NTNU
Norwegian University of
Science and Technology

JPEG2000 Image Compression In Hardware

Ole Kristian Hamre Sørlie

Master of Science in Electronics
Submission date: July 2017
Supervisor: Bjørn B. Larsen, IES

Norwegian University of Science and Technology
Department of Electronic Systems

Abstract

The NTNU Test Satellite (NUTS) is a double CubeSat satellite developed by students at the Norwegian University of Science and Technology (NTNU) with the intention of eventually launching it into low earth orbit. The goal of the project is primarily to establish two way communication between the satellite and a ground station on earth and to transmit telemetry data from onboard sensors and images from the onboard camera module.

This thesis aims to implement the JPEG2000 image compression format on an FPGA on the camera module of the satellite using the VHDL programming language. The purpose of image compression is to reduce the amount of data needed to store the images, which in turn reduces the amount of data which has to be transmitted down to earth through the limited downlink. The wavelet transform based JPEG2000 format was selected because of its superior quality at low bit rates and its robustness to bit errors compared to the cosine transform based JPEG format.

The JPEG2000 compression system was partially implemented in VHDL, with the final encoding stages of the system remaining incomplete due to time constraints. The current system consists of six independent modules: a demosaicing module, a gamma correction module, an intercomponent transform module, a wavelet transform module, a quantization module and finally the incomplete encoding module.

The demosaicing module transforms raw Bayer encoded images into RGB images, while the gamma correction module performs gamma correction to account for distortion effects added by the image sensor on the camera module. The intercomponent transform module performs an irreversible color transform on the image, while the wavelet transform module performs a multi-level discrete wavelet transform on the image before it is scalar quantized in the quantization module. Afterwards the result would have been encoded in the encoding module, which is only partially implemented.

Each module was tested against MATLAB implementations in addition to tests performed in Xilinx Vivado. The tests indicated that the modules performed as expected and that they were synthesizable with an acceptable hardware resource usage. Timing Analysis tests also show that the maximum operating speed of the compression system is 63 MHz, which makes it able to process a 2592x1944 resolution image in around 866 ms when not accounting for the encoding process of the system.

Sammendrag

NTNU Test Satellite (NUTS) er en dobbel CubeSat satellit utviklet av studenter ved Norges teknisk-naturvitenskapelige universitet (NTNU) med intensjonen om å sende den i opp i lav jordbane. Målet med satellitten er hovedsakelig å få opprettet toveis kommunikasjon mellom satellitten og en bakkestasjon samt å kunne sende telemetridata fra sensorer om bord og bilder fra kameramodulen.

Målet med denne avhandlingen er å implementere JPEG2000 bildekomprimering i VHDL for bruk på en FPGA som inngår i kameramodulen til satellitten. Målet er å komprimere bilder tatt av en kamerasensor for å redusere datamengden som trengs får å sende bildet til bakkestasjonen. Dette er hovedsakelig på grunn av satellittens begrensede båndbredde. Det wavelet transformasjonsbaserte JPEG2000 formatet ble valgt fordi den gir bedre komprimering ved lavere bitrate og er mer robust med tanke på bit feil enn det diskret cosine transformasjonsbaserte JPEG formatet.

Resultatet ble en delvis implementering av JPEG2000 komprimeringssystemet i VHDL, med en ukomplett encoding-del på grunn av tidsbegrensninger. Det implementerte systemet består av seks moduler: en demosaicingmodul, en gammakorreksjonsmodul, en interkomponent transform modul, en wavelet transform modul, en kvantifiseringsmodul og en delvis implementert encodingmodul.

Demosaicingmodulen transformerer et Bayerkodet råbilde fra kamerasensoren til et RGB bilde, mens gammakorreksjonsmodulen foretar en gammakorrigering av bildet som gjør det mulig å fjerne eventuelle artefakter forårsaket av kamerasensoren. Interkomponent transform modulen utfører en irreversibel fargetransformasjon, mens wavelet transform modulen foretar en multi-level diskret wavelet transformasjon på bildet før resultatet blir skalarkvanfitisert i kvantifiseringsmodulen. Resultatet ville deretter ha blit kodet i encoding-modulen, som i dette prosjektet kun er delvis implementert.

Alle modulene ble testet mot en MATLAB implementasjon med samme funksjonalitet, samt tester utført i Xilinx Vivado. Testene indikerer at alle modulene fungerer som forventet og at de kan konstrueres i hardware. Tidsanalyser viste at systemet kunne kjøre med en klokkefrekvens på 63 MHz, som gjør det mulig å prosessere et bilde med en resolusjon på 2592x1944 piksler i løpet av 866 ms når man utelukker encodingdelen.

Acknowledgements

I would first of all like to thank my academic supervisor Bjørn B. Larsen for the help and advice he has offered throughout the work on this thesis through weekly meetings. I would also like to thank my project supervisor Amund Gjersvik, who has provided vital feedback throughout the whole course of the project as well.

I also extend my gratitude to the entire NUTS team, who through weekly meetings motivate and inspire any and all involved in the project through the long hours spent on the NTNU rooftop.

Lastly i want to thank my friends and family for the never ending support and motivating words they have offered throughout the semester.

Ole Kristian Hamre Sørlie
Trondheim, 03.07.2017

Problem description

The NTNU Test Satellite is a CubeSat satellite being developed by students at the Norwegian University of Science and Technology. The satellite is divided into hardware modules, where each module has its own specific purpose.

The purpose of the camera module is to be able to capture, compress and store images while the satellite is in low earth orbit. A prototype hardware implementation of the camera module has already been developed, but the image compression aspect has yet to be completed.

The purpose of this task will be to develop a JPEG2000 image compression system which is to be implemented on an FPGA located on the camera module. The image compression system will take raw images from a camera sensor and compress it to reduce the space required to store the image. The work will be a continuation of work previously conducted on the camera module.

Key tasks will be:

- Test and verify the functionality of the previously developed parts of the compression system.
- Design and implement a JPEG2000 compression system in VHDL or Verilog to be used on the onboard FPGA of the camera module.
- The compression rate and other factors should be configurable after the satellite has been launched.
- The compression system should have a minimum operating frequency of 50 MHz.
- Test various aspects of the system, such as operating speed, hardware resources required, different compression rates, etc.

Contents

List of Figures

List of Tables

List of Abbreviations

ADC ..... Analog to Digital Converter

BAC ..... Binary Arithmetic Encoder

BPC ..... Bit Plane Coder

BPP ..... Bits Per Pixel

BRAM ..... Block Random Access Memory

CDF ..... Cohen–Daubechies–Feauveau

CFA ..... Color Filter Array

CP ..... Cleanup Pass

dB ..... Decibel

DSP ..... Digital Signal Processor

DUT ..... Device Under Test

DWT ..... Discrete Wavelet Transform

EBCOT ..... Embedded Block Coding with Optimized Truncation

FE ..... Failing Endpoints

FIR ..... Finite Impulse Response

FPGA ..... Field Programmable Gate Array

GB ..... Green-Blue

GBRG ..... Green, Blue, Red, Green

GR ..... Green-Red

ICT ..... Irreversible Color Transform

IDWT ..... Inverse Discrete Wavelet Transform

IEEE ..... Institute of Electrical and Electronics Engineers

IICT ..... Inverse Irreversible Color Transform

IRCT ..... Inverse Reversible Color Transform

LPS ..... Less Probable Symbol

LSB ..... Least Significant Bit

LUT ..... Look-Up Table

LWT ..... Lazy Wavelet Transform

HDL ..... Hardware Description Language

HPFP ..... Half Precision Floating Point

JPEG ..... Joint Photographic Experts Group

JPEG2000 ..... Joint Photographic Experts Group 2000

MPS ..... More Probable Symbol

MRC ..... Magnitude Refinement Coding

MRP ..... Magnitude Refinement Pass

MSB ..... Most Significant Bit

MSE ..... Mean Square Error

MSQE ..... Mean Square Quantization Error

MQ ..... Matrix Quantizer

NaN ..... Not a Number

NTNU ..... Norwegian University of Science and Technology

NUTS ..... NTNU Test Satellite

PSNR ..... Peak Signal-to-Noise Ratio

RAM ..... Random Access Memory

RCT ..... Reversible Color Transform

RGB ..... Red, Green, Blue

RLC ..... Run Length Coding

SC ..... Sign Coding

SNR ..... Signal-to-Noise Ratio

SoC ..... System on Chip

SP ..... Significance Pass

TCQ ..... Trellis Coded Quantization

THS ..... Total Hold Slack

TNS ..... Total Negative Slack

VHDL ..... VHSIC Hardware Description Language

VHSIC ..... Very High Scale Integrated Circuit

WNS ..... Worst Negative Slack

WHS ..... Worst Hold Slack

ZC ..... Zero Coding

1 Introduction

This chapter introduces the Norwegian University of Science and Technology Test Satellite (NUTS) project, selected previous work conducted for NUTS which is relevant to the thesis and an outline of the thesis structure.

1.1 Project background

The purpose of the NTNU Test Satellite (NUTS) program is to design, build and eventually deploy a miniaturized satellite following the double CubeSat standard[3] into low earth orbit.[4] The project was started in 2010, and consists of contributions made by both master and volunteer students from NTNU. Although no complete satellite has been build or launched yet from the program, an engineering module is scheduled to be finished by September 2017. From the beginning, the main goal of the NUTS program has been to provide students with an opportunity to learn about satellites and other space related technology.

Goals related directly to the NUTS satellite itself is to establish a two way communication between the satellite and a ground station once the satellite has reached low earth orbit, to be able to control the attitude of the satellite, to receive telemetry data from onboard sensors and to receive images taken from outer space by the camera module.

1.2 The NTNU Test Satellite

The NUTS satellite is a double CubeSat satellite which is built up from several submodules. Each module is designed to perform a specific set of tasks, such as power management, attitude control and radio communication. These modules interact with each other through a backplane, as is illustrated in Figure 1.1

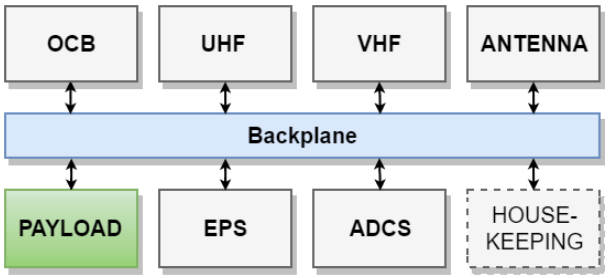

Figure 1.1: Overview of the NUTS Cubesat modules.

The backplane links the modules together and provides the them with power from the electric power system (EPS) module. The EPS module is responsible for charging the onboard batteries through solar panels mounted on the outside of the satellite, as well as to provide regulated power to the other modules.

The onboard computer (OCB) module is the brain of the satellite and is responsible for monitoring the other modules and receive commands from the ground station back on earth. The instructions are received through the ultra high frequency (UHF) and very high frequency (VHD) radio modules, which provides communication between the satellite and the ground station. The communication itself is performed through the antenna module, which contains the antennas needed for UHF and VHF communication. The housekeeping module is integrated in the backplane and ensures that the modules connected to the backplane behaves as they should. The attitude determination and control system (ADCS) provides telemetry about the attitude of the satellite. Through an electromagnetic attitude control system the orientation of the satellite can be changed through the ADCS. The Payload module, which is the focus of this thesis is covered in the next section.

1.2.1 The Payload module

One of the primary goals of the satellite is to be able to capture images from outer space and transmit them back to earth. Originally it was intended for the payload module to be a camera module with an infrared camera that would be capable of observing atmospheric gravitational waves.[5] However this idea was scrapped due to the expense of the IR camera itself.

It was decided that the camera module would instead consist of a normal image sensor capable of capturing light in the visible spectrum. The images would then be stored on the camera module until requested by the ground station, upon which the image would be transmitted down to earth. Figure 1.2 shows an overview of the components of the payload camera module.

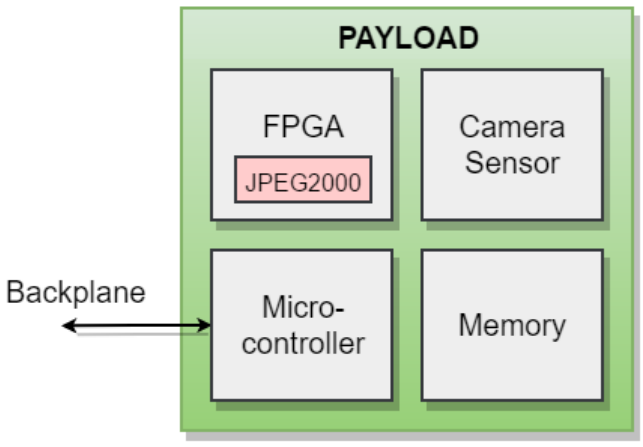

Figure 1.2: Overview of the camera module.

