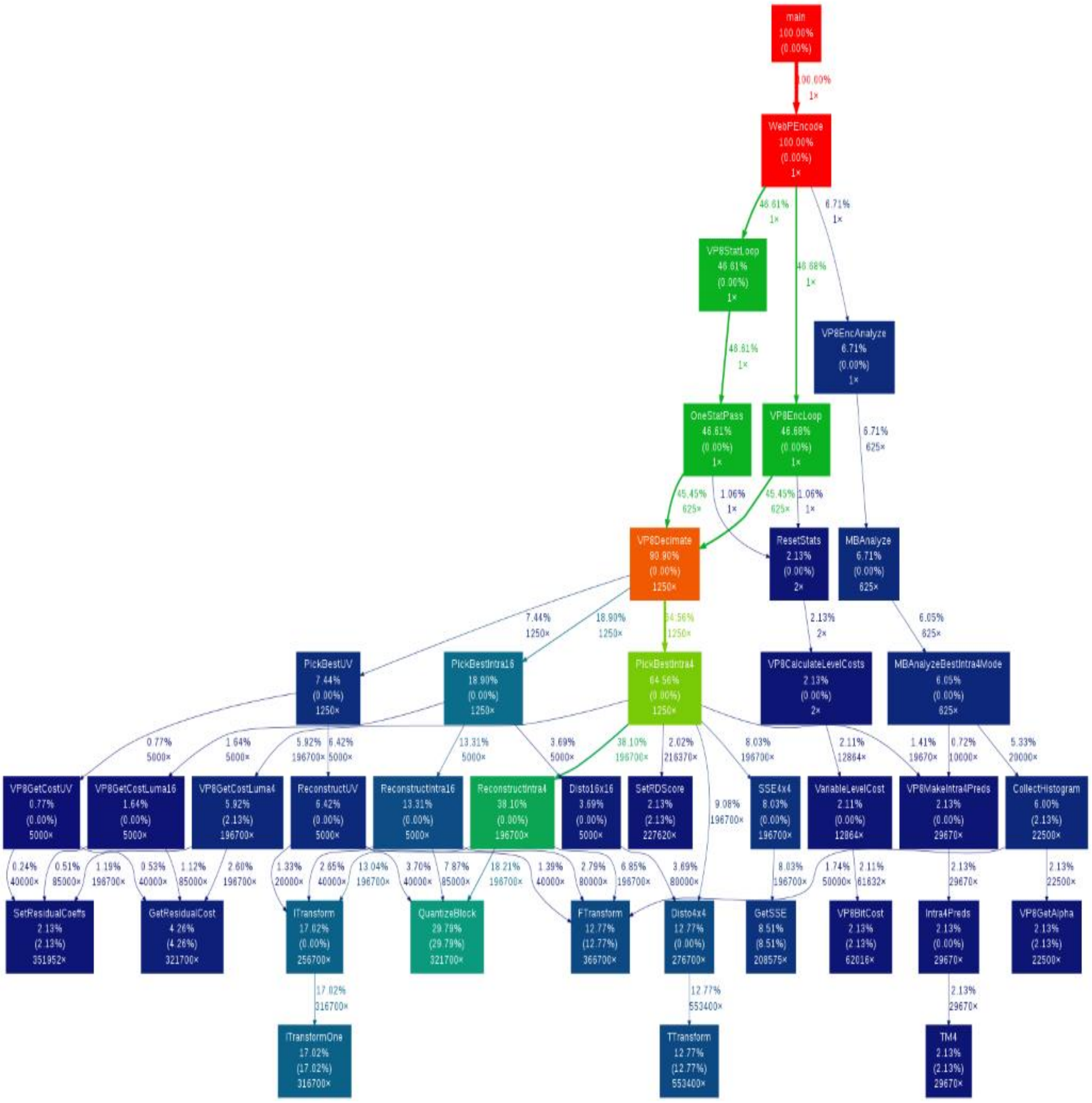

main 100.00% (0.00%)
WebPEncode 100.00% (0.00%) 1×
VP8StatLoop 46.61% (0.00%) 1×
VP8EncAnalyze 6.71% (0.00%) 1×
OneStatPass 46.61% (0.00%) 1×
VP8EncLoop 46.68% (0.00%) 1×
VP8Decimate 90.90% (0.00%) 1250×
ResetStats 2.13% (0.00%) 2×
MBAnalyze 6.71% (0.00%) 625×
PickBestUV 7.44% (0.00%) 1250×
PickBestIntra16 18.90% (0.00%) 1250×
PickBestIntra4 64.56% (0.00%) 1250×
VP8CalculateLevelCosts 2.13% (0.00%) 2×
MBAnalyzeBestIntra4Mode 6.05% (0.00%) 625×
VP8GetCostUV 0.77% (0.00%) 5000×
VP8GetCostLuma16 1.64% (0.00%) 5000×
VP8GetCostLuma4 5.92% (2.13%) 196700×
ReconstructUV 6.42% (0.00%) 5000×
ReconstructIntra16 13.31% (0.00%) 5000×
ReconstructIntra4 38.10% (0.00%) 196700×
Disto16x16 3.69% (0.00%) 5000×
SetRDScore 2.13% (2.13%) 227620×
SSE4x4 8.03% (0.00%) 196700×
VariableLevelCost 2.11% (0.00%) 12864×
VP8MakeIntra4Preds 2.13% (0.00%) 29670×
CollectHistogram 6.00% (2.13%) 22500×
SetResidualCoeffs 2.13% (2.13%) 351952×
GetResidualCost 4.26% (4.26%) 321700×
ITransform 17.02% (0.00%) 256700×
QuantizeBlock 29.79% (29.79%) 321700×
FTransform 12.77% (12.77%) 366700×
Disto4x4 12.77% (0.00%) 276700×
GetSSE 8.51% (8.51%) 208575×
VP8BitCost 2.13% (2.13%) 62016×
Intra4Preds 2.13% (0.00%) 29670×
VP8GetAlpha 2.13% (2.13%) 22500×
ITransformOne 17.02% (17.02%) 316700×
TTransform 12.77% (12.77%) 553400×
TM4 2.13% (2.13%) 29670×

Figure 4.1 WebP function call diagram

4.2 Profiling

4.2.1 Important Functions

Focusing on the time-consuming in the operative level functions , as for functions that actually perform computational tasks and not assign them to the function below in the hierarchy, the following profiling results are collected:

Function	***Execution Time Percentage (%)***
QuantizeBlock	29,79
ITranform	17,02
TTranform	12,77
FTransform	12,77
GetSSE	8,51
GetResidualCost	4,26

Table 4.1 WebP critical functions execution time percentage

Functions of mathematical nature seem to occupy the majority of algorithm's computation weight ,for instance *TTransform, FTransform, ITransform* , as well as quantization through *QuantizeBlock.*

The vast majority of calls to the above functions are performed during the reconstruction of the image's blocks (calls to *QuantizeBlock, ITransform, FTransform)* as well as the texture distortion measurement of the reconstructed pixels compared to the original ones (calls to *TTransform)* . Consequently, ascending a level in the function hierarchy above the time consuming functions and implementing in hardware the below functions can play a significant part in accelerating the algorithm:

- *ReconstructIntra4()/Disto4x4 ()*
- *ReconstructIntra16()/Disto16x16()*
- *ReconstructIntraUV()*

Fortunately, by applying minor modifications in the WebP algorithm *Reconstruct* and *Disto* functions can be connected to operate sequentially and communicate by passing the output of the first as input to the second one .As a result it is more efficient to map the hardware accelerated implementations of them one next to the other ,as it will be further analysed later in the system architecture section.

Additionally, it is possible to implement more that one function call to the above modules , as long as the necessary data are loaded to the board from the processor. As it is already mentioned in the VP8Encoder section of Chapter 2, the above function modules are executed in groups, depending on the prediction modes number of the specific block . There are 10 different prediction modes for

4x4 luma blocks and 4 prediction modes for 16x16 luma blocks as well as chroma blocks. More details about the implemented design will be discussed in following chapters.

Function	***Execution Time Percentage (%)***
ReconstructIntra4	38,10
ReconstructIntra16	13,31
ReconstructUV	6,42
Disto4x4	9.08
Disto16x16	3,69
Total	**70,60**

Table 4.2 Selected for HW acceleration functions execution time percentage

The functions selected for hardware implementation consume around 70,60 % of the total execution time of the algorithm , a rather significant proportion.

In terms of execution time, considering strictly computation time , using functions from the time library the results below are collected. Since *Reconstruct / Disto* functions are destined to be implemented in the same module for *intra4x4/16x16(luma)* reconstruction as they normally operate sequentially in software, the time they consume is calculated cumulatively. The execution time column of the following table regarding all prediction modes contains the time consumed for all modes of the corresponding block type(4x4/16x16 luma , chroma) and the last column displays a quite accurate approach of the total execution time of the specific modules during Lenna image conversion to WebP format in real conditions. The execution time is initially calculated through multiplication of the selected functions execution time for all prediction modes with the total number of calls to the modules.

Function	***Execution Time (μsec) / function call***	***Execution Time (μsec) / All prediction modes***	***Execution Time (sec) / Lenna image***
ReconstructIntra4 /Disto4x4	1,33	13,3	0,276
ReconstructIntra16 /Disto16x16	23	92	0,127
ReconstructUV	7.5	30	0,0405
Total	-	135,3	0,4435

Table 4.3 Functions Software Execution Time

Using the appropriate option available in the WebP conversion options , the program displays the input as well as the encoding time that the conversion demands . In order to convert Lenna , WebP needs 0,025 sec to read the input image and 0,651 sec for encoding. Considering profiling indicated that the above modules consume 70,6 % of total encoding time , the time required is about 0,456 sec , a very close value to the estimated execution time of the above table and a reassuring factor for the accuracy of the calculations.

4.2.2 Critical Functions and Hot Spots

This section extends the brief analysis of VP8 Encoding , as presented in section 2.4.1. The reader can resort to [18] ,[19] ,[20]and [29] to take a more detailed view in the discussed subjects.

Transforms

Discrete Cosine Transform

VP8 applies transform coding to the residue signal after intra prediction, . A standard image frame submitted for encoding is divided in macroblocks , and each macroblock contains a 16x16 block of luma pixels (Y) and 2 8x8 blocks of chroma pixels (U,V). Since transform functions operate strictly on 4x4 level , luma and chroma blocks are further divided into 4x4 blocks, applying a discrete cosine transform to these 4x4 luma and chroma blocks to convert the residue signals into transform coefficients. DCT is an orthogonal transform independent of the input signal that has fast implementations in its two dimensional form and VP8 facilitates a 2-D DCT as the basic transform coding technique of the signals and the corresponding inverse 2-D DCT to inverse transform the quantized residuals. The formal definition of the 4x4 DCT and 4x4 Inverse DCT are given in Equations (4.2),(4.3), however , VP8 facilitates an alternative form that uses multiple passes of the one dimensional version of the 2D-DCT.

$$F(u,v) = \frac{1}{4} CuCv \sum_{i=0}^{3} \sum_{j=0}^{3} f(i,j) \cos[\frac{\pi}{4}(i+\frac{1}{2})u] \cos\left[\frac{\pi}{4}(j+\frac{1}{2})u\right] \qquad (4.1)$$

$$f(i,j) = CuCv \sum_{i=0}^{3} \sum_{j=0}^{3} F(u,v) \cos[\frac{\pi}{4}(i+\frac{1}{2})u] \cos[\frac{\pi}{4}(j+\frac{1}{2})u] \qquad (4.2)$$

Values C_u and C_v are scaling coefficients defined below:

$$Cx = \begin{cases} \frac{1}{\sqrt{2}} \; if \;\; x = 0 \\ 1 \; otherwise \end{cases} \qquad (4.3)$$

Walsh-Hadamant Transform

DCT positions the most significant coefficients in the top left of the matrix, with the first coefficient known as the DC coefficient and the rest 15 as AC coefficients. In order to reduce the redundancy of the DC coefficients in 16x16 luma blocks predicted with 16x16 luma prediction modes, a 4x4 Walsh-Hadamant Transform is applied to the 4x4 block formed by the DC coefficients in each of the 16 4x4 luma blocks that exist in the macroblock. The WHT is defined as following:

$$H_m = \frac{1}{\sqrt{2}}\begin{pmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{pmatrix} \tag{4.4}$$

Where H_m is a 2^m x2^m matrix and H_0=1.