As shown in Figure 1.2, the module consists of four main components. The camera sensor captures raw images which are stored in the memory component. The current sensor used in the previously developed camera module is the MT9P031 5 megapixel image sensor with a resolution of 2592x1944.[6] The purpose of the microcontroller is to interface the camera module with the rest of the satellite. Lastly, the intended purpose of the FPGA is to compress the images taken by the camera sensor in order to reduce the space required to store and transmit the images. The field programmable gate array (FPGA) is a hardware unit which can be programmed using hardware descriptive languages (HDLs) such as VHDL and Verilog. The purpose of this thesis is to develop the compression system which is to be used on the FPGA. It was previously determined that JPEG2000 was the best suited image compression format for this application. [7] The goal is therefore to implement the JPEG2000 image compression format in VHDL.

1.3 Previous work

This section presents a brief summary of the most recent and relevant work conducted on the payload camera module for the NTNU Test Satellite, which provides the foundation for this thesis.

1.3.1 Demosaicing module

In 2016 a demosaicing module was developed for the FPGA which was intended to convert the raw Bayer pattern image from the camera sensor into an RGB image.[1] The work also included a gamma correction module and color transform module, which is needed when performing image compression using JPEG2000. It was intended to use these modules as part of the compression system developed in this thesis, however it was decided early on to develop new modules from scratch instead, with the reasoning behind this decision detailed in Section 3.1.

1.3.2 Prototype camera module

In 2016 a new prototype camera module was developed.[8]. It improved upon the previous camera module[9] and addressed noise issues experienced when operating the module at high frequencies. Controlling the camera module is an Spartan-6 FPGA[10] running a MicroBlaze[11] softcore microprocessor, which contains the necessary software framework to interface both the camera sensor and an external microcontroller with the FPGA. The framework was developed in 2014 as part of a separate project.[12] Although the prototype camera module and the software framework is not used directly in relation to this thesis, it nevertheless forms the platform which the compression system is to be implemented on.

1.4 Thesis outline

Chapter 2 covers the theory relevant to the thesis. It includes basic image compression metrics and an exposition of the JPEG2000 image compression standard, which covers topics such as image demosaicing, the wavelet transform, quantization and encoding.
Chapter 3 details the hardware implementation of the compression system in VHDL. It includes flow charts, module architectures, and implementation and timing analysis results.
Chapter 4 presents the testing strategy used to test the compression system and presents the results obtained.
Chapter 5 discusses the results obtained in Chapter 4 and presents suggestions for future work.
Chapter 6 presents the conclusion of the thesis.

2 Theory

This chapter covers the theory related to the implementation of the JPEG2000 compression system. Several sections make use of an image of Lena Söderberg[13], which is a well known test image often used in image processing studies.

2.1 Image compression

The purpose of image compression is to reduce the amount of data space needed for either storage or transmission of an image. The compression can be done either lossless or lossy. Lossless image compression means that none of the information contained in the image is lost during the compression process, and a perfect reconstruction of the original image is possible. This is in contrast to lossy image compression, where only an approximation of the original image can be reconstructed from the compressed image. This is because some information contained in the original image is sacrificed to achieve a higher rate of compression, which in turn reduces the data capacity needed to store the compressed image. The amount of compression performed on an image is often given as the compression ratio, where for instance a compression ratio of 1:40 means a 40 times reduction in the size of the compressed image compared to the original. Another metric used is bits per pixel (bpp), which is the average number of bits needed to represent a single pixel in an image. For an uncompressed 8-bit image the bpp is 8, while a compressed image can have a bpp in the range of 0.05-1, as is illustrated in Section 2.11.

Lossless image compression is the process of removing what is known as redundant information from the image. One way to achieve this is by exploiting spatial redundancy often found in images, which is the fact that neighboring pixels often have the same pixel intensity. Compression algorithms such as run-length encoding can use this to represent the image with less information, as illustrated in Figure 2.1, and still reconstruct a perfect image.

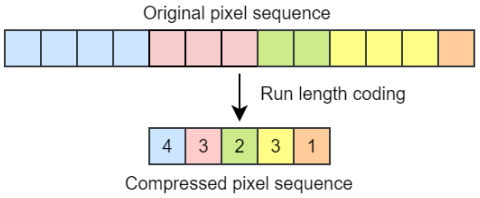

Figure 2.1: Lossless compression achieved by exploiting spatial redundancy through run-length encoding.

As Figure 2.1 illustrates, a pixel sequence of the colors [b,b,b,b,r,r,r,g,g,y,y,y,o] can be represented simply as [4b,3r,2g,3y,1o], which requires less data to store. This gives a simplified idea of how lossless image compression is achieved.

When performing lossy compression, both the redundant and irrelevant information in an image is removed. Irrelevant information is information in the image which is not considered important and can therefore be discarded in the compression process. Which aspects of an image that is considered to be important or not varies, but it is usually chosen to give the most optimal viewing experience for a human observer. For instance, the human eye can usually not discern small differences in intensities between two spatially close pixels in an image. If the difference is small enough, the intensity of the two pixels can be changed to the average of the two pixel intensities instead. The two pixels are now equal in terms of pixel intensity, which makes them spatially redundant, which again can then be exploited with for instance run-length encoding. This reduces the overall data storage needed to store the two pixels. However, the original information in the two pixels are now lost, meaning that they can not be reconstructed perfectly when the compressed image is decompressed. This is what makes the process lossy.

2.1.1 Image quality

As outlined in the previous section, lossy image compression aims to compress an image at the cost of the quality of the reconstructed image. The quality in this case is defined as the perceived or mathematically calculated difference between the original image and the compressed image after after it has been decompressed. In terms of perception, since most images are meant to be viewed by a human observer there is no perfect way to assess the quality of the reconstructed image, as the perceived quality is usually subjective. A mathematically calculated difference gives an objective measurement of the quality, but is not always reflective of the subjective quality given by a human observer. The mathematically determined difference can nevertheless provide an insight into the efficiency of an image compression or image manipulation process.

When judging image quality a term often used is the peak signal-to-noise ratio (PSNR) between the original image and the reconstructed image. The PSNR of a grayscale image can be defined through the mean squared error (MSE) between the original and reconstructed image. The MSE of an m x n image O compared to an image R of equal size is defined as

$$MSE = \frac{1}{mn} \sum_{i=1}^{m-1} \sum_{j=1}^{n-1} [O(i,j) - R(i,j)]^2 \tag{2.1}$$

From which the PSNR in dB is defined as

$$PSNR = 10 \cdot log_{10}\left(\frac{MAX_O^2}{MSE}\right) \tag{2.2}$$

Where MAX_O is the maximum pixel intensity in the original image, which for an 8-bit image is 255. For an RGB image the same definition of PSNR is used, but the MSE is calculated by summing the differences between the three image components and dividing the result by three. Simply determining the signal-to-noise (SNR) ratio can also be used to indicate image quality, but it is more common to express the quality using PSNR.

With lossy image compression, a common value for the PSNR between an original image and reconstructed image is generally between 30 and 50 dB, with the higher value generally providing a better image quality but also generally resulting in a lower rate of compression. An example of comparative compression quality and PSNR is shown in Section 2.11.

2.2 Overview of JPEG2000 compression

JPEG2000 is an image compression standard developed by the Joint Photographic Experts Group which was intended to supplant the already successful JPEG compression standard. Wavelet-transform based JPEG2000 offers several advantages over the discrete cosine transform based JPEG with regards to bit-rate performance and robustness to bit errors.[14] Figure 2.2 shows the building blocks of the JPEG2000 compression system. [15] It should be noted that the demosaicing part in the preprocessing block is not strictly a part of the JPEG2000 format, however, the raw image has to be converted into RGB before the compression process starts. The demosaicing process is further explained in Section 2.3.2.

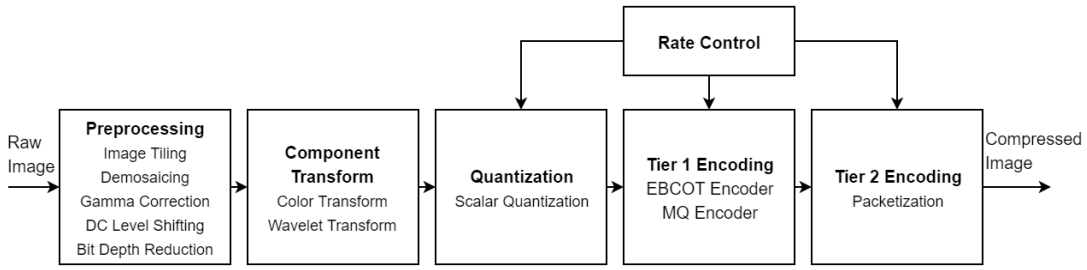

Figure 2.2: The components that make up the JPEG2000 compression system.

The JPEG2000 compression system can broadly be divided into five stages:

- **Preprocessing** - Prepares the raw image for compression.
- **Component transform** - Performs color transformation and a wavelet transform of the image.
- **Quantization** - Quantizes the transformed image using scalar quantization.
- **Tier 1 Encoding** - Encodes the quantized wavelet coefficients using Embedded Block Coding with Optimized Truncation (EBCOT) coding and binary arithmetic (MQ) coding.

- **Tier 2 Encoding** - Packages the encoded image into a compressed image.

The rate control block controls the compression rate by adjusting parameters for the quantization and encoding steps of the compression process. All the stages of the process are covered in the subsequent sections. To decompress a compressed image, the system is simply run in reverse, with the exception of the demosaicing process which is then omitted.

2.3 Preprocessing

Preprocessing is any manipulation of the raw image data from the camera sensor before the actual JPEG2000 compression begins. In this project, the image has to go through image tiling, demosaicing, gamma correction, dynamic range adjustment and bit depth reduction before the compression can begin. This section will deal with the theory behind each of these steps in order.

2.3.1 Image tiling

Image tiling is a process in which the image is broken down into rectangular tiles which are compressed separately. This reduces the internal memory and processing power required of the compression system since only parts of the image is processed at a time. The JPEG2000 compression standard allows tile sizes of NxN with N being a power of 2 and ranging from 1 to 4096.[16] Figure 2.3 shows a 512x512 image divided into 16 tiles, with each tile consisting of 128x128 pixels.

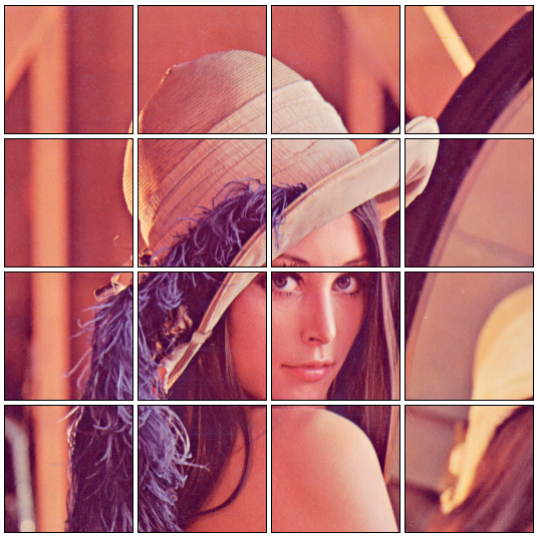

Figure 2.3: Tiling of a 512x512 image into 128x128 tiles.

Since a tile can be up to 4096x4096 pixels, it is possible to process the whole image as a tile. This would naturally require far more hardware resources to accomplish, which

is why a system with limited hardware resources benefits from processing the image as separate tiles. Once the image has been divided into tiles, the tile is split into its constituent components, which in this case is the red, green and blue layer. Figure 2.4 illustrates the components which make up the tile to be processed.

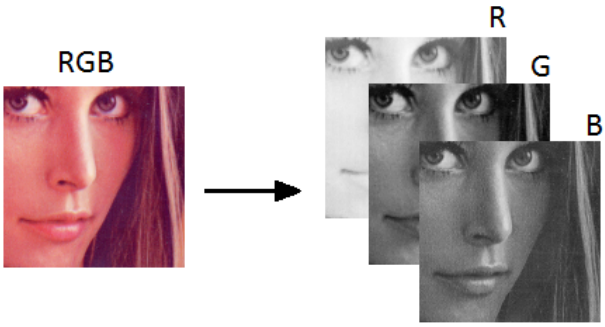

Figure 2.4: Tiling of the three color components.

Each component tile of the original tile is then processed separately. It should be noted that usually all the tiles that make up a color component are processed in order, meaning that all the tiles of the red component are processed, then all the tiles of the green, etc. This is not the case in this project, since the demosaicing module reads in a 128x128 Bayer pattern tile from the external memory, and the resulting red, green and blue component tiles are stored internally. If the system were to process each color component separately, it would have to first perform the demosaicing process on the entire Bayer pattern image, store the color components on the external memory and then read back each component and compress them in turn. This intermediate storage of the demosaiced image can be avoided if the compression of the color components is done out of order. Because each tile is treated as a separate image in the JPEG2000 standard, this reordering of the tiles will not affect the performance of the system. However, this means that the JPEG2000 decoding system would have to be designed to rearrange the tiles back into the correct order once the image has been received, or the tiles has to be numbered accordingly in the encoding part of the compression system.

As mentioned, the size of the tiles affect the memory and resource requirements of the compression system, with a larger tile size requiring more hardware resources in general. However, tile size also has an effect on the quality of the decompressed image. Choosing a smaller tile size introduces blocking artifacts similar to those seen in images compressed using the JPEG standard. Figure 2.5 shows a comparison of different tile sizes and the resulting image quality.

The images were compressed with JPEG2000 using the image compression program VCDemo, with a bitrate of 0.3 bpp and a 5-level wavelet transform. This program is covered more thoroughly in Section 2.13. The only difference between the images seen in Figure 2.5 was the tile sizes. As can be seen, a 32x32 tile size causes a significant drop in image quality, with blocking artifacts clearly visible in the image. The difference between the 128x128 and 64x64 tile size is not as visible, but the PSNR is 1.5 dB higher

Figure 2.5: The effects of tiling on the compressed image.

with the larger tile size. In the end its a choice between higher image quality or smaller memory footprint, with a 128x128 tile size yielding good results without requiring too much hardware resources to implement.

2.3.2 Demosaicing

A typical image sensor used in modern photography consists of thousands upon thousands of small pixel sensors arranged in a grid like pattern. Each pixel sensor is capable of measuring light intensities with a resolution determined by the ADC of the camera, which is usually somewhere in the range of 8 to 12 bits depending on the application. The camera sensor used in this project, the MT9P031[6], has an internal ADC with a 12 bit resolution, giving each pixel a value between 0 and 4096 which translates to the brightness of the light hitting the sensor. A value of 0 would indicate no light, while 4096 would indicate the highest possible brightness.

Since the pixel sensors only measures light intensities, the images they generate would only be grayscale images, as they are unable to capture colors innately. In order to capture images in color, a color filter array (CFA) is used. A CFA is a mosaic of

color filters placed over the entire surface of the image sensor, where the size of each filter corresponds to the size of the pixel sensor. The CFA is aligned in such a way that one color filter covers a pixel sensor completely, meaning that the only light hitting the sensor is the light that passed through that particular filter. This means that the pixel sensor now measures the intensity of whatever color passed through the color filter instead of light from the whole color spectrum. This is what allows the capturing of color information.

The CFA most often used is the Bayer filter mosaic, which is a pattern of red, green and blue filters arranged in a square pattern consisting of two green filters, one red and one blue filter. Figure 2.6 shows such a pattern and how it is used to construct the Bayer filter mosaic.

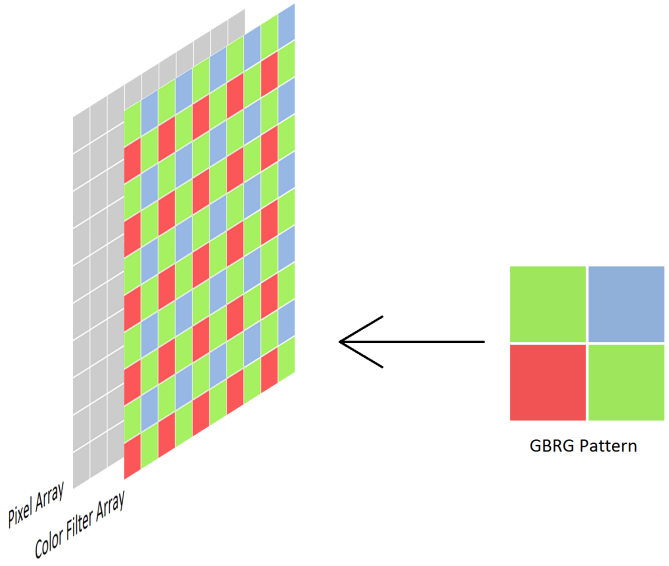

Figure 2.6: The principle of a Color Filter Array.

Here, "GBRG" stands for "green, blue, red, green", which is simply the ordering of the color filters from left to right, top to bottom. There exists several different patterns, which uses the same color combination but in different order. Figure 2.7 shows what a color coded raw Bayer encoded image would look like. The image was generated in MATLAB and is color coded for visual clarity, as in reality a Bayer encoded image is visually simply a grayscale image where each pixel is mapped to a particular color.

There exists various algorithms to transform the Bayer encoded image into an RGB image, which is what is referred to as demosaicing, an essential part of the preprocessing step of the JPEG2000 compression pipeline. However, they are all based on interpolation of the red, green and blue pixel values based on the neighbouring pixels. The simplest demosaicing algorithm is the nearest-neighbor interpolation algorithm, which simply copies the nearest neighboring pixel of the same color.

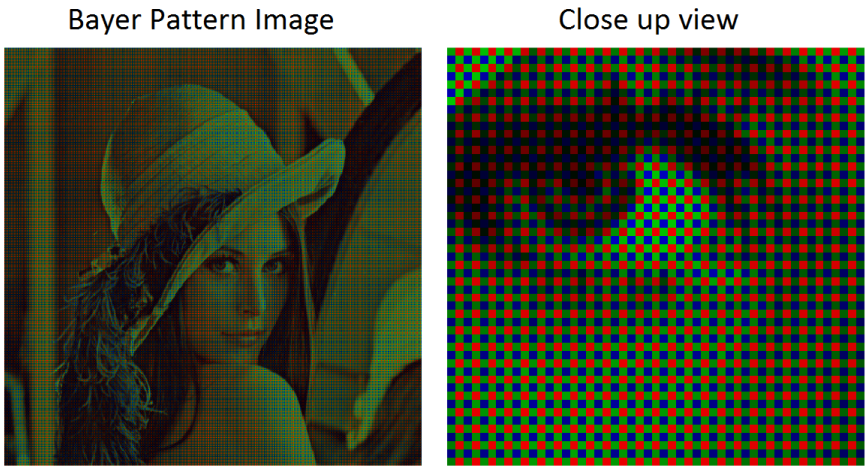

Figure 2.7: Color coded Bayer encoded image using the 'GBRG' pattern.

One such algorithm was proposed by Malvar et.al [17], and is based on weighted bilinear interpolation. It works by using a weighted average of several neighboring pixels, which yields better results than nearest-neighbour interpolation or unweighted average bilinear interpolation. The values for the weights and which of the neighbouring pixels that are used to perform the interpolation depends on which color in Bayer pattern image that is currently being processed. For a more in-depth explanation of the algorithm, it is recommended that the reader refer to the original article by Malvar et.al[17] Using the pattern and CFA from Figure 2.6, and the two nearest neighbouring pixels there are four different combinations of adjacent colors, as shown in Figure 2.8.

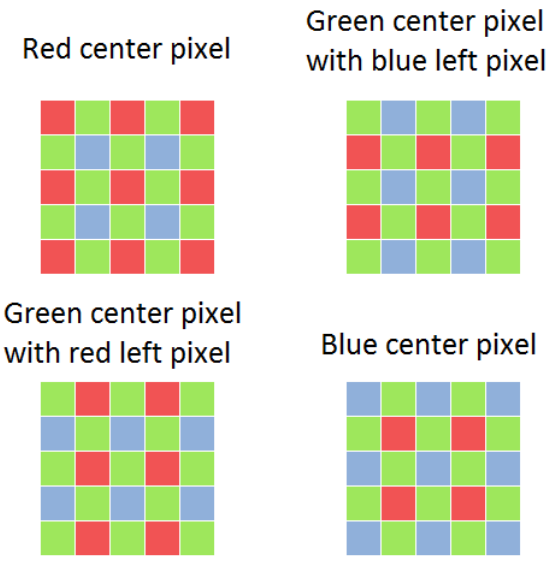

Figure 2.8: The four Bayer image color combinations.

As is shown in Figure 2.6 each combination forms a 5x5 array of pixels. For these four combinations, Malvar et.al suggested the weights as shown in Figure 2.9.

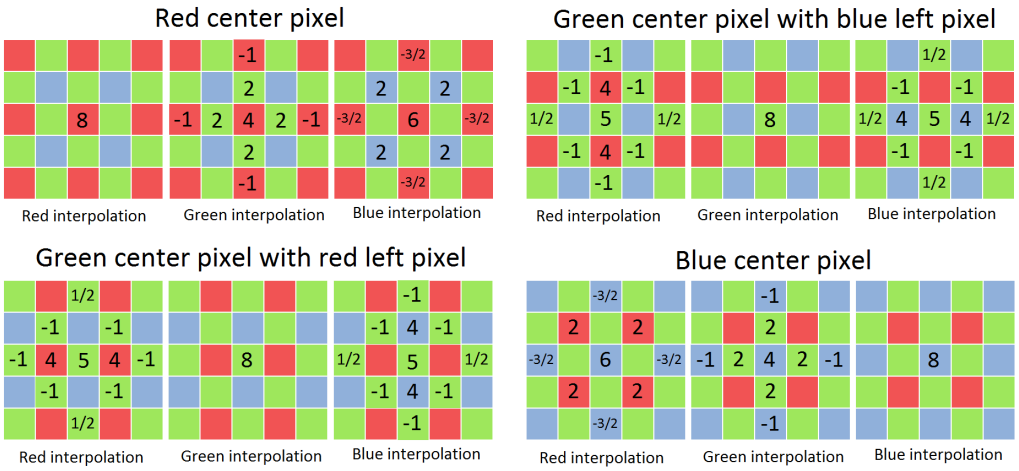

Figure 2.9: The weights used for interpolation, here multiplied by a factor of 8 to reduce the number of fractions.

From Figure 2.9 the weights attributed to the different pixels in the array gives rise to the four equations

$$\begin{bmatrix} R_R \\ G_R \\ B_R \end{bmatrix} = \begin{bmatrix} 8P_{2,2} \\ -P_{0,2} + 2P_{1,2} - P_{2,0} + 2P_{2,1} + 4P_{2,2} + 2P_{2,3} - P_{2,4} + 2P_{3,2} - P_{4,2} \\ -3P_{0,2} + 4P_{1,1} + 4P_{1,3} - 3P_{2,0} + 12P_{2,2} - 3P_{2,4} + 4P_{3,1} + 4P_{3,3} - 3P_{4,2} \end{bmatrix} \cdot \begin{bmatrix} 1/8 \\ 1/8 \\ 1/16 \end{bmatrix} \tag{2.3}$$

$$\begin{bmatrix} R_{GB} \\ G_{GB} \\ B_{GB} \end{bmatrix} = \begin{bmatrix} -2P_{0,2} - 2P_{1,1} + 8P_{1,2} - 2P_{1,3} + P_{2,0} + 10P_{2,2} + P_{2,4} - 2P_{3,1} + 8P_{3,2} - 2P_{3,3} - 2P_{4,2} \\ 16P_{2,2} \\ P_{0,2} - 2P_{1,1} - 2P_{1,3} + P_{2,0} + 8P_{2,1} + 10P_{2,2} + 8P_{2,3} + P_{2,4} - 2P_{3,1} - 2P_{3,3} + P_{4,2} \end{bmatrix} \cdot \begin{bmatrix} 1/16 \\ 1/8 \\ 1/16 \end{bmatrix} \tag{2.4}$$

$$\begin{bmatrix} R_{GR} \\ G_{GR} \\ B_{GR} \end{bmatrix} = \begin{bmatrix} P_{0,2} - 2P_{1,1} - 2P_{1,3} - 2P_{2,0} + 8P_{2,1} + 10P_{2,2} + 8P_{2,3} - 2P_{2,4} - 2P_{3,1} - 2P_{3,3} + P_{4,2} \\ 8P_{2,2} \\ -2P_{0,2} - 2P_{1,1} + 4P_{1,2} - 2P_{1,3} + P_{2,0} + 10P_{2,2} + P_{2,4} - 2P_{3,1} + 8P_{3,2} - 2P_{3,3} - 2P_{4,2} \end{bmatrix} \cdot \begin{bmatrix} 1/16 \\ 1/8 \\ 1/16 \end{bmatrix} \tag{2.5}$$

$$\begin{bmatrix} R_B \\ G_B \\ B_B \end{bmatrix} = \begin{bmatrix} -3P_{0,2} + 4P_{1,1} + 4P_{1,3} - 3P_{2,0} + 12P_{2,2} - 3P_{2,4} + 4P_{3,1} + 4P_{3,3} - 3P_{4,2} \\ -P_{0,2} + 2P_{1,2} - P_{2,0} + 2P_{2,1} + 4P_{2,2} + 2P_{2,3} - P_{2,4} + 2P_{3,2} - P_{4,2} \\ 8P_{2,2} \end{bmatrix} \cdot \begin{bmatrix} 1/16 \\ 1/8 \\ 1/8 \end{bmatrix} \tag{2.6}$$