Quantization

Since transforms coding follows a bit-exact pattern, quantization is the only actually lossy step of the encoding algorithm. Quantization stage divides the already transformed block by a quantization matrix, removing high frequency data from the residuals. The quantization matrix is formed depending on the quantization parameter chosen in the encoding process in order to achieve the desired quality levels and it is the same for all macroblocks of a particular image frame. However, VP8 supports a region adaptive quantization scheme , offering the capability to categorize macroblocks of a frame into 4 different segments having a separate quantization parameter.

Reconstruction

ReconstructIntra(4x4/16x16 luma, chroma) functions calculate the difference between source and reference samples based on prediction type and modes , then perform a DCT transform to the coefficients and continues with quantization of the blocks processed . In luma 16x16 blocks , a WHT is also performed to the DC coefficients before quantization and the corresponding Inverse WHT after Finally back- transforms the macroblocks using inverse DCT and gives as output the reconstructed blocks and the quantization levels .

Distortion

Through *Disto4x4 , Disto16x16* the algorithm intends to match the weighted spectral content between original and reconstructed samples after reconstruction of the macroblocks for Intra4x4 or Intra16x16 prediction mode respectively . The input arguments are the source and the reconstructed pixel values and a defined array used for distortion measurement . These functions perform 2 Hadamand transforms returning the weighted summary of the absolute value of transformed coefficients , source and reconstructed , and the output is the absolute value of the subtracted and shifted distortion weight summaries .

4.2.3 Expected Speed-Up

Accelerating the mentioned functions that occupy 70,6 % of the time by implementing them in hardware and leaving the rest of functions in software can give an important speed-up.

We can calculate the expected overall speed-up of the algorithm using Amdahl's Law equation[30]:

$$Speedup_{overall} = \frac{ExecutionTime_{old}}{ExecutionTime_{new}} = \frac{1}{(1-0{,}706)+\frac{f1}{x1}+\frac{f2}{x2}+\frac{f3}{x3}} \tag{4.5}$$

where f1,f2,f3 is the fraction of time that *ReconstructIntra4/Disto4x4* , *ReconstructIntra16/ Disto16x16* and *ReconstructUV* are executed and x1,x2,x3 is the speed-up of the hardware implemented version of them respectively.

4.3 Software/Hardware Partitioning

Considering the profile analysis of WebP and the overview of VP8 Encoding process displayed in *Figure 2.7* , we decided to partition the execution of the algorithm as following :

Software

- Block generation
- Intra Prediction
- Entropy Encoding

Hardware

- Reconstruction
  - DCT Transform
  - Walsh-Hadamant Transform (*luma 16x16 blocks)*
  - Block Quantization
  - Inverse Walsh-Hadamant Transform (*luma 16x16 blocks)*
  - Inverse DCT Transform
- Texture Distortion
  - Hadamant Transform

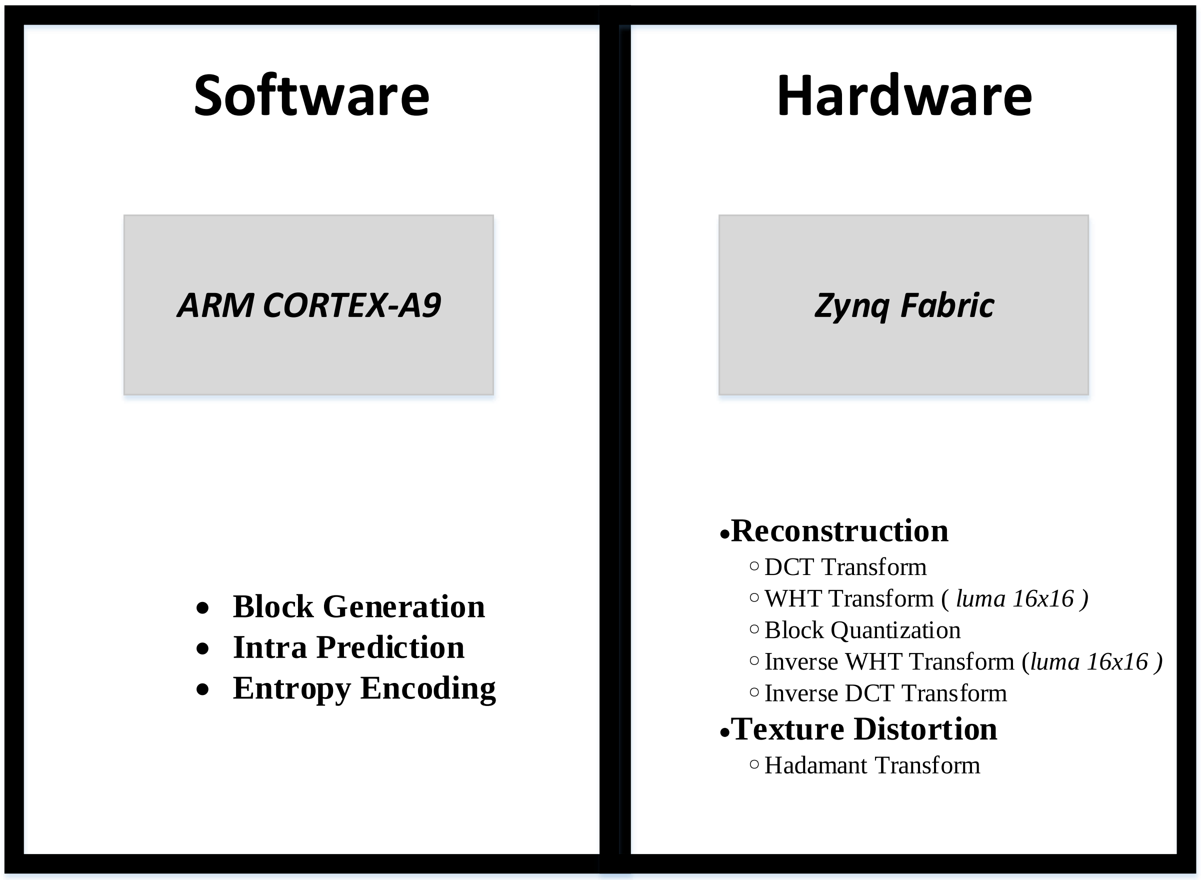
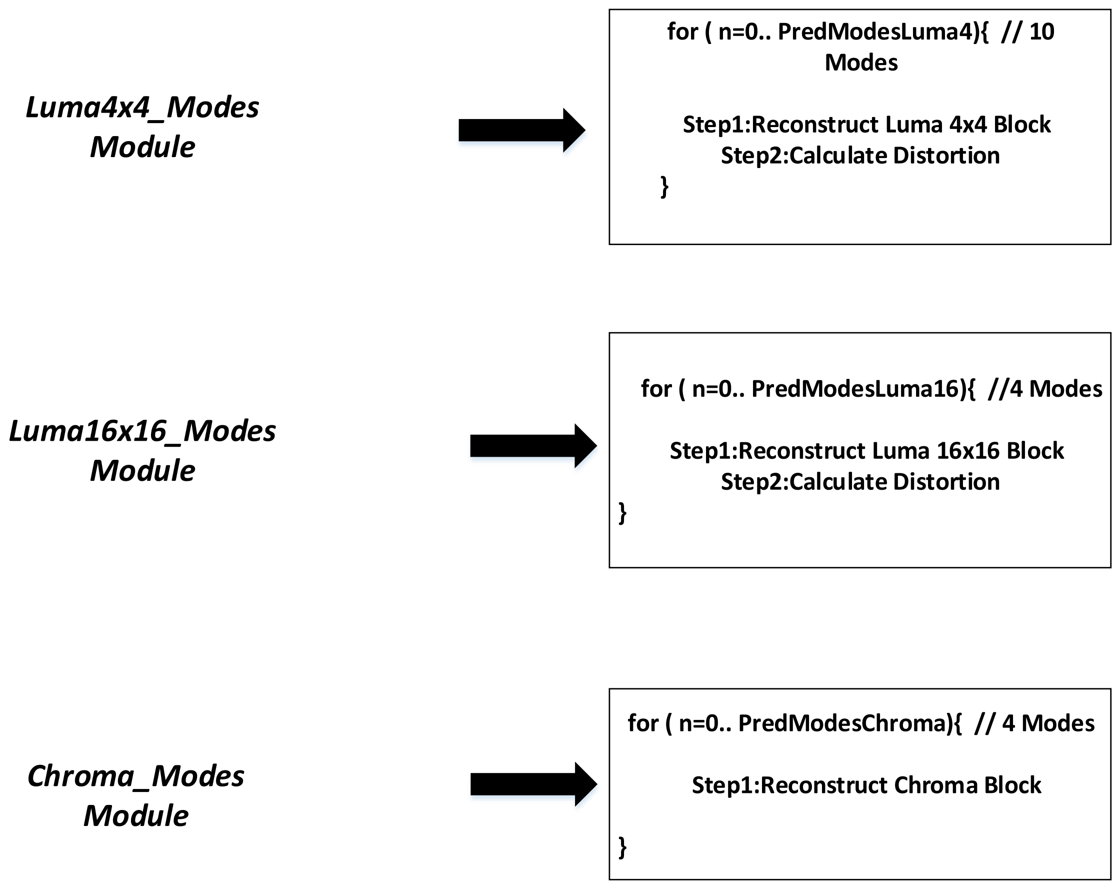


Figure 4.2 Hardware/Software Partitioning

In order to accelerate as much computational load as possible, the selected functions are grouped depending on the type of block they are destined to process. *ReconstructIntra* and *Disto* functions are executed sequentially in software , thus, they can be mapped together in hardware, avoiding unwanted transactions between the CPU and the circuit. Furthermore , the fact that these functions are always executed in loops containing as many iterations as the number of prediction modes of the particular block type and every iteration/function call is independent to the next one gives the opportunity to boost the system's performance by implementing in hardware a parallelized architecture that reconstructs the input block as well as computes the texture distortion between the reconstructed and the original pixels for all the prediction modes available for this block. It must be noted here that at least until the release of the studied 0.2.1 version of WebP , texture distortion measurement through *Disto* function is performed only for the luma blocks.

In other words, software execution computes reconstructed pixels and distortion compared to the source pixels for every prediction mode separately and then calculates the cost to decide for the best mode , whereas in the proposed hardware implementation the circuit pre-computes reconstructed pixels and the corresponding distortion for all the prediction modes and subsequently sends back the results to the processor to continue with the best mode decision and the rest of the processing stages of the algorithm. Schematically, the modules destined for hardware acceleration are the following:



```
for ( n=0.. PredModesLuma4){  // 10 Modes

    Step1:Reconstruct Luma 4x4 Block
    Step2:Calculate Distortion
}
```

```
for ( n=0.. PredModesLuma16){  //4 Modes

    Step1:Reconstruct Luma 16x16 Block
    Step2:Calculate Distortion
}
```

```
for ( n=0.. PredModesChroma){  // 4 Modes

    Step1:Reconstruct Chroma Block

}
```

Figure 4.3 Modules Schematic Description

Chapter 5

High Level Synthesis

This chapter contains an introduction to High Level Synthesis , briefly setting the concept of producing RTL designs from high level synthesis languages and the benefits that occur during this process in contrast to manual design. The primary characteristics of Xilinx's Vivado High Level Synthesis tool , which is used in this thesis, are presented next.

5.1 Background

Each new FPGA generation is implemented on new silicon process technology but maintain low manufacturing and market cost compared to ASICs. Nowadays, implementing complex processing applications such as image and video coding applications on embedded systems is a subject of great challenge. Indeed, increased complexity of the designs for FPGAs demands deep analyzing and advanced requirements regarding timing and area restrictions . Traditional manual hardware design includes hardware description languages [31] (VHDL, Verilog) which require deep hardware knowledge ,despite their high level nature , and provide advanced control over hardware implementation and limited uncertainty about synthesis results.

Nevertheless, manual optimization in the logic level as well as debugging hardware issues can be a long and painful procedure when it comes to complex and large systems. On top of that, results verification and validation require HDL test benches making comparison with the original software model difficult. Another issue would be portability of the hardware designs along the different FPGA types.