From the equations, the R, GB, GR and B notations indicate the color of the center pixel from which the interpolation is made. GB and GR stands for green center, blue on the left and green center, red on the left respectively. Some of the weight have also

been multiplied in order to make them integers instead of fractions, and therefore the subsequent scaling is also adjusted. As can be seen, the scaling is done by dividing the weights by a power of 2, in this case either 8 or 16. Because of this, the division can be accomplished in hardware using simple bit shifting, which makes the operation trivial compared to division by numbers which are not a power of 2. All the weights are also integers, which greatly simplifies the multiplication as well and removes the need for fixed or floating point arithmetic in the demosaicing part of the JPEG2000 pipeline. This also means that the resulting values from the interpolation equations will have to be rounded to integers, which naturally introduces some small rounding errors in the demosaicing process. It should also be noted that the interpolated pixel values have to be clipped to ensure that they do not go outside the dynamic range of the bits used to represent them. In the case of 8 bits, the valid range is 0 to 255 for unsigned and -128 to 127 for signed.

A special case which has to be handled during the demosaicing process is dealing with the edges of the image. The weights and subsequent interpolation equations derived from Figure 2.9 expects a 5x5 array of valid pixels. This is obviously not possible when handling the edges if the images, and doubly so for the corners. The pixels located in the corners of the image only have 3x3 valid pixels instead of 5x5, as the other pixels are located "outside" the image. One way to handle these cases is to simply set the pixel values that end up outside the image to 0. Naturally this is not an optimal solution, as it will cause the interpolated pixels to appear darker than they actually are, as shown in Figure 2.10.

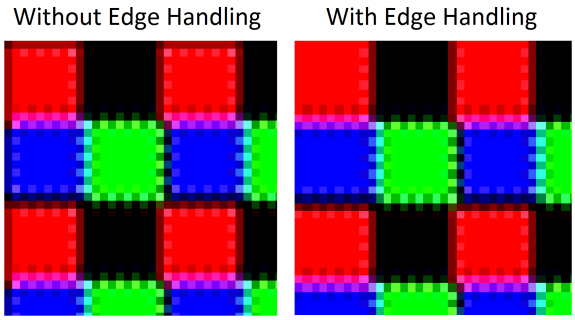

Figure 2.10: Comparisons of demosaicing with and without image edge handling.

A better way to deal with edges and corners is to treat them as special cases and reconfigure the interpolation equations accordingly. This essentially means changing the scaling part of the equations to reflect the number of "valid" pixels in the 5x5 pixel array. If a pixel is outside the image, it is treated as an "invalid" pixel and excluded from the equation. From the interpolation equations, we see that sum of the edge weights divided by the scaling factor equals 1. Once a weighted pixel becomes invalid, the scaling factor should be adjusted so that the sum of the valid weights multiplied by the new scaling factor equals or nearly equals 1. Figure 2.11 shows the valid pixels and their weights at the upper left corner of the image assuming that the center pixel is blue.

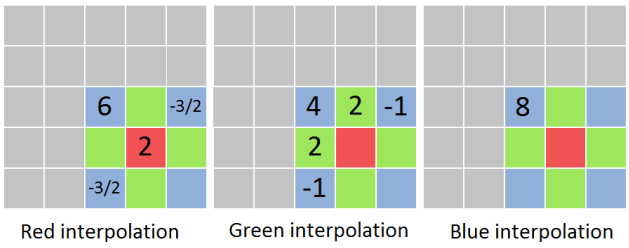

Figure 2.11: Weighted pixels for a blue center pixel at the corner of the image.

From Figure 2.11, if the pixel weights when interpolating the red pixel value is summed, the result is 5. The scaling factor therefor has to be adjusted from 1/16 to 1/10 in order to get a result of 1, and a similar re-scaling is done on the other interpolation equations as well, as shown in Equation 2.7.

$$\begin{bmatrix} R_B \\ G_B \\ B_B \end{bmatrix} = \begin{bmatrix} 12P_{2,2} - 3P_{2,4} + +4P_{3,3} - 3P_{4,2} \\ 4P_{2,2} + 2P_{2,3} - P_{2,4} + 2P_{3,2} - P_{4,2} \\ 8P_{2,2} \end{bmatrix} \cdot \begin{bmatrix} 1/10 \\ 1/6 \\ 1/8 \end{bmatrix} \tag{2.7}$$

As can be seen from the equation, the scaling factor no longer has a divisor which is a power of 2. Therefore, an approximation of the scaling factor is performed, which adjusts the scaling factors divisor to the closest integer which is a power of 2. The scaling factors in Equation 2.7 is therefore adjusted to 1/8, 1/8 and 1/8. This introduces a small error in the demosaicing process around the edges, but the error is considered to be acceptable when compared to the hardware cost of implementing proper division.

Figure 2.12 shows the result of a 512x512 RGB image which was transformed to a Bayer pattern image, and then demosaiced using the Malvar et.al bilinear interpolation algorithm using MatLab.

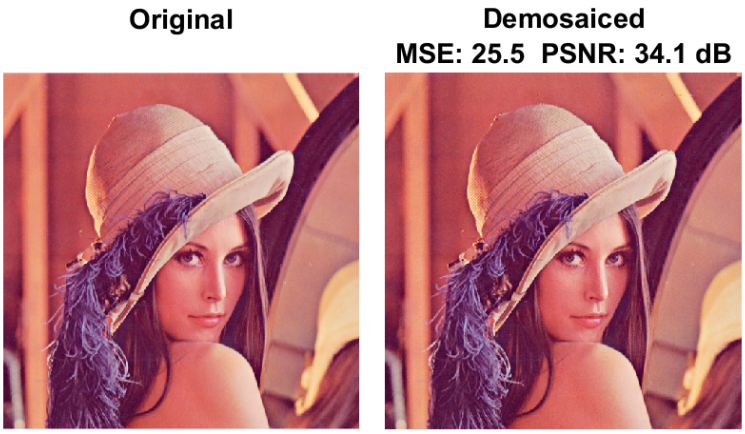

Figure 2.12: Original image vs image transformed to Bayer pattern and then demosaiced.

The demosaiced image in Figure 2.12 shows that the bilinear approach produces good results given its simplicity and the ease of which it can be implemented in hardware. It is also hard to see if there are any artifacts caused by the demosaicing process as well, because the image has an overall low frequency, meaning that there are few very sudden changes in pixel intensities. Figure 2.13 shows the same demosaicing process performed on another image which contains more abrupt changes from one color to another. The image was generated using a test image script from an earlier project for NUTS.[1]

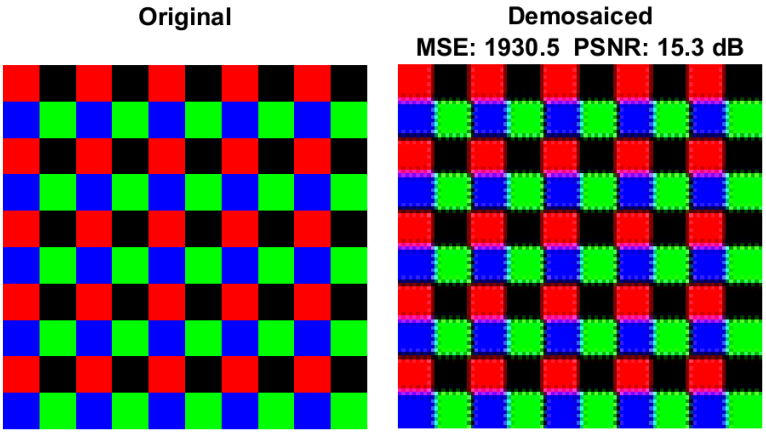

Figure 2.13: Example of visible demosaicing artifacts in high frequency regions of an image.

It is now easy to see that the demosaicing does indeed introduce some zipping artifacts around high frequency areas of the image. Other demosaicing algorithms, such as bicubic interpolation or a Lanczos resampling based demosaicing algorithm might have produced a result with a lesser degree of artifacts. It is however expected that the image taken by the camera module will be so called "natural" images, which are images of nature scenes, which are in general low frequency images. The hardware cost of implementing the more complex algorithms would also be far greater.

2.3.3 Gamma correction

Gamma correction is a process in which the perceived brightness of an image is corrected using the power law expression

$$Y = AX^{\gamma} \tag{2.8}$$

where X is the pixel value to be corrected, Y is the corrected pixel value, A is a constant and γ is the gamma value, usually in the range $\gamma = (0,1)$. The constant A is usually set to 1, but it can also be set to a value in the range $A = [0,1]$ and act as a scaling constant.

Figure 2.14 shows an example of gamma correction being used on a grayscale image. The gamma correction were performed in MATLAB by implementing Equation 2.8 together with different constants, which causes the brightness and contrasts of the image to change.

Figure 2.14: Gamma corrections on a grayscale image.

When gamma correction is performed on a multicomponent image, such as an RGB color image, the process is performed on each color component independently. The gamma correction of a RGB image can be performed using the equations

$$\begin{bmatrix} Y_{\mathrm{R}} \\ Y_{\mathrm{G}} \\ Y_{\mathrm{B}} \end{bmatrix} = \begin{bmatrix} A_R \cdot {X_R}^{\gamma_R} \\ A_G \cdot {X_G}^{\gamma_G} \\ A_B \cdot {X_B}^{\gamma_B} \end{bmatrix} \tag{2.9}$$

where the R, G and B subscripts indicate the red, green and blue components. By using the scaling factor A and gamma factor γ, one can manipulate the colors of the image, making each color appear brighter or darker. This can then for instance be used to reduce the influence one color has on the image. This is useful when using camera sensors where the sensitivity of the pixel sensors are stronger for one color than another. This would cause that color to "tint" the image in that specific color, and gamma correction could be used to remove or reduce this effect.

Figure 2.15 shows some examples of gamma correction being used to alter the colors of an image. The gamma correction was performed using Matlab with an implementation of Equation 2.9. Since the original image in this case needed no gamma correction, the "corrected" images will appear to be tinted in a particular color.

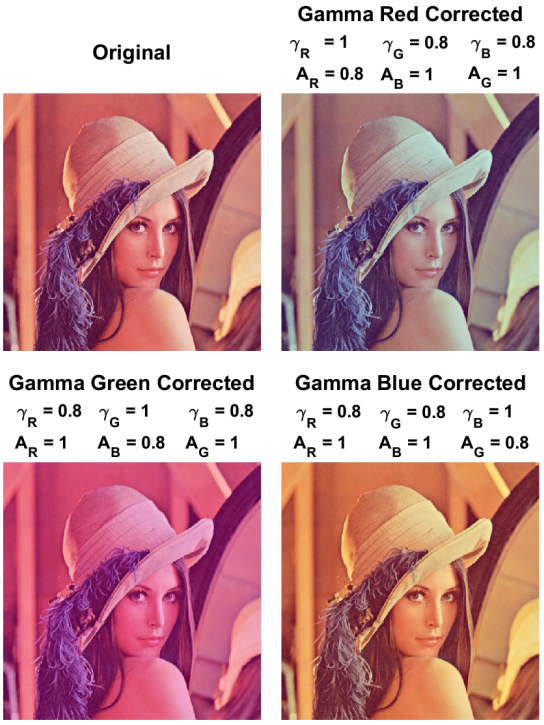

Figure 2.15: Gamma corrections on a RGB image.

2.3.4 Zero-centered dynamic range

Each pixel in the raw image from the camera sensor used on the camera module has a bit depth of 12 bits. This means that every pixel is represented using 12 bits. Each pixel thus has a decimal value in the range

$$[0,\ 2^{bitdepth} - 1] = [0,\ 4095] \tag{2.10}$$

This means that the minimum value a pixel can have is 0, which corresponds to black, and the maximum value is 4095, which corresponds to white. The nominal dynamic range is in this case therefore centered around the middle of the minimum and maximum value, which equals 2047. The JPEG2000 codec however expects a nominal dynamic range centered around zero.[16] In order to shift the centering to a zero-centered dynamic range, a bias of $2^{bitdepth-1} - 1$ is subtracted from each sample, and the sample is also converted from unsigned to two's compliment signed.

This shifting of the dynamic range is later undone in when the image is decompressed. The shift results in the new zero-centered decimal range of

$$[-2047,\ 2048] \tag{2.11}$$

for each pixel, which is now centered around 0. This process is also known as DC level shifting.

2.3.5 Bit depth reduction

Bit depth reduction reduces the number of bits needed to represent a pixel in an image. The MT9P031 image sensor, which is the sensor used in the prototype payload module, has an ADC resolution of 12 bits, which means that each pixel in the image is represented using 12 bits. If the raw output from the image sensor was a gray-scale image, that means that the pixels in the image can have 2^{16} different intensity levels.

After demosaicing, each pixel of the raw image is now represented as a combination of red, green and blue samples. This corresponds to the layers of an RGB image, which naturally is the output of the demosaicing module. The red, green and blue components are also represented using 12 bits per sample. Consequently, without bit depth reduction, each pixel in the demosaiced image needs 36 bits in total since each color component uses 12 bits each. This means that the image can have a total of 2^{36} different colors, or over 68 billion colors. Naturally, this is far more colors than the camera sensor itself is actually capable of capturing, and also far more colors than the human eye can discriminate, which is around 10 million. [18]

Thus, in order to reduce the bits needed to store the three color layers, all the samples are downscaled from 12 bits to 8 bits after the demosaicing stage of the compression pipeline. This gives the color component samples, red, green and blue, a total of 2^8 intensity levels each, which is a decimal range of

$$[0,\ 255] \tag{2.12}$$

Now, the total number of bits needed to represent a pixel using the three color components is 24, and the amount of colors that can be represented in the picture is 2^{24}, or just under 17 million. This bit depth reduction of the color components reduces the total storage needed for the components by 1/3, which again reduces the hardware resources needed for the rest of the JPEG2000 compression pipeline.

Before demosaicing, a raw image from the MT9P031 image sensor, which is the sensor used in the prototype payload module, has a size of 2592x1944 pixels, which with 12 bits per pixel results in 7 588 kB of storage needed for each raw image. After demosaicing, and without bit depth reduction each color component has the same dimensions as the raw image and has the same 12 bit resolution, which translates to three times the size of the raw image in total, or 22 764 kB. With bit depth reduction from 12 to 8 bits, the size is reduced to 15 116 kB. It is however possible to downscale further, reducing the size further, although at a cost of image quality. Figure 2.16 shows the result of reducing the bit depth of a grayscale image further.

Figure 2.16: Different bit depths for an grayscale image.

As can be seen Figure 2.16, reducing the bit depth in order to save on data storage space has a big impact on the picture quality. The same is true for RGB images, as shown in Figure 2.17.

Figure 2.17: Different bit depths for an RGB image.

Here, the R, G and B components are each represented by a bit depth 8, 4, 2 or 1 bit. This makes the total bits needed per pixel 24, 12, 6 and 3 bits respectively. Although the storage space required can be reduced by simply reducing the bit depth of the color components, it is not advisable to do so below 8 bits per color component, as the trade-off between picture quality and storage space is too great. Further reduction in the image data size is accomplished during the JPEG2000 compression.

2.4 Color transform

The color transform part of the compression pipeline transforms the demosaiced image from the RGB color space to the YCbCr color space. This transformation can improve the PSNR of the decoded image compared to the original by as much as 1.5 dB [15]. There are two types of transformations specified in the JPEG2000 compression format, the reversible color transform (RCT) and the irreversible color transform (ICT)[19]. The RCT can be used with both lossless and lossy compression, and is primarily used together with the CDF 5/3 wavelet transform, as this transform is also reversible. The ICT can only be used in lossy compression, since rounding errors are introduced when performing the transformation, which causes a loss of information. The ICT is best suited together with the CDF 9/7 wavelet transform, since it too is irreversible in the same way. In the equations in the subsequent sections the subscripts R, G and B corresponds to the red, green and blue components of the image respectively while Y is the luminance, while Cb and Cr are the blue-difference and red-difference chroma components respectively.

2.4.1 The Reversible Color Transform

The reversible color transform (RCT) is considered reversible because it is an integer based operation, meaning that performing the transform and reverse transform several times on the same image will not incur any additional losses in information. The forward RCT is defined as[16]

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} \frac{R+2G+B}{4} \\ B-G \\ R-G \end{bmatrix} \tag{2.13}$$

While the reverse RCT, which transforms the image back from YCbCr to RGB, is defined as

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} Cr+G \\ Y-\frac{Cb+Cr}{4} \\ Cb+G \end{bmatrix} \tag{2.14}$$

Because the RCT is based on integer arithmetic, and because the division is constant and a power of two, its implementation in hardware is trivial. There is no need to implement floating-point or fixed-point arithmetic to perform the RCT. Although the process is called reversible, some rounding occurs when the luminance component Y is calculated, as it involves a division by 4. Figure 2.18 shows the output of an Matlab implementation of the RCT.

Figure 2.18: Conversion from the RGB color space to YCbCr using the reversible color transform.

2.4.2 The Irreversible Color Transform

The irreversible color transform (ICT) is considered irreversible because its equations is based on fractional numbers as opposed to the reversible color transform. This means that when implemented in hardware, the transform loses precision due to rounding errors caused by imprecise fixed or floating-point notations. The forward ICT is achieved by means of Equation 2.15, which is a six decimal digits of precision variant of the ICT defined in the ITU-T T.871 standard for JPEG images [20].

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.584 & 0.114 \\ -0.16875 & -0.33126 & 0.5 \\ 0.5 & -0.41869 & -0.08131 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.15}$$

While reverse ICT is achieved using the equation

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.0 & 0 & 1.402 \\ 1.0 & -0.34413 & -0.71414 \\ 1.0 & 1.772 & 0 \end{bmatrix} \cdot \begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} \tag{2.16}$$

Figure 2.19 shows ICT performed on a image using a Matlab implementation of Equation 2.15. As can be seen, the resulting YCbCr color space image differs from the one achieved using RCT.

Implementing the ICT is far more costly in terms of hardware resources than the RCT because of the fractional arithmetic needed to perform the transformation. In order to perform the arithmetic needed in Equation 2.15 either fixed-point or floating-point representation has to be implemented. Fixed-point or floating-point arithmetic, in this case multiplication, addition and subtraction also has to be implemented. If the ICT of a red, green and blue pixel is to be performed in a single clock cycle, this correspond to

Figure 2.19: Conversion from the RGB color space to YCbCr using irreversible color transform.

a large amount of hardware resources. Using floating-point notation, it would require 3 conversions of integer to floating-point, 9 floating-point multiplications, 3 floating-point additions and 4 floating-point subtractions. Performing all these operations in a single clock cycle would obviously be require an substantial amount of the resources to implement in hardware. This amount could be reduced by performing the arithmetic sequentially, thereby reusing the multipliers, adders and subtracters needed, but at the cost of processing speed as more clock cycles would be needed to perform the ICT.

To reduce the hardware resources required to perform the ICT, an integer approximation of the transformation coefficients is proposed. The fractions numbers from Equation 2.15 is multiplied by 256 and rounded to nearest integer. Thus, the proposed integer ICT is achieved using the equation

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 76 & 151 & 28 \\ -43 & -85 & 128 \\ 128 & -107 & -21 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{256} \\ \frac{1}{256} \\ \frac{1}{256} \end{bmatrix} \tag{2.17}$$

As the equation shows, the fractional numbers have been replaced with integers, removing the need to implement fixed-point or floating-point number representation and its associated arithmetic operations. The scaling factor of 256 was chosen because its a power of two, meaning that the integer division can be achieved using bit shifting, making the operation trivial. Figure 2.20 shows a side by side comparison of the real number ICT and the integer ICT, both having been implemented in MATLAB. Here, the original image is first transformed using one of the ICT methods, and then transformed back using the inverse ICT as outlined in Equation 2.16. The mean square error (MSE) and peak signal-to-noise ratio are determined by comparing the result of the inverse ICT with the original image.

Figure 2.20: Comparison between real number ICT and integer number ICT.

As Figure 2.20 shows, the fractional number ICT results in a MSE of 0 and a PSNR of infinity, which is to be expected since the implementation was done in MATLAB using double precision floating-point numbers, which only introduce very small rounding errors which are not noticeable in this example. By using the integer ICT described in Equation 2.17, a MSE of 0.348 and PSNR of 52.7 dB is achieved. This demonstrates that the proposed integer ICT only introduces minor errors while removing the need for fixed or floating-point implementation in the color transformation stage of the JPEG2000 pipeline. In JPEG2000, the color transformation stage is also referred to as the intercomponent stage.

2.5 Wavelet transform

The discrete wavelet transform is an essential part of the JPEG2000 image compression pipeline. The wavelet transform is similar to the Fourier transform in that it transforms incoming data to a different representation, however in this case consisting of wavelets. Wavelets are a relatively new concept, with the first wavelet being discovered by Alfréd Haar in 1909.[21] Subsequent research on wavelets led to Ingrid Daubechies work on orthogonal wavelets in 1988 and biorthogonal wavelets in 1992.[22] This led to the development of the Cohen–Daubechies–Feauveau (CDF) family of wavelets, of which the CDF 5/3 and CDF 9/7 wavelets form the basis for lossless and lossy image compression respectively in the JPEG2000 standard.[23] A MATLAB implementation of the CDF 9/7 wavelet transform using the lifting scheme, developed by Pascal Getreuer, is used as a "golden standard" to verify the implementation of the wavelet transform in this project. [24] This MATLAB implementation is also used to generate some of the images used in the subsequent theoretical discussion on the wavelet transform in this section.

The purpose of the wavelet transform is to split the incoming image into multiple frequency bands, called subbands, in a process called decomposition. Each subband consists of either high or low frequency coefficients from the incoming image.

Figure 2.21 illustrates how a 1-dimensional discrete wavelet transform decomposes an image of size N x N into two sub-bands, each with a size N x N/2. In the figure, L represents the low-frequency coefficients and H represents the high-frequency coefficients. The low-frequency coefficients are also referred to as the approximate coefficients or smoothing coefficients, while the high-frequency coefficients are also referred to as the detail coefficients.

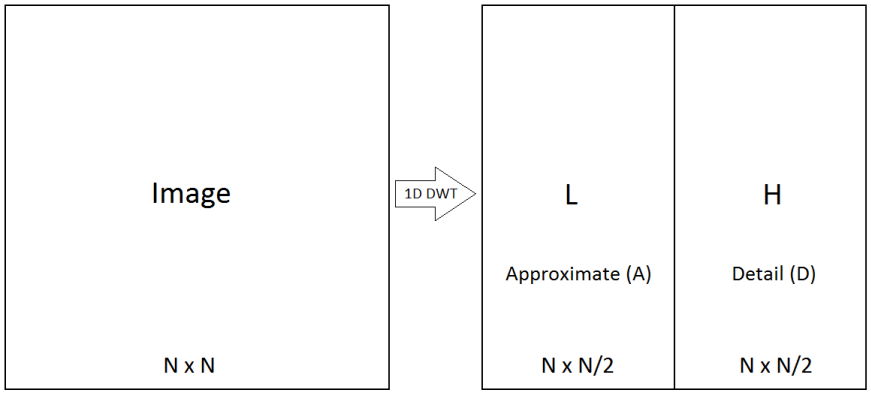

Figure 2.21: The decomposition of a 1D discrete wavelet transform.

The wavelet transformed used in JPEG2000 is 2-dimensional, meaning that it operates in both the x and y plane of an image. While a 1D wavelet transform only processes the rows of an image, a 2D transform also processes the columns. Therefore, to perform a 2D transform, all that is needed is to perform the 1D transform on an the rows of the image, then perform the 1D transform again on columns of the decomposed image. Figure 2.22 illustrates a 2D wavelet transform consisting of two 1D wavelet transform stages.