These reasons, amonst others, led the industry to seek alternative ways of hardware implementation as the complexity of the designs rose. Necessity for increased productivity and efficiency of the design teams considering the progression in FPGA technology has also contributed to thorough efforts to find a more automatic procedure for the designing process , allowing the designer to control the overview of the implementation and its various constraints but keeping many possibly confusing details away from his scope.

Electronic system-level (ESL) design automation [32] is ought to fulfill this gap , boosting productivity for the hardware industry , where high-level synthesis (HLS) is the primary axis. HLS assists in the automatic synthesis of high-level, untimed or partially timed specifications(C, SystemC) for cycle-accurate RTL specifications and efficient implementation in application-specific integrated circuits (ASICs) or FPGAs. The designs can be further optimized regarding performance ,power and cost of a particular system.

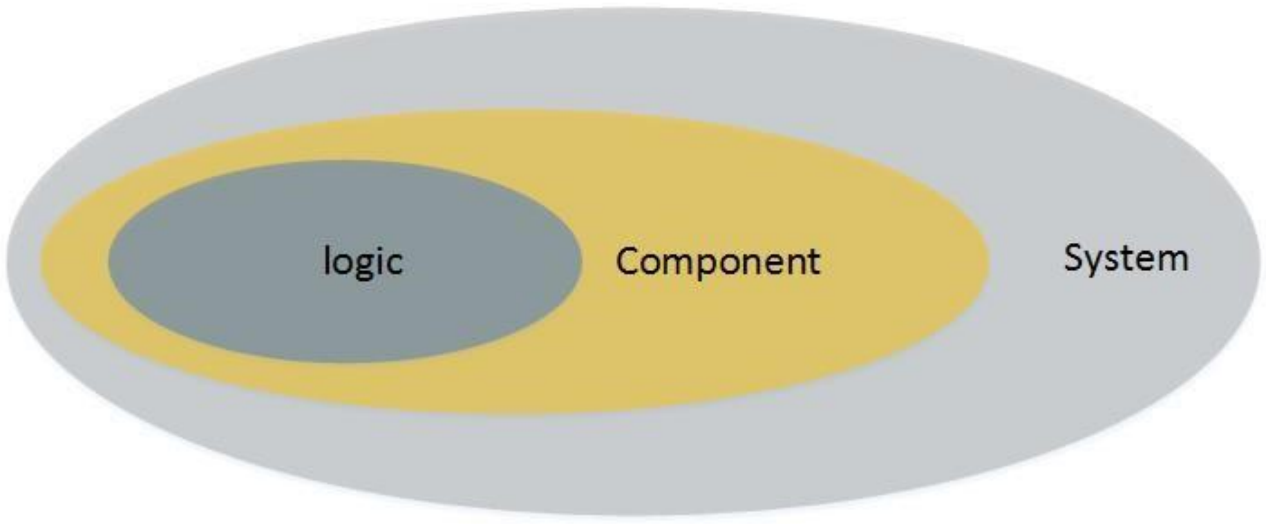


Figure 5.1 Abstraction levels

Complexity of the designs as wells as time-wasting verification procedure create a bottleneck for image processing applications and developing implementations by raising the abstraction level to the ESL assists in sooner time-to-market. System level is the highest abstraction level , where the system is overviewed as a whole, studying the communication amongst components. At the component level, an RTL description is synthesized from a high-level language algorithm, this is the HLS stage. Many tools can perform HLS automatically, taking an C,C++ or SystemC input model and creating a corresponding RTL implementation in harware description language such as VHDL or Verilog.

HLS handles a number of tasks[33] and allows the designer to focus on other aspects of the process. Analysing the source code leads to *resource allocation*, specifying the types and number of operators and memory elements needed. Then, *scheduling* takes place assigning each source code operation to a certain time slot (clock cycle). Finally, software operations and data elements are assigned to operators and memories in *resource binding*. *Interface syntesis* can be issued by high-level synthesis also, creating a suitable interface regarding data and control signals between the generated circuit and its peripherals.

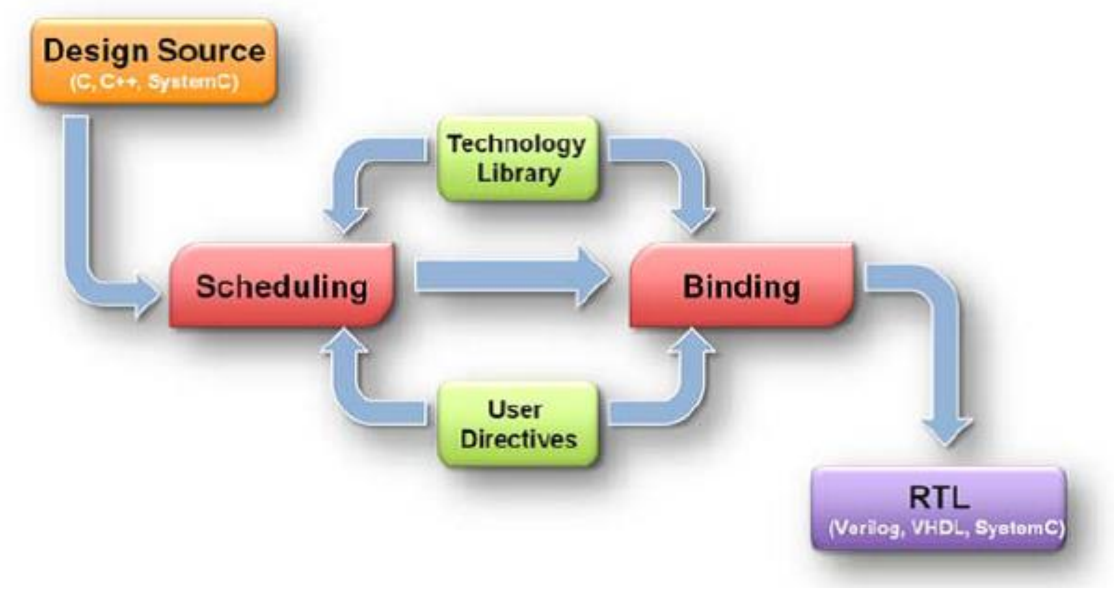


Figure 5.2 Stages of synthesis[35]

Design abstraction is considered to be one of the most effective ways to control complexity and improve design productivity. Modern HLS tools allow the designer to handle the trade-off between performance/power/cost and time , reducing the design time in expense of results or reaching performance relevant with hand-written RTL depending on the goals.

Hardware design flow is benefited in various ways through the use of HLS. First of all, designers are required to write much more smaller amount of code , limiting mistakes and speeding up the process.A study from NEC[34] shows the benefits of HLS in the designing process of an one million gates circuit. At the RTL level about 300K lines of code are required to describe the system whereas the behavioral description would demand only 40K , limiting the simulation and debugging time and improving the performance. Moreover, a design can optimized by HLS tool options and tweaks, creating opportunities for extended design space exploration. Automated HLS flow designers to conceive the system's functionality in high-level languages such as C/C++ and rapidly experiment with different hardware/software boundaries and explore various performance/area/power tradeoffs from the same functional specification .Finally, reusing test data for the verification of the design aside from the validation of the source code though HLS tools generated test benches assist in limiting the verification time, which is sometimes even exceeding the design time of the traditional manual hardware design flow. Especially for FPGA based embedded systems , moving to a higher abstraction level enables controlling increasing design complexity without the need to insert hardware architecture and timing into the algorithm manually and ,therefore , hardware accelerators for embedded software can be implemented with minimal effort. HLS and FPGAs together are a vital step towards faster prototyping and quick time to market.

For software developers specifically, recent FPGA's advances have made reconfigurable computing platforms more attractive for the implementation of many high-performance computing (HPC) applications , such as image and video processing , financial analytics , bioinformatics and scientific computing applications. This is because of the fact that the majority of software application developers consider HDL languages as unacceptable for their purposes and seek a highly automated compilation/sythesis flow from C/C++ to FPGA's. The inherent paralellism of FPGA harware is of vital contribution to accelerate software applications and HLS tools help to improve performance advantages of exploiting the parallelism in FPGA's fabric ease of use and access for those with limited hardware knowledge.

5.2 Vivado HLS (High Level Synthesis)

Software is in the core of applications all across different industry fields such as entertainment, games, communication or medicine. Besides advancements in software-related technologies to enhance algorithmic performance , interest in parallelization and concurrency raised due to the progression in application-specific integrated circuit (ASIC) and FPGA design. FPGA's are generally preferable to designers ,except for very large circuits, because of performance and power consumption gains and reduced cost and complexity.

Initially[3], increased processor clock frequency and use of specialized processors were the primary ways to increase software performance. Progressions in both standard and specialized processors led to replacing clock frequency as a speedup factor by adding more processing cores per chip, introducing program parallelization as a software performace boosting technique. The obstacle now for unifying software and hardware design was the programming model, high level programming languages for software applications and register-transfer level (RTL) descriptions for FPGA's . Vivado High-Level Synthesis compiler facilitates the same functionality for C/C++ programs targeted to FPGA's just as other compilers from high level languages to different processor architectures. The differnce is that HLS compiler exceeds the sequential nature of processor architectures as well as cache and memory space restrictions and exploits the parallel processing capabilities of the FPGA reconfigurable fabric.

In the process of extracting the best ciruit-level implementation of the software program input considering throughput and memory bandwith , the HLS compiler works through the following basic stages:

- ***Scheduling*** different operations of the algorithm respectively with any data dependecies involved , then grouping and overlaping them accordingly.
- ***Pipelining*** the design, increase the level of parallelism in the hardware implementation and improve throughput and performance.
- ***Dataflow,*** another digital design technique, which expresses parallelism in the funtion level based on the communication of inputs and outputs between them.

Vivado HLS compiler facilitates a similar programming environment with standard and specialized processors , sharing technology for the interpretation, analysis and optimization of C/C++ applcations but differentiates by targeting an FPGA s the execution platform. This assists a software engineer to implement computationally intensive software algorithms in an optimized way in terms of throughput, latency and power exceeding possible performance bottleneck due to software programming limitations .The compiler analyzes programs regarding *operations, conditional statements* , *loops* , *functions, arrays* and other *logic structures* and can perform various optimizations upon them based on the user's directives and constraints.

However[36], a number of transformations are required to the original C code , restructuring data as well as organizing them in a more suitable way for the tool to understand in order to remove any unnecessary dependencies or unsupported formats such as dynamic memory allocation and recursive or system calls, that can prevent Vivado from extracting an optimal parallelized implemention. Performing these modifications in a high level environment provides obvious benefits compared to hand-written RTL code as it is easier and errors can be avoided . Furthermore, FPGA's can facilitate arbitrary-precision data types through the tool's supported features, reducing unnecessary resource usage and improving performance. The designer then can direct Vivado HLS to produce RTL HDL code implementing the specified functionality , providing an estimate of clock frequency and resource utilization for this initial design . This way the user can evaluate the implementation and by tweaks in the high-level representation can perform early design exploration in terms of performance and resource usage.