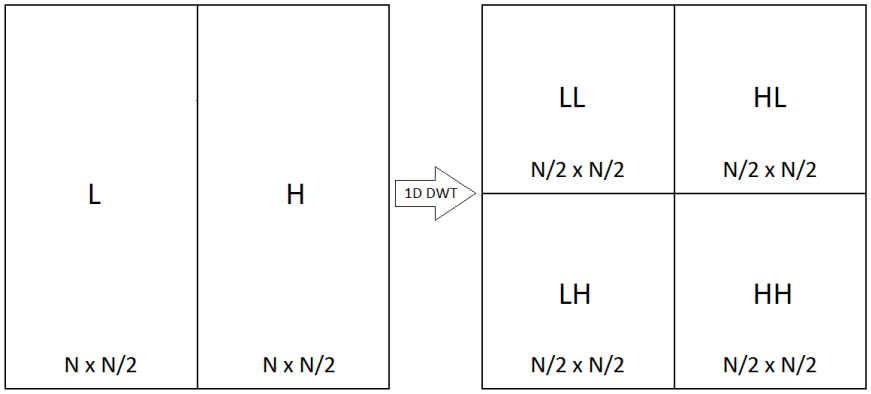

Figure 2.22: The dyadic decomposition of a 1 level 2D discrete wavelet transform.

As Figure 2.22 shows, performing the 2D discrete wavelet transform on an image results in four sub-bands, each with a size N/2 x N/2. This is also known as a 1 level 2D DWT. By performing the DWT several times on the resulting lowest frequency component subband (LL), the image can be decomposed into even more sub-bands, exemplified by Figure 2.23.

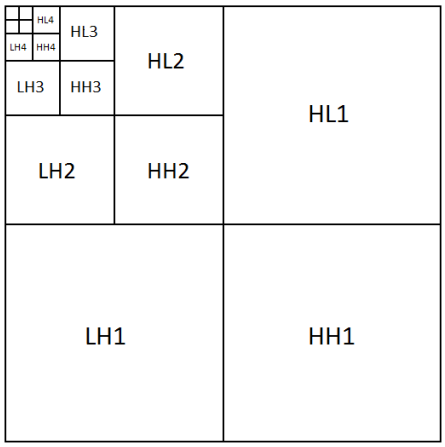

Figure 2.23: The dyadic decomposition of a 5 level 2D discrete wavelet transform.

In Figure 2.23, a 5 level 2D DWT has been performed on an image. This is achieved by first performing a 1 level 2D DWT on the image, then performing a 1 level 2DWT on the resulting LL subband of the decomposition. This process is repeated three more times, which results in the dyadic decomposition illustrated in Figure 2.23. The JPEG2000 standard does not set a specific number on how many decomposition levels which should be used in the wavelet transform stage of the compression pipeline.[16] In general, the wavelet transform decreases the information content, or entropy level, of the image. In other words, the subbands can be more easily compressed in the encoding part of the compression pipeline. The entropy level of an image and its relevance is more thoroughly explored in Section 2.7. Performing multiple levels of the wavelet transform also increases the dynamic range of decomposition coefficients. Table 2.1 shows the relationship between wavelet decomposition levels and the dynamic range when performed on a grayscale version of Lena.

Table 2.1: Comparison between decomposition levels and dynamic range

Level	Dynamic range	Dynamic range level shifted
0	25 to 245	-103 to 117
1	-64 to 484	-208 to 228
2	-238 to 915	-394 to 403
3	-366 to 1819	-803 to 795
4	-688 to 3480	-1632 to 1432
5	-1201 to 6760	-2610 to 2664

The dynamic range of a level shifted image is also shown, meaning that the dynamic range of the original image is shifted before the transformation take place in the same way outlined in Section 2.3.5. As Table 2.1 shows, the dynamic range of the decomposition coefficients quickly grows after each transform. The original image had a dynamic range from 25 to 245, which is typical for a grayscale image, which has a maximum dynamic range of 0 to 255, or 256 intensity levels. This means that each pixel can be represented using 8 bits. After performing a single transformation, the dynamic range has increased to -64 to 484, which means that we need a minimum of 10 bits to store each coefficient. After 5 levels, the dynamic range has increased to -1201 to 6760, meaning that each coefficient now requires 14 bits of storage. This increase in dynamic range is something that has to be taken into consideration when implementing the wavelet transform in hardware. When level shifting is employed, the dynamic range after 5 transforms is -2610 to 2664, meaning that 12 bits is required to store the coefficients, which is 2 bits less than without level shifting. As mentioned, the JPEG2000 standard does not specify a preferred composition level, but a common number given is between 4 and 6 levels.[14] The transform level chosen is transmitted as part of the code stream in the encoding part of the compression pipeline as detailed in Section 2.9.

Figure 2.24 shows the result of a 1D DWT performed on the image of Lena, which decomposes the image into two subbands. The left subband contains the low frequency coefficients of the original image, while the right subband contains the high frequency coefficients. The brightness of the image has been adjusted to clearly show the high-frequency coefficients, as they would otherwise be nearly invisible.

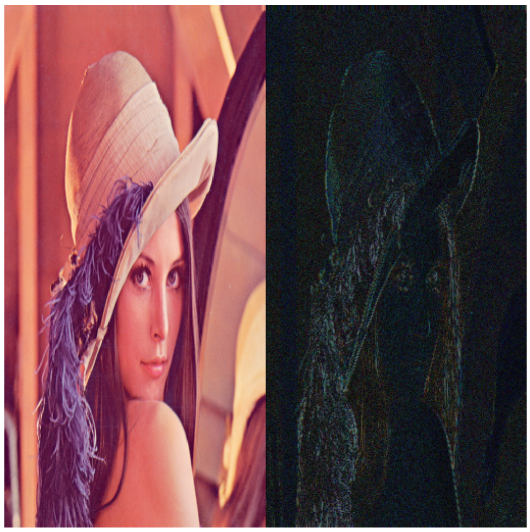

Figure 2.24: An image decomposed into high frequency and low frequency components using the 1D discrete wavelet transform.

As can be seen from Figure 2.24, the low frequency subband appears to be a resized version of the original image, while the high frequency subband appears to be an outline of the original image. This is because a "natural" image, such as Lena, is essentially

a low frequency image. This means that abrupt changes between colors is statistically unlikely. An abrupt change in color would be a high-frequency event, and this is exactly what the high frequency subband shows.

Figure 2.25 show the result of a 1 level 2D DWT on the same image of Lena. Again, the brightness of the image has been adjusted. The image has now been decomposed into four subbands, each with a different frequency range.

Figure 2.25: An image decomposed into high frequency and low frequency components using the 2D discrete wavelet transform.

The wavelet transform can be implemented in hardware in several ways, but the most common methods is either convolution-based or lifting-based, which is the subject of the following sections.

2.5.1 Convolution-based DWT

One way to perform the DWT is through the means of convolution. It is based on Mallat's pyramid algorithm, and works by passing the image through two finite impulse response (FIR) filters placed in parallel, which acts as high-pass and low-pass filters.[25][26]

The result of the transform is a decomposition of the input signal S[n] into two subband coefficients, the detail coefficients D[n] and the approximation coefficients A[n], as expressed by the equations

$$A[n] = \sum_{k=0}^{l-1} g[k]S[2n-k] \tag{2.18}$$

$$D[n] = \sum_{k=0}^{j-1} h[k]S[2n-k] \tag{2.19}$$

Where h(k) and g(k) are the high-pass and low-pass filter coefficients and j and l are the length of the filters respectively. Figure 2.26 shows the basic building blocks that make up a 1D DWT based on Mallat's pyramid algorithm. As can be seen, each sample is sub-sampled by a factor of two.

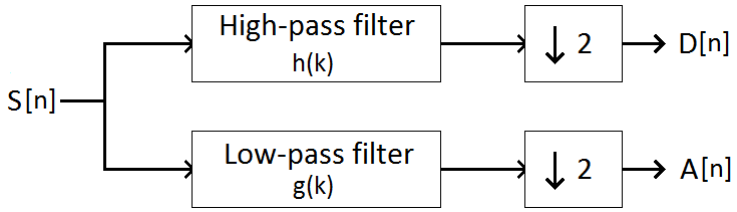

Figure 2.26: A 1D forward wavelet transform based on convolution.

Similarly, the inverse discrete wavelet transform is achieved by first up-sampling the detail and approximate coefficients and filtering them through high-pass and low-pass filters, as shown in Figure 2.27.

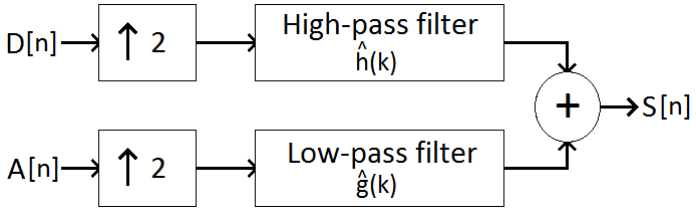

Figure 2.27: A 1D inverse wavelet transform based on convolution.

By cascading the filter bank outlined in Figure 2.26, one can achieve a higher level decomposition of the image. Performing the convolutions necessary to perform the wavelet transform requires a significant amount of resources in terms of hardware and processing time. This is because the convolution process itself requires a relatively large amount of multiplications and additions, with the former being a fairly costly process when it comes to hardware and processing. The process also requires a significant amount of storage. In order to reduce the number of multiplications and additions needed to perform the transformation, a different technique was proposed called the lifting scheme. The lifting scheme is the preferred method to perform the transform in this project, and will be discussed in the next section.

2.5.2 Lifting-based DWT

The lifting scheme, also referred to as the second-generation wavelet transform, was first introduced by Wim Sweldens in 1996.[27] He proposed performing the wavelet transformation through a number of split, predict, update and scaling steps as shown in Figure 2.28.

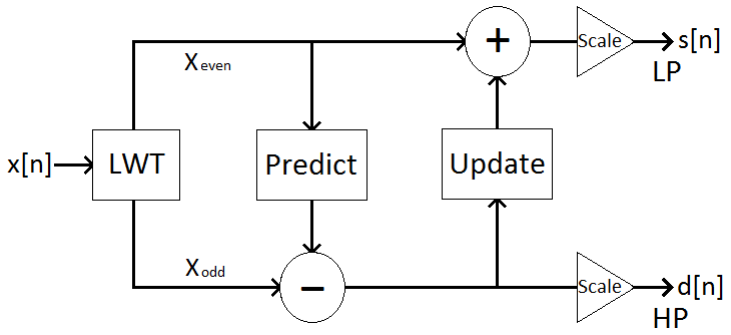

Figure 2.28: A 1D forward wavelet transform based on the lifting scheme.

In Figure 2.28, the Lazy Wavelet Transform block (LWT) splits the incoming samples x[n] into odd and even samples as shown in the equation

$$\begin{aligned} x_{odd} &= x[2n+1] \\ x_{even} &= x[2n] \end{aligned} \tag{2.20}$$

The prediction step computes a prediction for the detail signal samples based on the even samples. This prediction is then subtracted from the odd samples. Following this step is an updating step where the resulting odd predicted samples update the even samples. A final scaling step is used to normalize the resulting samples, resulting in the detail d[n] and approximation s[n] wavelet coefficients.

The inversion of the lifting steps can be achieved by using the same structure but with the sign of the addition and subtraction changed as shown in Figure 2.29.

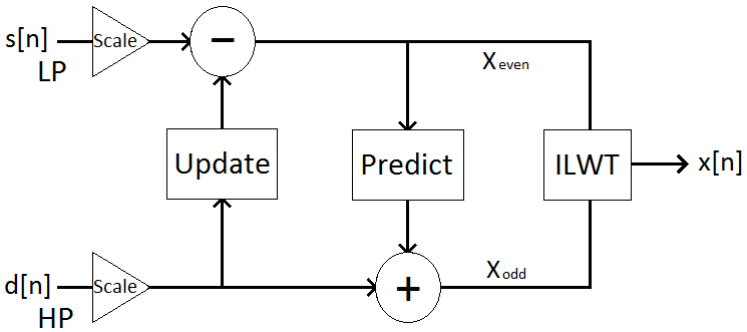

Figure 2.29: A 1D inverse wavelet transform based on the lifting scheme.

An implementation of the wavelet transform using the lifting scheme can have an arbitrary number of predict and update steps depending on the filter coefficients employed in the transform. This is exemplified with the reversible 5/3 and irreversible 9/7 transform, which requires 2 and 4 steps respectively.

2.5.3 Reversible Wavelet Transform

The CDF 5/3 wavelet transform was developed in 1992 in a cooperation between Cohen, Sweldens and Feauveau.[23] A lifting scheme implementation of the wavelet transform based on one prediction, one update and no scaling step was developed in 1998 by Calderbank, Sweldens and Daubechies, where they expressed the transformation lifting scheme using the equations[28]

$$d[n] = x[2n+1] + \left[\alpha\Big(x[2n+2] + x[2n]\Big)\right] \tag{2.21}$$

$$s[n] = x[2n] + \left[\beta\Big(d[n] + d[n-1]\Big) + \frac{1}{2}\right] \tag{2.22}$$

And specified the constants α and β as $\alpha = -\frac{1}{2}$ and $\beta = \frac{1}{4}$. One simplified hardware implementation of this lifting scheme based wavelet transform is illustrated in Figure 2.30. Here, the D blocks illustrate delay blocks which delays the input samples by a predefined number of samples depending on the delay caused by the arithmetic entities in the architecture.

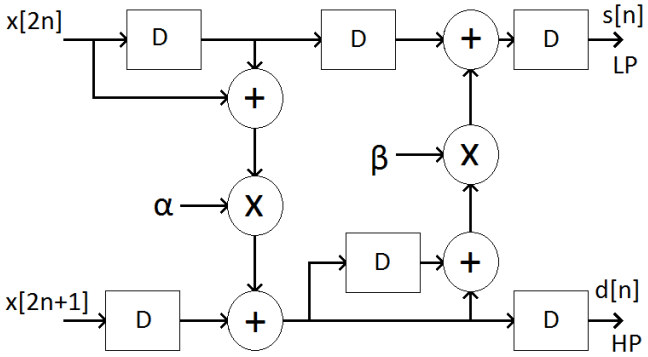

Figure 2.30: 1D discrete wavelet transform lifting scheme block diagram for the 5/3 transform.

The CDF 5/3 wavelet is specified as the preferred wavelet transform when performing lossless image compression in the JPEG2000 standard.[15] One significant advantage of the CDF 5/3 wavelet transform is that because it uses simple fractions it is possible to implement without resorting to real number arithmetic, which requires the fixed of floating point numbers in hardware. This could for instance be achieved by multiplying the α and β constants by a scaling factor, perform the predict and update steps, and then divide the result by the same scaling factor. As long as the factor remains a power of two, the division is trivial to implement in hardware.

2.5.4 Irreversible Wavelet Transform

The CDF 9/7 wavelet transform was also developed in 1992 by Cohen, Daubechies and Feauveau and was described in the same paper as the CDF 5/3 wavelet transform.[23] A lifting implementation of this wavelet transform was proposed by Sweldens and Daubechies in 1998. [29] The lifting implementation uses two prediction, two update and two scaling steps to achieve the wavelet transform, for a total of six steps. The lifting process can be expressed with the equations

$$d_0[n] = x[2n+1] + \alpha\Big(x[2n+2] + x[2n]\Big) \tag{2.23}$$

$$s_0[n] = x[2n] + \beta\Big(d_0[n] + d_0[n-1]\Big) \tag{2.24}$$

$$d_1[n] = d_0[n] + \gamma\Big(s_0[n+1] + s_0[n]\Big) \tag{2.25}$$

$$s_1[n] = s_0[n] + \epsilon\Big(d_1[n] + d_1[n-1]\Big) \tag{2.26}$$

$$s[n] = ks_1[n] \tag{2.27}$$

$$d[n] = \frac{1}{k}d_1[n] \tag{2.28}$$

And with the constants α, β, γ, ϵ and k which can be approximated as $\alpha = -1.586134$, $\beta = -0.052980$, $\gamma = 0.882911$, $\epsilon = 0.443506$ and $k = 1.149604$.[19] It is referred to as irreversible because the constants are irrational numbers and not simple fractions as with the CDF 5/3 transform. This means that some information is lost because of rounding of numbers, hence then name irreversible. Although the process can be reversed with an inverse wavelet transform, the resulting image is not reconstructed perfectly. Figure 2.31 shows a simple hardware implementation of the lifting scheme based CDF 9/7 wavelet transform.

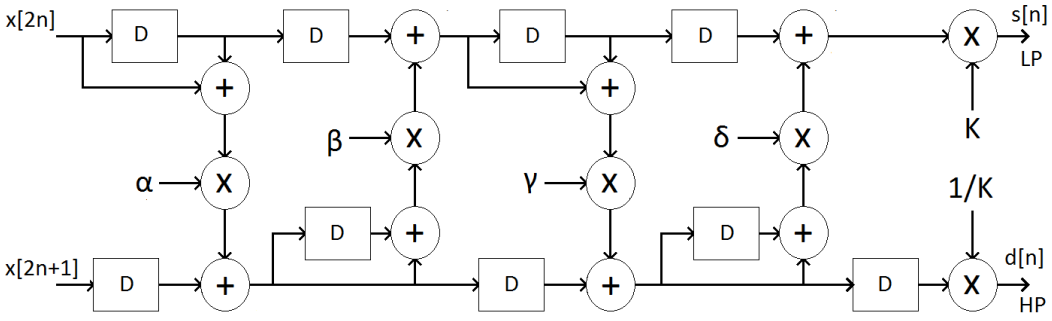

Figure 2.31: 1D discrete wavelet transform lifting scheme block diagram for the CDF 9/7 transform.

The D blocks again acts as delay registers which delays the samples a specified amount in order to pipeline the process. This hardware implementation is very similar to the one shown in Figure 2.30, and it is possible to modify it to perform the CDF 5/3 by simply setting $\gamma = 0$, $\epsilon = 0$, $k = 1$ and $\alpha = -\frac{1}{2}$ and $\beta = \frac{1}{4}$. This means that by implementing the CDF 9/7 in hardware, it is easy to switch over to the CDF 5/3 instead. The CDF 9/7 is the recommended wavelet transform for lossy image compression in the JPEG2000 standard.[15]

Unlike the CDF 5/3 wavelet transform, the coefficients for the CDF 9/7 can not be expressed as simple fractions, meaning that either fixed or floating point numbers have to be implemented to perform the arithmetic of the lifting process. It is however possible to avoid this by multiplying the constants by a scaling factor, rounding them, perform the lifting process and then divide the result by the same scaling factor. As an example, with a scaling factor of 256, the resulting rounded constants would be $\alpha = -406$, $\beta = -14$, $\gamma = 226$, $\epsilon = 114$ and $k = 294$. The resulting wavelet coefficients would then be divided by 256 after the scaling step in the lifting scheme. Since 256 is a power of 2, implementing this division in hardware is trivial. This method removes the need for fixed or floating point numbers, but introduces significant rounding errors, especially when a multilevel wavelet transform is performed. It would also increase the dynamic range of the intermediate samples in the wavelet transform pipeline considerably, meaning that more bits is needed to store the values in the pipeline.

2.5.5 Symmetric signal extension

Symmetric signal extension is used when dealing with the edges of the tiles during the wavelet transform. This is because the lifting scheme algorithm will require samples which are "outside" the tile once it reaches the edge of the tile. By extending the signal symmetrically, distortions around the edge of the image is kept to a minimum. Figure 2.32 illustrates how the samples are extended when at the edge of a tile.

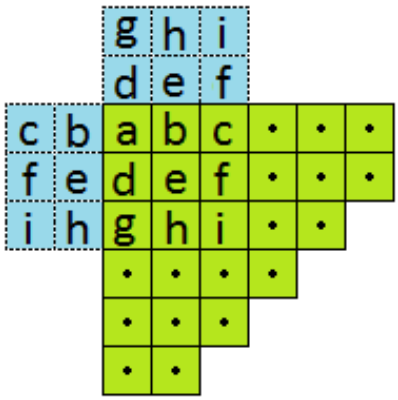

Figure 2.32: Illustration of symmetric signal extension on a tile corner.

A more mathematical representation of symmetric signal extension of a tile row is shown in Figure 2.33, which shows the extension of a N sample tile row.

$$\ldots\ X_3\ X_2\ X_1 \,\big|\, X_0\ X_1\ X_2\ X_3\ \ldots\ X_{N-3}\ X_{N-2}\ X_{N-1} \,\big|\, X_{N-2}\ X_{N-3}\ X_{N-4}\ \ldots$$

Figure 2.33: A more mathematical representation of the signal extension. Here, X represents a pixel in a row or column and the vertical bars represents the image boundaries.

The same symmetric extension also applies to tile columns when transposed. From Figure 2.33, each row and column of a tile component is extended at the start of the of the image and at the end of the image in a process named pre-extension and post-extension respectively. The amount of extension needed depends on the wavelet transform and on the number of lifting stages employed. The 9/7 irreversible wavelet transform needs an extension of two samples at the edges during the transformation.

As mentioned, symmetric signal extension is important during the wavelet transform to avoid distortion of the edges around the image. If the extension is not employed, the resulting coefficients from the wavelet transform will be based on undefined values, namely values "outside" the tile component. Figure 2.34 shows the effect that symmetric signal extension has on an image which have been forward and inverse wavelet transformed using the 9/7 transform.

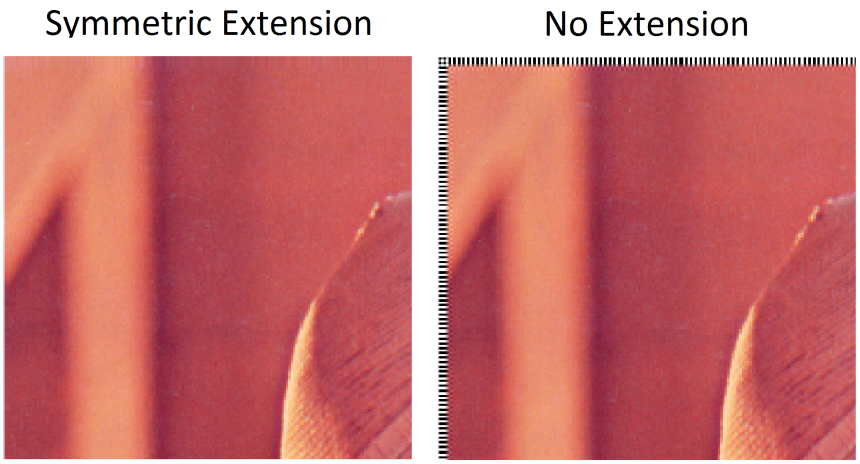

Figure 2.34: A comparison of the DWT with and without symmetric signal extension.

As is apparent in Figure 2.34, a significant border artifact appears around the image when no signal extension is used.

2.6 Rate control

The rate control of the JPEG2000 compression pipeline is what determines the level of compression done to the image, which ultimately also determines the resulting quality of the decompressed image. The rate control determines the resolution of the quantization performed in the quantization step of the compression system. The quantization rate is determined by the scalar quantization stepsizes for the subbands of the decomposed image after the wavelet transform. A larger stepsize results in a higher compression rate, at the cost of image quality. Quantization is further elaborated in Section 2.7. The rate control also influences the encoding part of the compression system, meaning that some parts of the encoded image can be removed in order to reduce the final size of the compressed image, although at the cost of image quality. This is explained more fully in Section 2.8.

2.7 Quantization

Quantization is a process which maps a large set of numbers into a smaller set of numbers. It is an irreversible and inherent non-linear operation, because information about the original signal is lost during the process. By reducing the precision of the original signal through quantization, the amount of information contained in the image is reduced, which makes it more compressible in the encoding part of the compression pipeline. Quantization is the essentially the process which decides if the compression is done lossless or lossy, as information is lost during the quantization process. If the quantization process is skipped, then the compression of the image is can be made lossless if one disregards the losses caused by the irreversible color transform and the irreversible 9/7 wavelet transform and no data is discarded during the encoding process. A simple quantization scheme used in the JPEG2000 standard is uniform scalar quantization with a dead-zone centered about the origin. The standard also offers more sophisticated quantization schemes, such as trellis coded quantization (TCQ) and variable dead-zone quantization, however these will not be covered in this report.

Uniform scalar quantization is performed on the wavelet coefficients resulting from the discrete wavelet transform of the pixel samples from the after the color transform step. Figure 2.35 illustrates the uniform scalar quantizer with a stepsize designated as Δ.