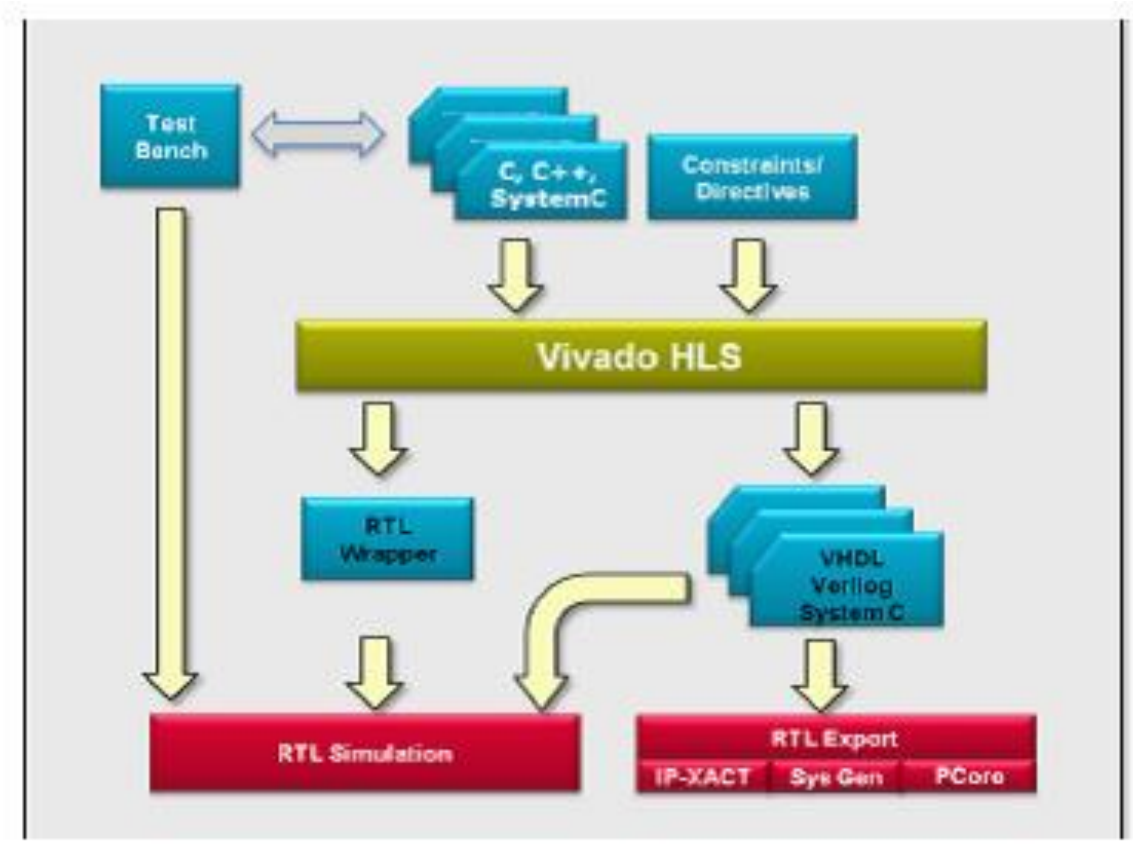


***Figure 5.3 High-Level Synthesis use model** [35]*

High-Level Synthesis applies two differnet types of synthesis to the input software model:

- *Algorithm Synthesis*, synthesizing the function contents and statements into RTL statements over a number of clock cycles.
- *Interface Synthesis,* transforming the function arguments into RTL ports by implementing specific timing protocols to them and this way enabling communication of the design with other designs. There are differnet types of interfaces supported such as *wire, register , bus, FIFO,RAM, one/two way handshakes*.

The design implementation and verification [37] is accelerated significantly by directly synthesizing C/C++/SystemC programs into VHDL or Verilog, after exploring a variety of micro-architectures considering the requirements set by the designer. Simulation of the functionality of the program is perfromed in C, a much faster way compared to hardware description languages simulation. A C-test bench can be included in the input and can be used to verify both the C functionality of the specification and the output RTL, removing the need of RTL test benches. HLS automatically creates the adapters and wrappers to instantiate the RTL implementation into the C test bench and use co-simulation of C and HDL to verify the design.

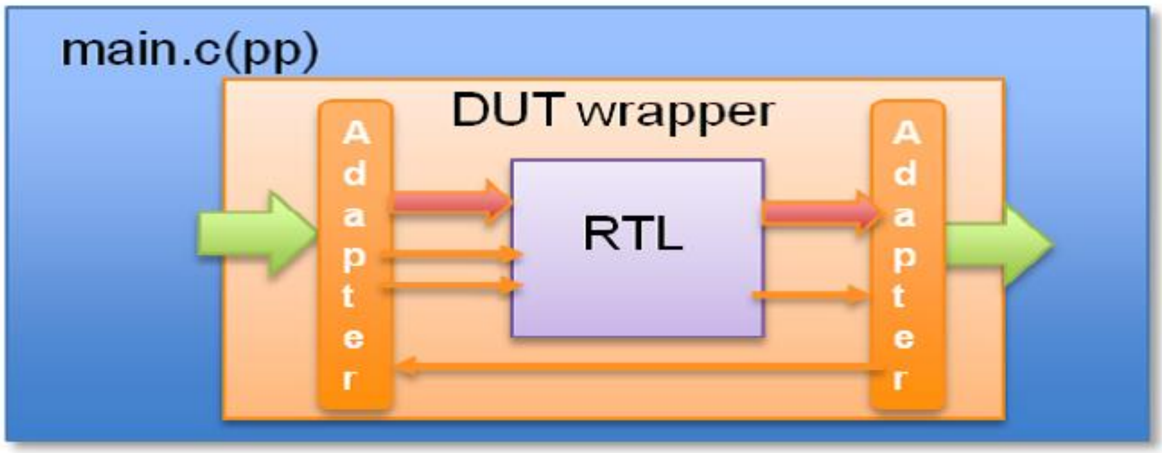


Figure 5.4 Co-simulation design Wrapper overview [35]

Co-simulation guarantees that the hardware design produced preserve the correct functionality of the software algorithm and parallelization directives applied by the designer did not compromise it.

Aside from simulation/verification, the design flow[38] can be categorized in the following parts:

Software Optimization

Transform the C code in a way that will best fit an FPGA platform and benefit by the provided advantages. Examples of common techniques for optimization of software programs destined for a hardware imlementation are:

- *Inlining,* flattening the hierarchy of the component and allowing increased reuse of resources.
- *Memory allocation,* arranging arrays and other elements accordingly to prevent increase in resource and power consumption.

Interface Generation

The tool provides a convenient and direct way for the user to attach interfaces to the implementation arguments or even functions , connecting them appropriately with the rest of the system through standard Xilinx bus architectures and other interfaces like BRAM and FIFO.

Architecture Implementation

Determine the nature of the design architecture depending on the desired area of focus:

- *Parallel ,* if performance is demanded and resource consumption is a minor issue
- *Sequential,* to minimize resource usage
- *Semi-parallel,* combining high throughput of parallel designs with limited usage of resources.
- *Pipelining,* improving throughput and latency between processed inputs that can be applied to the refered architectures.

Optimization directives and settings used in Vivado HLS will be described in a more thorough way during the process of implementing the necessary modules in the following chapter.

High-Level Synthesis uses clock uncertainty to support a user defined timing margin. The timing of operations in the design is estimated , but the final component placement and net routing is unknown and so are the exact delays. The tool will use the usable clock period to schedule the design's operations which can be calculated by subtracting the clock uncertainty from the clock period defined in the implementation settings. By default the clock uncertainty is 12,5% but can be changed by the user . More details about different architectures generated by changing the clock period will be discussed in the implementations section.

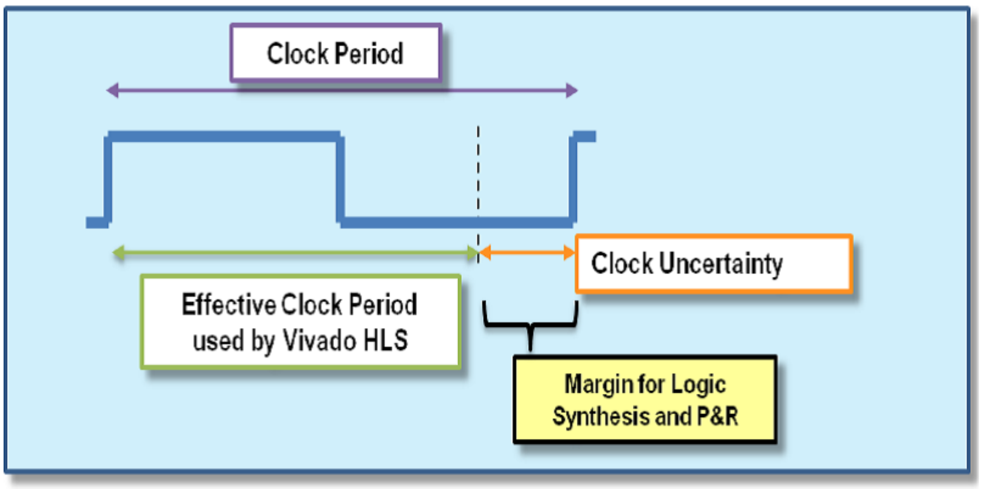

Clock Period
Clock Uncertainty
Effective Clock Period used by Vivado HLS
Margin for Logic Synthesis and P&R

Figure 5.5 Usable Clock Period and Clock Uncertainty [35]

Chapter 6

Hardware Design and Optimizations

In this chapter the implemented modules are described in details regarding targeted board ,clock timing and optimization directives. Specifically, the designing procedure in High Level Synthesis environment is overviewed ,from any necessary source code transformations and C code validation to synthesizing the design and apply various optimizations to explore new architectures. Finally, the optimal solution is verified using the featured C/RTL Co-Simulation and device utilization of the implemented modules is presented.

6.1 Source Code Modifications and C input validation

When it comes to the critical functions modules , transformed functions were validated with the appropriate C test benches through C Simulation , then synthesized to RTL design as well as optimized in terms of timing, latency and area and finally the hardware implementation of each module was verified in the same Vivado HLS environment .

Source code modifications

High-Level Synthesis tools generally and Vivado HLS specifically still operate with some certain restrictions on what they can accept as input in C/C++ language and tranlate it to RTL design. Complex data structures and/or pointers may cause problems in the process of producing a functional design in hardware if they are not treated cautioutly . Especially array pointers have to be declared of their specific dimensions because the tool has to know in advance for example how many registers or BRAMs would have to commit for this array.

- In functions *ReconstructIntra4/ ReconstructIntra16 / ReconstructUV* the input VP8EncIterator struct process was replaced with all the array and array pointers that this specific struct was containing and would be read or computed inside the body of the function. This helps also regarding the C test bench that comes with the design destined for synthesis as it is much more easier to take data from the software execution of the algorithm to use as input in the test bench for validation and ensure that the same exact functionality is implemented. The algorithm and a look to its operations makes it clear that array pointers can be transformed to arrays with specific dimensions that will be easier to handle in a high-level synthesis environment through the use of directives. Some complex array indexing methods also were substituted by a more traditional indexing way as the HLS compiler stumbled upon certain malfunctions .
- In *Disto4x4 , Disto16x16* functions the only necessary modification was the same with above for array pointers in the function input arguments that were changed to finite arrays and the alternate indexing method for array accessing.

Arbitrary-precision data types

Arbitrary-precision data types can be used to limit the size of elements if for example a variable needs 17 bits for storage then we do not have to commit 32 bits for this purpose as Vivado HLS supports arbitrary widths. Fortunately, data types used in our modules are restricted to 8-bit boundaries of C-based native data types(8 ,16,32 bits) so we can avoid the increase in the complexity of the design as well as the amount of time the tool processes the user's directives and constraints to result in a functional RTL design that comes with using arbitrary-precision data types.

Unsupported C language constructs

Possible system calls , dynamic memory allocation functions or recursive functions must be removed from the design code before synthesis . This I because of the fact that the design submitted for synthesis must contain all the required functionality and specify the exact resources needed as during the synthesis process the implying resources are created and released during runtime. As a result , the C input function must include all necessary information to implement the demanded functionality without interventions for tasks executed in the operating system.

Function Inlining

Very small functions are automatically inlined by Vivado HLS , removing the function hierarchy . The benefit that is granted through this feature is that typically there is a cycle overhead to enter and exit functions, so removing function hierarchy may improve latency and throughput as well as area by allowing better sharing of the components that this function consists.

C Simulation

Before synthesis, the C input program has to be checked that it implements the desired functionality. For this *C simulation* the tool needs a test bench file to verify the function destined for hardware implementation. High-Level Synthesis compiles the input and the test bench using a version of gcc, however there is also an alternative compiler . The test bench needs to be self-checking, comparing the results of the generated circuit with the original software execution results that are retrieved and stored in a file. The same C test bench can be used later to verify the hardware design through the C/RTL co-simulation feature of Vivado HLS , simplifying the process and saving us a lot of time used to be devoted to manual verification of the implementation by creating RTL test benches .

6.2 Hardware Modules Implementation

Initially ,Xilinx's Zynq XC7Z045 device residing in the ZC706 evaluation board was targeted for the modules hardware implementation , with the corresponding block diagram given in Figure 6.1 [39] .After some early experimentation the clock period was set at 3 nsec, as despite of the fact that a slight increase was noticed in the design's latency compared to higher clock period designs, the overall execution time of the circuits was significantly improved. In the next chapters alternative implementations with different clock period are studied , so as to evaluate performance of the hardware modules at more than one frequencies and to explore additional parallelization possibilities , taking advantage of the architecture exploration capabilities Vivado HLS provides.

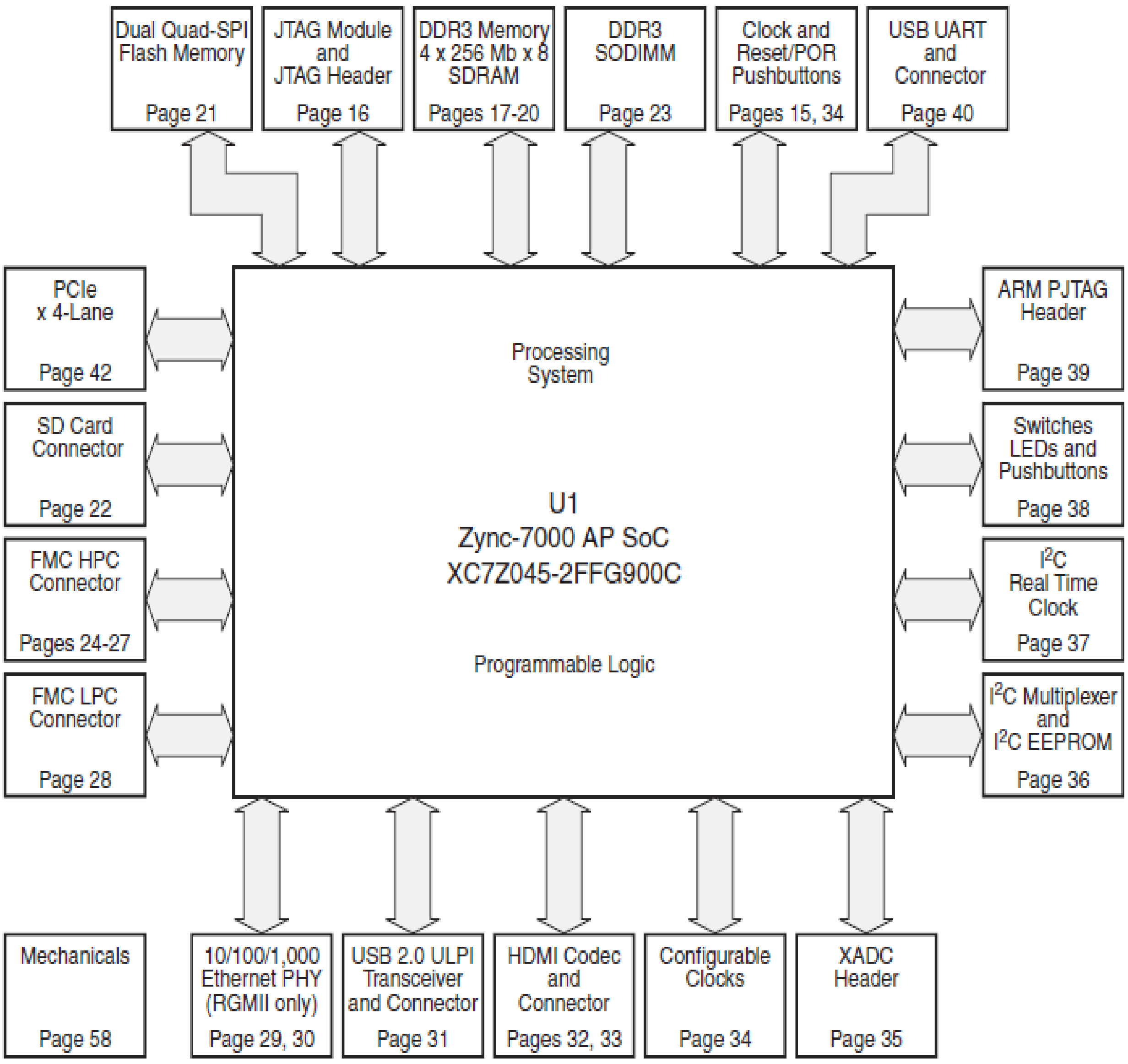

Dual Quad-SPI Flash Memory Page 21
JTAG Module and JTAG Header Page 16
DDR3 Memory 4 x 256 Mb x 8 SDRAM Pages 17-20
DDR3 SODIMM Page 23
Clock and Reset/POR Pushbuttons Pages 15, 34
USB UART and Connector Page 40
PCIe x 4-Lane Page 42
SD Card Connector Page 22
FMC HPC Connector Pages 24-27
FMC LPC Connector Page 28
Processing System
U1
Zync-7000 AP SoC
XC7Z045-2FFG900C
Programmable Logic
ARM PJTAG Header Page 39
Switches LEDs and Pushbuttons Page 38
I2C Real Time Clock Page 37
I2C Multiplexer and I2C EEPROM Page 36
Mechanicals Page 58
10/100/1,000 Ethernet PHY (RGMII only) Page 29, 30
USB 2.0 ULPI Transceiver and Connector Page 31
HDMI Codec and Connector Pages 32, 33
Configurable Clocks Page 34
XADC Header Page 35

Figure 6.1 Zynq ZC706 Evaluation Board Block Diagram[39]

6.2.1 Luma4x4_PredModes Module

In this section the designing procedure for *Luma4x4_PredModes* module is described , focusing on the optimization decisions that had the biggest impact in terms of performance and area. In the end of the section the optimization directives summary is demonstrated , along with the final design's latency details.

Design Basics

Initially the design is synthesized with the default HLS interpretation of C language constructs that will produce an architecture defined by the dependencies in code without any optimization directives. The *clock period* was set at *3 nsec* and after synthesis the tool resulted in a design that facilitates similar sequential nature with the original software program and as a result is rather slow in terms of latency.

Interface Synthesis

When the tool synthesizes the C input program to RTL design , top level function arguments are synthesized into RTL data ports with specific interface protocols set by the designer . Interface synthesis applies differently to functions and function arguments, as in the first case adds control signals to the function/block to control the start of operation , when the data is ready and when the block completes its operation and in the second case an interface protocol is attached to the function argument port , for instance *ap_memory, ap_bus* or *ap_fifo* if we want to implement this specific port as a single interface for memory , bus or fifo respectively.

In the specific module's implementation add block-level handshakes must be added in the design through *ap_ctrl_hs* interface to specify when the block operation can start and when it ends . In addition to this we must ensure that all output ports use an interface that indicates when a write operation is occurred. The above interfaces ensure that we can verify the RTL design without creating RTL test benches through the C/RTL co-simulation feature in Vivado HLS , as a result *ap_ctrl_hs* interface directive and an *ap_memory* interface directive are ensured to be attached to the top level function and the output arrays respectively .

Pipelining

The primary target is to reduce the design latency and pipelining is a very useful technique to parallelize operations for this purpose . Pipeline directives can be applied either on functions to pipeline the operations in the function body or on loops level to explore concurrent execution of these loops in order to reduce latency and improve throughput.

Function Pipelining

In *Luma4x4_PredModes* module *pipeline* directives are applied through Vivado HLS GUI to functions that would be benefited from the issued parallelised execution . However, considering that when a region is pipelined , inner loops of this region are unrolled to satisfy the optimization , pipelining must be treated carefully as unrolling loops of multiple operations and heavy computational load can be a tricky task regarding design's performance. The tool processes the directives and produces a design with pipelined functions and the corresponding timing and latency results. In some cases, significant reduction in latency comes with a necessary increase of the clock period , because of the effort of the tool to pipeline the function .This means that Vivado HLS pursues concurrent execution of the operations and if the defined clock period is not enough to "fit" the scheduled operations the tool tries to extend it until it does.

In *Luma4x4_PredModes* module , functions *ReconstuctIntra4* and *Disto4x4* and the internal *FTransform4, ITransform4* , *QuantizeBlock* and *TTransform4* respectively are mapped as instances into the available logic resources and therefore , they are pipelined , whereas *ITransformOne4* is inlined in the hierarchy as instantiating it would not come with any benefits. This way we can also avoid the latency overhead that transitions amongst the synthesized functions costs.

Loops Pipelining

Since the above instances are pipelined , all internal loops are unrolled and as a result the only unrolled loop remaining in the design is the *PredModes4* loop that executes reconstruction and distortion measurement for 4x4 luma blocks.

Array Optimization

The bottleneck in the specific occasion is the arrays in the function arguments that are implemented with a memory interface and their elements are grouped together . This prevents maximum parallelization when pipelining as concurrent access in different elements of an array is not possible. Even if dual port BRAMs are assigned to the input arrays difficulties occur in the effort to improve initiation interval and consequently design latency . The solution to the problem is *partition* directive that is featured and supports array partitioning, breaking the elements of an array and implementing each one as a register in the RTL design . Arrays can be partitioned in as many different small arrays as desired or even be scalarized totally which is the preferable choice for maximum parallelization. This allows the tool to achieve the minimum initiation interval possible for the function pipeline by scheduling multiple operations in the input arrays to take place at the same time . Arrays containing 4 or less elements function are automatically partitioned by default ,but for the function input arrays a *config_array_partition* command can be set through the configuration settings of the implementation's design solution. With this command all input port arrays with an elements number smaller than the user specified threshold can be scalarized . The impact in the design is radical latency reduction as the tool now can reach a lower initiation interval for the function pipelines and remove the obstacle for reducing loop pipeline latency. Indeed the clock cycles needed to execute the circuit droped around 100% in expense of area cost though, however not in a restrictive manner .

Latency Constraints

We can specify the minimum and maximum latency that is acceptable for either functions or loops as a constraint through the *latency* directive . When no further optimizations in terms of latency and throughput seem to improve the design performance , the tool can be assigned to handle the exploration of additional scheduling and binding alterations to satisfy the defined latency constraints.

Function Dataflow Pipelining

Apart from pipelining operations inside a loop or function to improve throughput and reduce latency , we can optimize the communication between functions with the *dataflow* directive applied to the top level function . With this directive we can create a parallel process architecture for our implementation to enable function call executions to overlap and achieve the lowest latency allowed by data dependencies in the code . Considering that *ReconstructIntra4* and *Disto4x4* are mapped together as   data flow inside the design requires communication amongst the two generated instances , *dataflow pipelining* can improve throughput by enabling the execution of *Disto4x4* before *ReconstructIntra4* finishes as long as data dependencies allow it.

Scheduling and Binding control

Scheduling and binding procedures can be controlled by the designer through the implementation solution configuration settings . When selecting high effort levels for the tool , HLS will explore alternative ways to schedule operations to result in a smaller or faster design consuming more time and system memory as well as spend additional CPU cycles to determine different operation implementations through the device technology library so as to provide better balance of timing and area . Specifically, in the case that optimization decisions force a higher clock period that the targeted high efforts levels in the scheduling process allow HLS to generate a RTL design with acceptable clock period , finding ways to schedule the optimized operations in the program in order to still satisfy the desired functionality and constraints but with a clock period close to the target value.

Final Design

The following tables list the summary of optimization directives used in the specific module’s optimal design , along with the final latency and resource usage details.

Optimization Directives

Directive	*Applied Region*
% HLS DATAFLOW	`“Luma4x4_PredModes”`
% HLS PIPELINE	`“FTransform4”`
% HLS PIPELINE	`“ITransform4”`
% HLS PIPELINE	`“QuantizeBlock”`
% HLS PIPELINE	`“TTransform4”`
% HLS PIPELINE	`“Disto4x4”`
% HLS PIPELINE	`“ReconstructIntra4”`
% HLS INLINE	`ITransformOne4`
% HLS ARRAY_PARTITION -type block -factor 10 -dimension 1	uint8_t ref4[160]

Table 6.1.1 Luma4x4_PredModes Optimization Directives

The final design's timing and latency for *Luma4x4_PredModes* module are presented in tables below , after the application of the optimizations used. Parallelization techniques helped us in order to improve latency radically compared to the initial sequential scheduling of the module's operations and at the same time keep the clock period in the acceptable range . Device utilization also was maintained at low levels and the final resource usage of the fully optimized design is presented in the device utilization section.

Clock	*Target*	*Estimated*	*Uncertainty*
default	3.00	2.63	0,38

Table 6.1.2 ***Luma4x4_PredModes clock timing (ns)***

Latency		***Interval***		***Type***
min	***max***	***min***	***max***	
961	961	961	961	none

Table 6.1.3 Luma4x4_PredModes design latency (clock cycles)

Information about functions below top level function in the function hierarchy are presented separately if this function is not or cannot be inlined. These functions are implemented as independent instances so that further optimizations could be applied either on the function level or the operations inside its body such as loop pipelining , array optimization etc .

Instance	***Latency***		***Interval***		***Type***
	min	***max***	***min***	***max***	
ReconstructIntra4	74	74	75	75	function
Disto4x4	18	18	1	1	function

Table 6.1.4 Luma4x4_PredModes instantiated functions latency

Loops located in the top level function can also be treated separately when it comes to optimization and the latency and pipelining information are produced by HLS always in connection with the designer's directives and constraints. Considering loops in instantiated functions below top level in the hierarchy, latency details are also presented in a specific section in the particular instance’s synthesis report, for the convenience of the designer to locate any critical loops in need of optimizations to boost performance.

Loop Name	***Latency***		***Iteration Latency***	***Initiation Interval***		***Count***	***Pipelined***
	min	***max***		***achieved***	***target***		
Pred_Modes4_Loop	960	960	96	-	-	10	no

Table 6.1.5 Pred_Modes4_Loop latency

Finally , synthesis results include details for the resources required by the submitted design instantiated functions below top level function in the function hierarchy.

Instance	***BRAM_18K***	***DSP48E***	***FF***	***LUT***
ReconstructIntra4	0	32	5081	4257
Disto4x4	0	128	12598	9779

Table 6.1.6 Luma4x4_PredModes instantiated functions resource usage

6.2.2 Luma16x16_PredModes Module

The same procedure was followed for all the implemented modules and this is the reason the other 2 modules design process is presented in a more brief way . The most optimization directives used as well as the strategy followed in the optimization process were similar and any designing differences are declared below . HLS micro-architecture exploration capabilities helped to assess various designs for the implemented module before the optimal solution is selected . Optimization directives used in this module are demonstrated in the end of the section along with the optimal design's latency and resource usage on Zynq XC7Z045 device.

Design Basics

The clock period for this module was initially set at 3 nsec resulting in a design that occurs from the default HLS interpretation of the C input model.

Interface Synthesis

Data control signals were attached to the design's top-level function and the output arrays with *ap_ ctrl_hs* and *ap_memory* interface respectively , as a way to inform the tool about data operations and keep combatibility with C/RTL co-simulation .

Pipelining

Instantiated functions in this module are *ReconstructIntra16* and *Disto16x16* containing *FTransform, ,ITranform, FTransformWHT, ITranformWHT, QuantizeBlock* and *Transform* respectively. It must be highlighted that 2 separate instances are created for *QuantizeBlock* because of the existence of inner loops with variable loop bounds that prevent unrolling. *QuantizeBlock* is called for both dc and ac quantization levels and execution differentiation points imply that independent instances have to be created in order to allow pipeline of the function .The above instances are pipelined whilst *ITranformOne* is inlined into *ITranform* as merging the two instances improves latency.

Loops inside *ReconstructIntra16* that apply 16 4x4 DCTs and inverse DCTs to cover the 16x16 processed block as well as quantization for ac levels are also pipelined as sequential execution occupies a large number of clock cycles.

In the generated design, every loop transition in nested loops costs 1 clock cycle, so we can nearly eliminate this delay by inlining *DistoB* function which is called 16 times inside two nested loops to measure distortion across the 16x16 luma block . This way, HLS compiler will combine the nested loops in one final loop without the intermediate transitions. Consequently we can pipeline this loop with the *pipeline* directive allowing the design to process more data in every cycle and parallelize operations.

Array Optimizations

As mentioned , input arrays allow limited number of accesses when implemented as memories and this leads to not achieving the targeted initiation interval for the function and loops pipeline . Partitioning the arrays guides the tool to implement a parallelized architecture with lower initiation interval as more operations are allowed to operate concurrently , resulting in a 3 times faster circuit.

Function dataflow pipelining was also added to the implementation top level function to improve communication with the rest functions of the design in a parallelized execution of function instances that could boost the performance ..

Final Design

The following tables list the summary of optimization directives used in the specific module's optimal design , along with the final latency and resource usage details.

Optimization Directives

Directive	*Applied Region*
% HLS DATAFLOW	"Luma16x16_PredModes"
% HLS PIPELINE	"ITransform"
% HLS PIPELINE	"FTransform"
% HLS PIPELINE	"ITransformWHT"
% HLS PIPELINE	"FTransformWHT"
% HLS PIPELINE	"TTransform"
% HLS PIPELINE	"Disto16x16_inner_loop"
% HLS PIPELINE	"QuantizeBlock"
% HLS PIPELINE	"QuantizeBlock2"
% HLS INLINE	"ITransformOne"
% HLS INLINE	"DistoB"
% HLS PIPELINE	"ReconstructIntra16_DCT_loop"
% HLS PIPELINE	"ReconstructIntra16_ QuantizeBlock2_loop "
% HLS PIPELINE	"ReconstructIntra16_IDCT_loop "
% HLS ARRAY_PARTITION -type cyclic -factor 16 -dimension 1	`uint16_t y_ac_levels[16][16]`
% HLS ARRAY_PARTITION -type block -factor 4 -dimension 1	`uint8_t ref16[1024]`
% HLS ARRAY_PARTITION -type cyclic -factor 16 -dimension 1	`uint8_t src16[256]`
% HLS ARRAY_PARTITION -type cyclic -factor 16 -dimension 1	`uint8_t yuv_out16[256]`
% HLS ARRAY_PARTITION -type cyclic -factor 16 -dimension 1	"ReconstructIntra16" `int16_t tmp[16][16]`

Table 6.2.1 Luma16x16_PredModes Optimization Directives

Clock	*Target*	*Estimated*	*Uncertainty*
default	3.00	2.63	0.38

Table 6.2.2 Luma16x16_PredModes clock timing (ns)

Latency	*Interval*	*Type*

min	*max*	*min*	*max*	
4529	4529	4529	4529	none

Table 6.2.3 Luma16x16_PredModes design latency (clock cycles)

Instance	*Latency*		*Interval*		*Type*
	min	*max*	*min*	*max*	
ReconstructIntra16	1052	1052	1052	1052	none
TTransform	20	20	3	3	function
TTransform_2	20	20	3	3	function

Table 6.2.4 Luma16x16_PredModes instantiated functions latency

Loop Name	*Latency*		*Iteration Latency*	*Initiation Interval*		*Count*	*Pipeline d*
	min	*max*		*achiev ed*	*target*		
Pred_Modes16_Loop	4528	4528	1132	-	-	4	no
--Disto16x16_inner_loop	72	72	28	3	1	16	yes

Table 6.2.5 Luma16x16_PredModes loops latency

Instance	*BRAM_18K*	*DSP48E*	*FF*	*LUT*
ReconstructIntra16	0	47	15038	32238
TTransform	0	12	3073	3130
TTransform_2	0	12	3073	3130

Table 6.2.6 Luma16x16_PredModes instantiated functions resource usage

6.2.3 Chroma_PredModes Module

Designing procedure for *Chroma_PredModes* module is presented next. Optimization directives applied as well as the outcoming circuit's latency and device utilization detiles are displayed in the end of the section.

Design Basics

Initially a 3 nsec clock period is set corresponding to 333 MHZ frequency and the tool quickly results in an early design based on sequential processor-like execution without any parallelization optimizations and as a result the amount of clock cycles needed for the implementation is large.

Interface Synthesis

This module is also designed with the intention to verify that functional correctness of the algorithm was not compromised because of parallelization and other optimizations . As a result the necessary block level control signals were added to the top level function as well as data validation signals to the output arrays (*ap_ctrl_hs, ap_memory)*, enabling the C/RTL co-simulation procedure to take place inside the same HLS platform . C test benches used to validate the C input algorithm are used again by the tool with the proper adaptation to ensure that RTL design generated produces the same results with the software .

Pipelining

Instantiated functions inside *Chroma_PredModes* module main function (*ReconstructUV)* are *FTransform, ITransform* and *QuantizeBlock* are executed in loops in order to reconstruct 8 4x4 chroma blocks (4 blocks for each plane ,U and V). Consequently , besides pipelining the instances to implement a parallel architecture for the inner operations , the loops containing the function calls can also be pipelined in order to enable reconstruction overlapping for the different 4x4 chroma blocks .

After function -level and loop- level pipelining the conclusion that limited ports in the input arrays block additional parallelization optimizations is reached and array partitioning discussed next will increase throughput and allow more operations to overlap.

Array Optimizations

Pipelining has not resulted in the best possible design in terms of throughput and latency due to restrictions issued by the input arrays memory interface . Limited concurrent accesses can occur and as a result the design created is semi- parallel. Applying *partition* directives to the input arrays can allow more operations to overlap with acceptable resources cost , resulting in a 4 times faster circuit in terms of latency .

Final Design

The following tables list the summary of optimization directives used in the specific module's optimal design , along with the final latency and resource usage details.

Optimization Directives

Directive	*Applied Region*
% HLS DATAFLOW	"Chroma_PredModes"
% HLS PIPELINE	"ITransform"
% HLS PIPELINE	"FTransform"
% HLS PIPELINE	"Disto16x16_inner_loop"
% HLS PIPELINE	"QuantizeBlock"
% HLS INLINE	"ITransformOne"
% HLS PIPELINE	"ReconstructUV_DCT_loop"
% HLS PIPELINE	"ReconstructUV_QuantizeBlock_loop "
% HLS PIPELINE	"ReconstructUV_IDCT_loop "
% HLS ARRAY_PARTITION -type cyclic -factor 8 -dimension 1	`uint16_t uv_levels[4+4][16]`
% HLS ARRAY_PARTITION -type block -factor 4 -dimension 1	`uint8_t ref_uv[512]`
% HLS ARRAY_PARTITION -type cyclic -factor 8 -dimension 1	`uint8_t src_uv[128]`
% HLS ARRAY_PARTITION -type cyclic -factor 8 -dimension 1	`uint8_t yuv_out_uv[128]`
% HLS ARRAY_PARTITION -type cyclic -factor 8 -dimension 1	"ReconstructUV" `int16_t tmp[8][16]`

Table 6.3.1 Chroma_PredModes Optimization Directives

After the application of the above directives the generated circuit's clock period needs fixing as it exceeds the targeted 3 nsec value and using high effort levels for scheduling and binding procedure through the configuration settings of the implementation guides HLS to explore more possibilities in the process of scheduling groups of operations and binding the suitable operators and cores to them. This way a more balanced design is expected to be produced and indeed after a longer elaboration time the tool results in a pipelined and fully optimized design with an acceptable clock period value . High level tool efforts during scheduling and binding has also slightly benefited the design regarding resources used as the elements are allocated in a more effective way.

Clock	*Target*	*Estimated*	*Uncertainty*
default	3.00	2,63	0.38

Table 6.3.2 Chroma_PredModes clock timing (ns)

Latency		*Interval*		*Type*
min	*max*	*min*	*max*	

2201	2201	2201	2201	none

Table 6.3.3 Chroma_PredModes design latency (clock cycles)

Instance	***Latency***		***Interval***		***Type***
	min	***max***	***min***	***max***	
ReconstructUV	548	548	548	548	none

Table 6.3.4 Chroma_PredModes instantiated functions latency

Loop Name	***Latency***		***Iteration Latency***	***Initiation Interval***		***Count***	***Pipelined***
	min	***max***		***achieved***	***target***		
ChromaPredModes_Loop	2200	2200	550	-	-	4	no

Table 6.3.5 Chroma_PredModes loops latency

Instance	***BRAM_18K***	***DSP48E***	***FF***	***LUT***
ReconstructUV	0	15	6600	20022

Table 6.3.6 Chroma_PredModes instantiated functions resource usage

6.3 RTL Verification

The hardware modules generated need to be verified for correct functionality opposed to the software edition of them . Vivado HLS C/RTL co-simulation feature allows us to avoid generating RTL test benches in hardware language to verify the design as it can use the already created C test bench destined for C input software validation and automatically verify the RTL design using an HDL simulator . As already mentioned , functions and output ports must be attached to specific interface protocols during interface synthesis to set HLS able to handle the communication between the C test bench and the generated RTL creating the necessary wrapper and adapters . The main purpose of co-simulation is to check that parallelization optimizations defined by the designer during high-level synthesis did not compromise the original functionality of the C input program and this is the reason HLS tests RTL design's results against the original software function outputs as retrieved during various executions of the program .

6.4 Modules Device Resource Utilization

In the table below the overall utilization on Zynq XC7z045 device is listed for each implemented module, The resources used are limited in a low range despite the extensive parallelization pursued ,with the distinctive note of DSPs used in *Luma4x4_PredModes* module .Even though the particular module is the smallest regarding instantiated functions computational load, the HLS-produced design requires a relatively high number of DSP blocks to reach the achieved performance.

Usage of distributed LUT memory in *Luma16x16_PredModes* is also high , considering the high demands in array storage for the specific module in contrast to the other 2 modules as the size of the blocks to process is 16x16 .

- **Luma4x4_PredModes Module**

Zynq XC7Z045				
Name	***BRAM_18K***	***DSP48E***	***FF***	***LUT***
Expression	-	-	-	1241
FIFO	-	-	-	-
Instance	-	160	17679	14036
Memory	-	-	-	-
Multiplexer	-	-	-	1865
Register	-	-	1897	-
ShiftMemory	-	-	-	-
Total	0	160	19576	17142
Available	1090	900	437200	218600
Utilization (%)	***0***	***17***	***4***	***7***

Table 6.4.1 Luma4x4_PredModes device utilization

- **Luma16x16_PredModes Module**

Zynq XC7Z045				
Name	***BRAM_18K***	***DSP48E***	***FF***	***LUT***
Expression	-	-	-	531
FIFO	-	-	-	-
Instance	-	71	21184	38498
Memory	-	-	-	-
Multiplexer	-	-	-	1119
Register	-	-	1302	1
ShiftMemory	-	-	-	-
Total	0	71	22486	40149
Available	1090	900	437200	218600
Utilization (%)	***0***	***7***	***5***	***18***

Table 6.4.2 Luma16x16_PredModes device utilization

- **Chroma_PredModes Module**

Zynq XC7Z045				
Name	***BRAM_18K***	***DSP48E***	***FF***	***LUT***
Expression	-	-	-	48
FIFO	-	-	-	-
Instance	-	15	6600	20022
Memory	-	-	-	-
Multiplexer	-	-	-	79
Register	-	-	93	-
ShiftMemory	-	-	-	-
Total	0	15	6693	20149
Available	1090	900	437200	218600
Utilization (%)	***0***	***1***	***1***	***9***

Table 6.4.3 Chroma_PredModes device utilization

Chapter 7
Design Space Exploration

This chapter focuses on the hardware accelerated modules interconnection as well as details about their communication with the CPU. The system's high-level architecture is presented along with design space exploration discussion. Finally, different architectures on the targeted device are studied by setting an alternative clock period for the synthesized design and the modules overall device utilization is demonstrated

7.1 WebP System Architecture

The principal concept of the system is that each HW accelerated module will be executed independently as reconstruction of different types of blocks (luma4x4/16x16, chroma) takes place at separate time in the original software algorithm . Consequently, the data required for each module must be sent from the CPU to the board and this costs some time that will be calculated in the overall execution time of the module. Specifically, each module needs the source pixels block(4x4, 16x16. 8 x16)* and the already predicted reference blocks of the same size for all the available prediction modes as well as the quantization matrices for the specific segment of the frame.

In numbers , 0.375 Kbytes need to be transferred for *Luma4x4_PredModes* module, 2.125 Kbytes for *Luma16x16_PredModes* module and finally 1Kbyte for *Luma4x4_PredModes* module . Supposing a 390 MB/sec bus transfer rate the data input time is estimated to be 0.939 , 5.321 and 2.504 μsec respectively for each module. When it comes to the results, data written back are calculated to be 0.507, 3.14 and 1.257 Kbytes for each implemented module, resulting in a corresponding delay of 1.27 , 7.86 and 3.14 μsec .

Fortunately the input data as well as the time needed to write the results back to the CPU that handles the software-executed functions are the only data transaction delays ,as the internal HW modules processing already has the necessary data to produce the fully reconstructed blocks and the corresponding distortion for all prediction modes , taking advantage by the fact that the modules internal instances are placed one next to other and no further communication with the CPU is demanded since the flow has reached the computation stage. The system high level architecture is demonstrated in *Figure 7.1* below along with data flow across the modules.

*Chroma blocks are treated as 2 8x8 blocks , one for each plane (U,V)

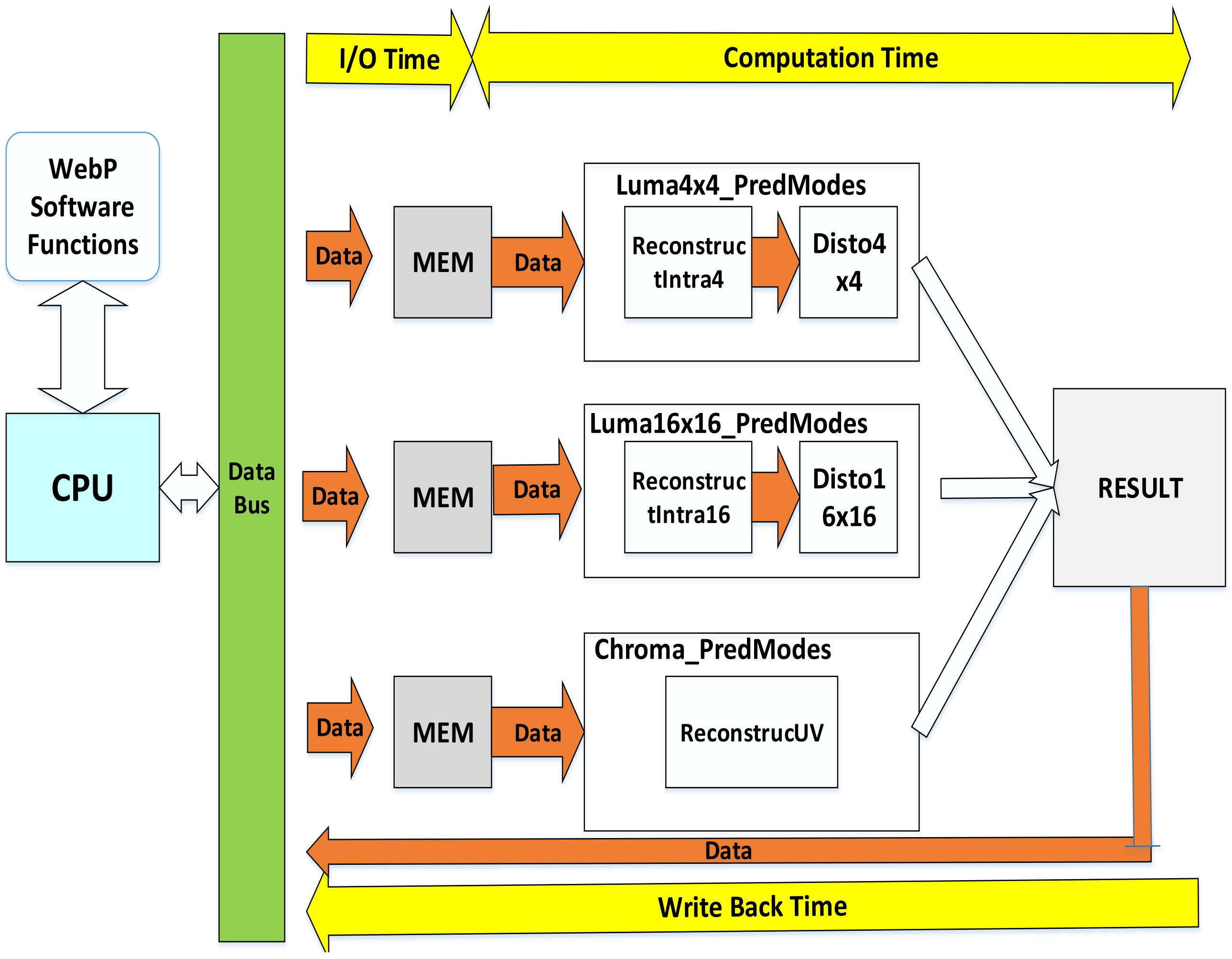


Figure 7.1 System high-level architecture

Taking as a fact that each image block processing in the implemented circuit is independent, the proposed architecture can adopt a pipelining logic by placing appropriate memories right before and after the hardware computation stage. This way software processing and data loading to the board can overlap with the execution of the hardware accelerated modules .

Additionally, the reconstructed blocks as well as the distortion values can be written back to the CPU while the next block is submitted to the design for processing . Considering that input data loading and write back time are smaller but of a comparable scale with the strict computation time , any achieved reduction in communication delay can have a greatly positive impact on the design's performance.

Furthermore, as long as device utilization allows it , more than one module cores as well as the necessary additional memories can be added to the implemented system in order to make the simultaneous processing of multiple blocks possible.

7.2 Overall Device Resource Utilization

Tables below demonstrate overall device utilization on 2 different target boards as well as 2 different clock periods for each board , 5 and 3 nsec , corresponding to 200 and 333 MHZ frequency respectively .

We notice that resource usage for the system is relatively low consuming as only LUT usage exceeds 30% for the 3 nsec clock period architecture that ,as the performace evalutation indicates ,offers the biggest performance speedup. This allows the implementation of more than one instances of the modules created to achieve further acceleration of the system.

Zynq XC7Z045 (3 nsec clock period)				
Module	***BRAM_18K***	***DSP48E***	***FF***	***LUT***
Luma4x4_PredModes	-	160	19576	17142
Luma16x16_PredModes	-	71	22486	40149
Chroma_PredModes	-	15	6693	20149
Total	***0 / 1090 (0 %)***	***246 / 900 (27 %)***	***49135 / 437200 (11%)***	***78485 / 218600 (35 %)***

Table 7.1 Overall device utilization on Zynq XC7Z045 (3 nsec clock period)

The 5 nsec clock period architecture demonstrates slightly higher utilization on the targeted device but addition of processing cores is still possible.

Zynq XC7Z045 (5 nsec clock period)				
Module	***BRAM_18K***	***DSP48E***	***FF***	***LUT***
Luma4x4_PredModes	-	160	15205	16534
Luma16x16_PredModes	-	73	15918	40073
Chroma_PredModes	-	44	7455	18377
Total	***0 / 1090 (0 %)***	***277 / 900 (30 %)***	***38958 / 437200 (8 %)***	***75997 / 218600 (34 %)***

Table 7.2 Overall device utilization on Zynq XC7Z045 (5 nsec clock period)

Chapter 8

Hardware Performance Evaluation

In this chapter the performance of the hardware modules generated is evaluated ,compared to the software version of them .

The following table demonstrates the clock cycles needed to execute the hardware accelerated modules in our generated circuit . Clock periods set through the design process were transformed to frequencies in order to express the board's clock rate. It is worth mentioning that since a new design clock period is set , HLS tool performs synthesis process from the beginning and that is the reason the generated circuit differs regarding clock cycles needed for its execution, as a totally new architecture is produced that issues proper scheduling of the operations in the designing process , as defined by the available clock period.

Module	***Clock Period (5 nsec)***	***Clock Period (3 nsec)***
Luma4x4_PredModes	681	961
Luma16x16_PredModes	3841	4529
Chroma_PredModes	1845	2201

Table 8.0.1 Clock cycles for hardware design execution

Based on the clock cycles table above , the time needed to execute the hardware modules is calculated and presented in the following table for 2 different clock frequencies , 200 and 333 MHZ :

Module	***Time @200 MHZ(μsec)***	***Time@333 MHZ (μsec)***
Luma4x4_PredModes	3,405	2,883
Luma16x16_PredModes	19,205	13,587
Chroma_PredModes	9,225	6,603

Table 8.0.2 Time needed for hardware design[1]

***[1] Not including data read/write from/to CPU**

Speedup results over software execution are presented next.

8.1 Speedup @ 200 MHZ

As the following table shows, the achieved speedup at 200 MHZ clock rate is slightly above 2X for *Chroma_PredModes* and a little lower than 2,5X and 3X for *Luma4x4_PredModes* and *Luma16x16_PredModes* respectively. Calculated hardware time includes the communication delay between CPU and the hardware modules as well as the actual computation time consumed .

Module	***Software Time (μsec)***	***Hardware Time (μsec)***	***Speedup***[2]
Luma4x4_PredModes	13,3	5,615	***2,37***
Luma16x16_PredModes	92	32,390	***2,84***
Chroma_PredModes	30	14,878	***2,02***

Table 8.0.3 Speedup @200 MHZ

***[2] Including data read/write from/to the CPU**

8.2 Speedup @ 333 MHZ

When it comes to the design that can achieve a 333 MHZ clock rate, a significant improvement is noticed in all three hardware accelerated modules with a speedup that reaches almost up to 3,5X for *Luma16x16_PredModes* and slightly above and lower than 2,5X for *Luma4x4_PredModes* and *Chroma_PredModes* respectively.

Module	***Software Time (μsec)***	***Hardware Time (μsec)***	***Speedup***[3]
Luma4x4_PredModes	13,3	5,094	***2,61***
Luma16x16_PredModes	92	26,772	***3,44***
Chroma_PredModes	30	12,256	***2,45***

Table 8.0.4 Speedup @333 MHZ

***[3] Including data read/write from/to the CPU**

8.3 Overall Speedup

According to Amhdahl's law equation the overall speedup results are calculated and displayed in *Table 8.5.* Two different configurations were selected in order to relate the performance gains to as real as possible the studied conditions can be. More specifically, the selected configurations is a comparison of the execution time that the 3 modules accumulatively require in software or hardware accelerated version, taking into consideration the execution times calculations mentioned in the previous chapters.

Considering around 70% of the WebP algorithm is hardware accelerated , the theoretical maximum speed up would be about 3,33 , leading to the conclusion that above half of the acceleration potential is exploited by the proposed implementation.

Configuration	*Speedup @200 MHZ*	*Speedup@333 MHZ*
Lenna Image (400x400)	1,71	1,82
High Resolution Image(4096x2074)	1,59	1,67

Table 8.0.5 Overall Speedup[4]

***[4] Including: I/O time for software**
Data read/write from/to CPU for hardware

The implemented modules that run at 333 MHZ can assist in speeding up the algorithm for Lenna image conversion up to 1,8X and a high resolution image up to 1,67X , showing a more promising perspective .

Even though the architecture running at 200 MHZ presents slightly higher utilization in the targeted board and lower speedup performance for the studied configurations , it may result in a more efficient usage for devices that operate under certain restrictions and have difficulties to reach a 333 MHZ clock rate.

8.4 Overall Speedup with overlapped I/O

Adopting the pipelined architecture described in section 7.1 can result in further increase of the expected speedup considering the proposed implementation. The time required for the data transfer from the CPU to the board is smaller than the processing time of the hardware accelerated modules, but the delay is still important considering that for a whole image the modules will be executed multiple times. The following table presents the speedup for the same configurations and clock frequencies when the input data transfer is overlapped with the processing stage in the hardware accelerated modules. It must be noted that the first data transfer to the design from the CPU can be ignored as the total number of calls to the modules will be several thousands.

Configuration	***Speedup @200 MHZ***	***Speedup@333 MHZ***
Lenna Image (400x400)	1,86	1,99
High Resolution Image(4096x2074)	1.71	1,81

Table 8.6 Overall Speedup with overlapped I/O

The speedup results in this case follow a similar pattern as for Lenna the achieved speedup is larger than the second configuration, however for both configurations and board frequencies a significant improvement in accelerating the implemented modules in noticed. More specifically, a clock rate of 200 MHZ can lead to a 1,86X speedup of the algorithm for Lenna and slightly higher than 1,7X for the high resolution image, whereas a design running at 333MHZ can reach a speedup factor of almost 2X for Lenna and above 1,8 for 4K image. The desired trade-off between performance and the achievable clock rate as well as possible power consumption limitations are the factors that will determine the final design choice.

Chapter 9

Conclusions and Future Work

9.1 Conclusions Summary

This thesis proposes an effective way to improve WebP algorithm's performance. Time consuming functions that occupy around 70.6% of the total execution time as the applied profile analysis indicated are implemented into reconfigurable fabric using Xilinx's Vivado High Level Synthesis as an efficient alternative to traditional manual hardware design in Hardware Description Languages. The achieved overall speed up calculated using Amhdal's Law equation reaches up to 1,99X regarding the studied configurations , maintaining a relatively low resources usage in the targeted device.

9.2 Future Work

Techniques proposed in section 7.1 considering communication overhead reduction between the CPU and the hardware accelerated modules can be expanded to a more complicated pipelined architecture to maximize parallelism in the design. In addition to this, adoption of a multi-core processing architecture in a suitable device can lead to further improve of the system's performance.

Furthermore, the power consumption of the implemented hardware design can be calculated and alterations in the modules and overall system architecture can be explored so as to satisfy the low power profile indicated by current trends in the related industry applications.

Finally , the described system architecture can be implemented beyond the abstract level that this thesis presents by prototyping the embedded system and studying its behavior in real conditions.

Bibliography

[1] Wikipedia , "*Digital Image Processing*".

[2] S. Hauck, A. Dehon," *Reconfigurable Computing: The Theory and Practice of FPGA-based Computing "* , Elsevier 2008 , pp 26-36

[3] Xilinx,"*User Guide 998,Introduction to FPGA Design Using High-Level Synthesis* ".

[4] A. Manan, " *Implementation of Image Processing Algorithm on FPGA*", Akgec Journal of Technology, Vol 2 , No 1.

[5] C.Shen, W. Plishker, S.S.Bhattacharyya, "*Dataflow-based Design and Implementation of Image Processing Applications",* Multimedia Image and Video Processing(2nd Edition), L. Guan, Y. He, S.Kung , pp 609-629., ISBN: 978-1-4398-3087-1

[6] A. Boudabous , L. Khriji, A.Ben Atitallah, P. Kadionik, N. Masmoudi, "*Efficient Architecture and Implementation of Vector Median Filter in Co-Design Context",* Radioengineering, Vol 16 , No 3, September 2007.

[7] Chip Design Trends 2012 ,"*ASIC/ASSP Prototyping with FPGAs*" survey. http://chipdesignmag.com/sld/blog/tag/asic/

[8] D. Salomon, "*Handbook of Data Compression",* 5th edition, Springer, 2010, 1383 p., ISBN 1848829027.

[9] Li R. Jung, K., Al-Shamakhi, N. 2002, "*Image compression using transformed vector quantization. Image and Vision Computing*", Volume 20, Issue 1, 1 January 2002, Pages 37-45

[10] WebP homepage, https://code.google.com/speed/webp/

[11] Webp Compression Study, https://developers.google.com/speed/webp/docs/webp_study

[12] Z. Wang, A.C. Bovik, H.R. Sheikh, E.P. Simoncelli," *Image quality assessment: from error visibility to structural similarity"*, IEEE Transaction on Image Processing 13 (4) (2004) 600–612.

[13] F. De Simone, L. Goldmann, J.Lee, T. Ebrahimi, " *Perfomance Analysis of VP8 image and video compression based on subjective evaluations",* Proc. SPIE 8135 , Applications of Digital Image Processing XXXIV (September 2011)

[14] J.Alakuijala, "*Lossless and Transparency Encoding in WebP",* http://developers.google.com /speed/webp/docs/webp_lossless_alpha_study.

[15] WebP Container Specification , https://developers.google.com/speed/webp/docs/riff_container

[16] M. Pintus, G. Ginesu, L. Atzori, D. Giusto, " *Objective Evaluation of WebP Image Compression Efficiency",* 7th International ICST Conference, MOBIMEDIA September 2011, Cagliari, Italy , pp 252-265.

[17] WebP's Compression Techniques, https://developers.google.com/speed/webp/docs/compression

[18]] J. Bankoski, P. Wilkins, Y. Xu, Google Inc," *Technical Overview of VP8, an open source video codec for the Web ",* ICME '11 Proceedings of the 2011 IEEE International Conference on Multimedia and Expo .

[19] Pankaj Kumar Bansal, Vijay Bansal, Mahesh Narain Shukla, Ajit Singh Motra , " *VP8 Encoder - Cost Effective Implementation* ", Software, Telecommunications and Computer Networks (SoftCOM), 20th International Conference (2012)

[20] J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, Y. Xu, Google Inc, "*VP8 Data Format and Decoding Guide* ", http://datatracker.ietf.org/doc/rfc6386/ , November 2011.

[21]Inside WebM Technology: VP8 Intra and Inter Prediction ,http://blog.webmproject.org/2010/07/inside-webm- technology-vp8-intra-and.html

[22] R. Kordasiewicz, S. Shirani, "*ASIC and FPGA Implementations of H.264 DCT and Quantization Blocks",* IEEE International Conference on Image Processing, Vol. 3, 2005, pp. 1020-1023.

[23] K. Suh, S. Park, H. Cho,"*An Efficient Hardware Architecture of Intra Prediction and TQ/IQIT Module for H.264 Encoder",* ETRI Journal, Volume 27, No 5, October 2005.

[24] Heng-Yao Lin, Yi-Chih Chao, Che-Hong Chen, Bin-Da Liu, and Jar-Ferr Yang," *Combined 2-D Transform and Quantization Architectures for H.264 Video Coders",* Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on, May 2005, pp 1802-1805 Vol. 2

[25] I. Amer, W.Badawy, and G.Jullien," *A High Performance Hardware Implementation of the H.264 Simplified 8x8 Transformation and Quantization",* IEEE International Conference on Acoustics, Speech, and Signal Processing 2005 (ICASSP'05), vol. 2, pp. ii/1137 -ii/1140, Philadelphia, PA, 18-23 March 2005.

[26] Osman, H., Mahjoup, W., Nabih, A. , Aly, G.M.,"*JPEG Encoder for low-cost FPGAs",* Computer Engineering & Systems, 2007. ICCES '07. International Conference on, November 2007, Cairo, pp 406-411.

[27] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, D. Sciuto*," A Pipelined Fast 2D-DCT Accelerator for FPGA-based SoCs",* ISVLSI'07. IEEE Computer Society Annual Symposium on, 331-336.

[28] W. Elhamzi, T. Saidani, M. Atri and R. Tourki*," On Hardware Implementation of DCT/IDCT for Image Processing*", Signals, Circuits and Systems, 2008. SCS 2008. 2nd International Conference on , 7-9 November 2008.

[29] S. Cassidy, "*An Analysis of VP8, a New Video Codec for the Web*", Rochester Institute of Technology, 2011.

[30] S. Krishnaprasad, "*Uses and Abuses of Amdahl's Law,*" J. Computing Sciences in Colleges, vol. 17, no. 2, pp. 288-293, Dec. 2001.

[31] G. Martin , G. Smith," *High-Level Synthesis: Past, Present, and Future* ", Design & Test of Computers, IEEE (Volume:26 , Issue 4) , July-Aug. 2009 , p 18-25

[32] J. Cong, B.Liu , S. Neuendorffer , J. Noguera , K. Vissers , Z. Zhang," *High-Level Synthesis for FPGAs: From Prototyping to Deployment",* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on (Volume:30 , Issue: 4) *,* April 2011 *, p. 473-491*

[33] W. Meeus , K.,Van Beeck , T. Goedemé , J. Meel , D. Stroobandt," *An overview of today's high-level synthesis tools",* Design Automation for Embedded Systems, pp. 1-21, 2012

[34] K. Wakabayashi, *"C-based behavioral synthesis and verification analysis on industrial design examples,"* in *Proc. ASPDAC*, 2004, pp. 344–348

[35] Xilinx,"*User Guide 902, Vivado Design Suite : High Level Synthesis ",* xilinx.com

[36] Berkeley Design Technology*,* Inc.*,"An Independent Evaluation of High-Level Synthesis Tools for Xilinx FPGAs"*

[37] Tom Feist, "*Vivado Design Suite", White Paper ,June 2012*

[38] F.M. Sanchez, R. Mateos, E.J. Bueno, J. Mingo and I. Sanz , *"Comparative of HLS and HDL` Implementations of a Grid Synchronization Algorithm"* in Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE p 2232-2237

[39] Xilinx , "*User Guide 954, ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC* ", July 13, Xilinx.com