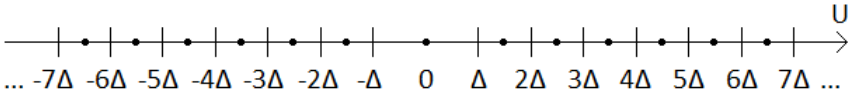

Figure 2.35: Uniform scalar quantizer with a dead-zone around origin and stepsize Δ.

The coefficients which appears on the line are mapped to the nearest point which is a multiple of Δ, rounded down. While the wavelet coefficients are usually fractional numbers, the resulting quantized digits are rounded down to integers. This then leads

to some information being lost, even with a step-size Δ of 1, because the coefficients are rounded down. As can also be seen from Figure 2.35, the coefficients that are within the interval (-Δ, +Δ) are mapped to zero. This is what is known as the dead-zone of the quantizer, which we see is centered around the origin.

Mathematically, uniform scalar quantization can be expressed using the equation

$$V_b(x,y) \;=\; sgn(U_b(x,y))\left\lfloor \frac{|U_b(x,y)|}{\Delta_b} \right\rfloor \tag{2.29}$$

Here, $V_b(x,y)$ represents the quantized wavelet coefficients, $U_b(x,y)$ is the incoming wavelet coefficient, and Δ_b is the step-size. The subscript $_b$ specifies the subband of the decomposed image. This is because the different subbands can be quantized using different stepsizes, which means that several different stepsizes can be employed in the quantization process. The variables x and y denotes the location of the coefficient in the sub-band. The "sgn" function is the sign function, which simply extracts the sign from the real number, and which is defined as

$$sgn(x) \;:=\; \begin{cases} \text{-1} & \text{if x} < 0, \\ 0 & \text{if x} = 0, \\ 1 & \text{if x} > 0 \end{cases} \tag{2.30}$$

The floor function $\lfloor x \rfloor$ maps the real number x to the greatest integer less than or equal to x, while $|x|$ simply returns the magnitude of x. Equation 2.31 shows how the quantization indices are dequantized to produce an approximation of the original signal.

$$\hat{U}_b(x,y) \;=\; (V_b(x,y) + r \cdot sgn(V_b(x,y))) \cdot \Delta_b \tag{2.31}$$

Here, $\hat{U}_b(x,y)$ represents the approximated original signal, and r is a parameter. The r is a user selectable bias parameter within the range $0 \leqslant r < 1$ and is set to a value which provides the best result at the reconstruction of the original image. The parameter is typically set as $r = 0.5$, but it depends on the particular decompression implementation. The parameter value itself is not specified in the JPEG2000 standard. The r parameter can cause the image to gain or lose several dB when dequantization takes place, which means finding an optimal value for it is paramount.

The JPEG2000 standard does place certain limitations on the step-size Δ.[19] This is because it is represented as a two-part quantity (ϵ_b, μ_b) in the codestream later in the encoding part of the standard. The two quantities are used by the decoder to determine the individual step-sizes for the different sub-bands.

The quantities are chosen such that

$$\Delta_b = \left(1 + \frac{\mu_b}{2^{11}}\right) 2^{R_b - \epsilon_b} \tag{2.32}$$

Here, ϵ_b is a 11 bit unsigned integer and μ_b is a 5 bit unsigned integer.[19] R_b is the predicted bit-depth of the wavelet coefficients. For instance, with an expected bit-depth $R_b = 16$, a step-size of $\Delta_b = 1$ would result in $\epsilon_b = 16$ and $\mu_b = 0$. This is important since it limits the number of available stepsizes to be used in the quantization step.

Figure 2.36 illustrates how a set of integers are mapped into quantization indices, and then dequantized back into an approximation of the original integers. U represents the original integers, V is the quantization indices and $\hat{U}$ is the reconstructed integers.

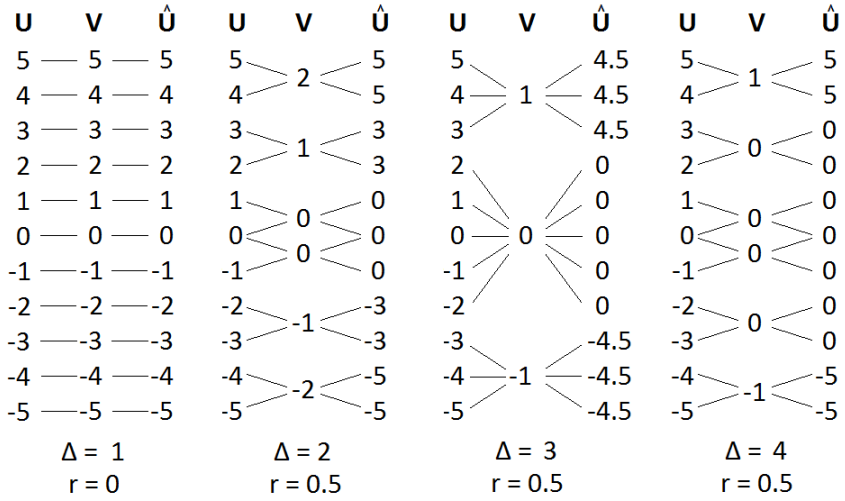

Figure 2.36: Uniform scalar quantization of integers with $\Delta = 1$, $\Delta = 2$, $\Delta = 3$ and $\Delta = 4$.

As Figure 2.36 illustrates, the quantization process introduces quantization errors which are dependent on the quantization step-size. The Mean Square Quantization Error (MSQE) of the quantizer can be determined using the equation

$$MSQE = \sum_{i=1}^{k} \int_{t_{i-1}}^{t_i} (x - q_i)^2 p(x) dx \tag{2.33}$$

Here, q_i is the quantized value of an input value x, t_{i-1} and t_i is the lower and upper threshold for the input values respectively, k is the level of quantization and p(x) is the probability density function for x. For the examples in Figure 2.36, we assume an uniform distribution for the input values. This results in a pdf(x) = 0.1 for $-5 \leqslant x \leqslant 5$. This results is a MSQE of 0, 38, 68 and 78 for the four examples. Although it is a small sample size, it is easy to see how the quantization error grows as the step-size increases.

The main purpose of performing quantization is to reduce the overall entropy in the wavelet coefficients. The entropy is a measurement of the amount of information contained in the image, and is defined by the equation

$$E = -\sum_{k=0}^{M-1} p_k log_2(p_k) \tag{2.34}$$

Here, M is the intensity levels of the pixels in the image, which is 256 for an 8-bit grayscale image, while p_k is the probability if each intensity level occurring. The entropy gives a crude estimate on the number of bits per pixel (bpp) required to encode an image without distortion. Although probably not very accurate in this instance, it nevertheless puts a number on the efficiency of the quantization process. Figure 2.37 shows the entropy of two images.

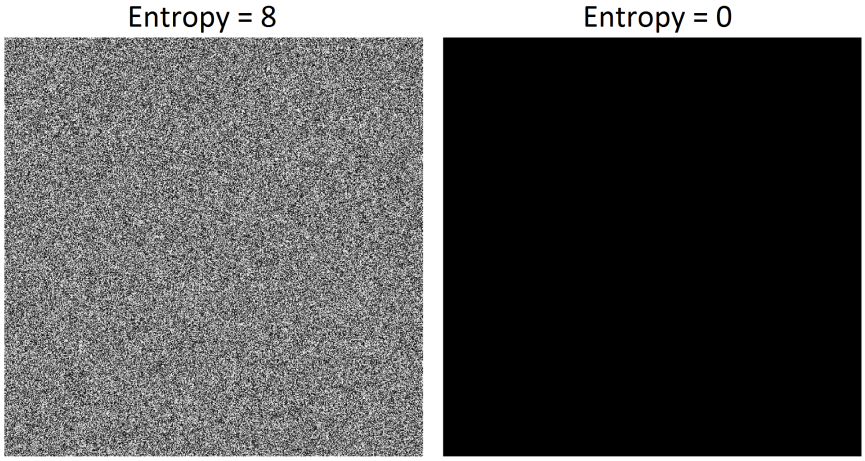

Figure 2.37: The entropy difference in a random image and uniform image.

As Figure 2.37 shows, the image on the left, which is a image with random intensity levels, has an entropy of 8, while the black image has an entropy level of 0. What this illustrates is that the random image is incompressible, with each pixel requiring 8 bits per pixel, which is equivalent to the number of bits required to represent the intensity levels of the image. With an entropy level of 0, very little data is needed to represent the whole image.

Figure 2.38 shows the entropy of an image and its wavelet coefficients. As can be seen, the wavelet transformation has significantly reduced the entropy in the image, meaning that theoretically far fewer bpp is needed to encode the image than for the untransformed image. This makes it clear why the wavelet transform is performed in the first place, as lossless compression skips the quantization step and simply encodes the wavelet coefficients directly. Performing the wavelet transform multiple times on the image reduces the entropy level further.

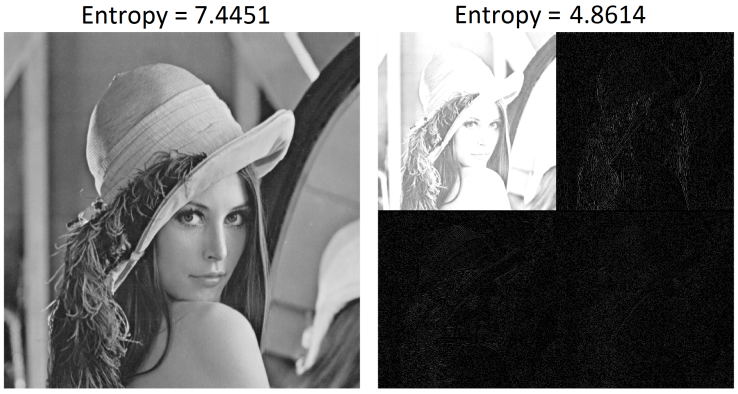

Figure 2.38: The entropy difference of an image and its 1 level DWT decomposition.

The quantization process reduces the entropy level of the wavelet decomposition further, but at the cost of the precision of the wavelet coefficients. The reduction of the entropy depends on the chosen stepsize Δ, with a higher stepsize resulting in a bigger reduction in entropy. Figure 2.39 shows scalar quantization with various step-sizes performed on the wavelet coefficients of an image.

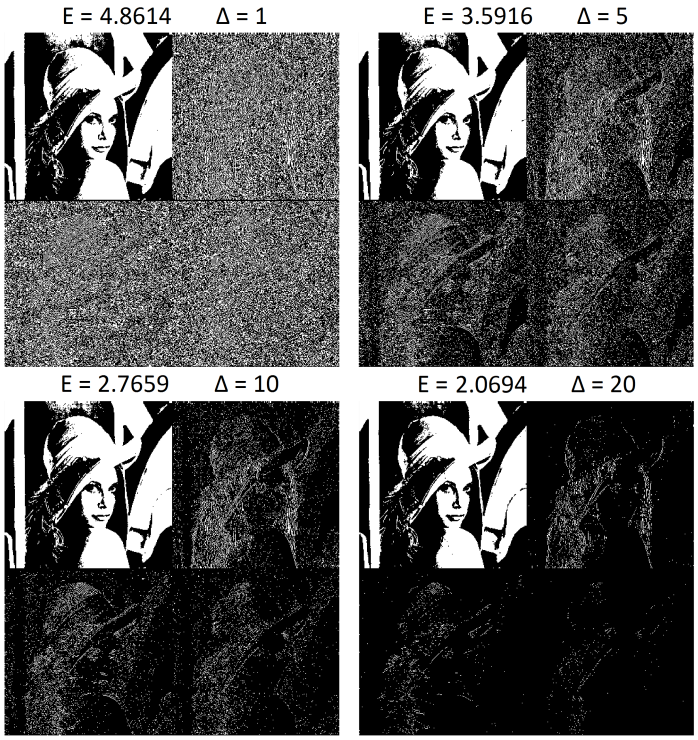

Figure 2.39: Scalar quantization performed on a 1L 2D DWT with different stepsizes and comparing entropy levels.

The image in Figure 2.39 has been enhanced to more clearly show the effect of the quantization process. The image, and most of the subsequent images were generated by implementing Equation 2.29 and Equation 2.31 in MATLAB. As is seen, the stepsize is uniform for all the sub-bands in a decomposition, and a bigger stepsize causes a significant reduction in the entropy of the wavelet coefficients, which translates to a higher compression ratio during encoding. However, as mentioned quantization will cause a distortion in the image during reconstruction, and a bigger step-size increases the level of distortion. Figure 2.40 shows the reconstructed image from the quantized wavelet decompositions shown in Figure 2.39.

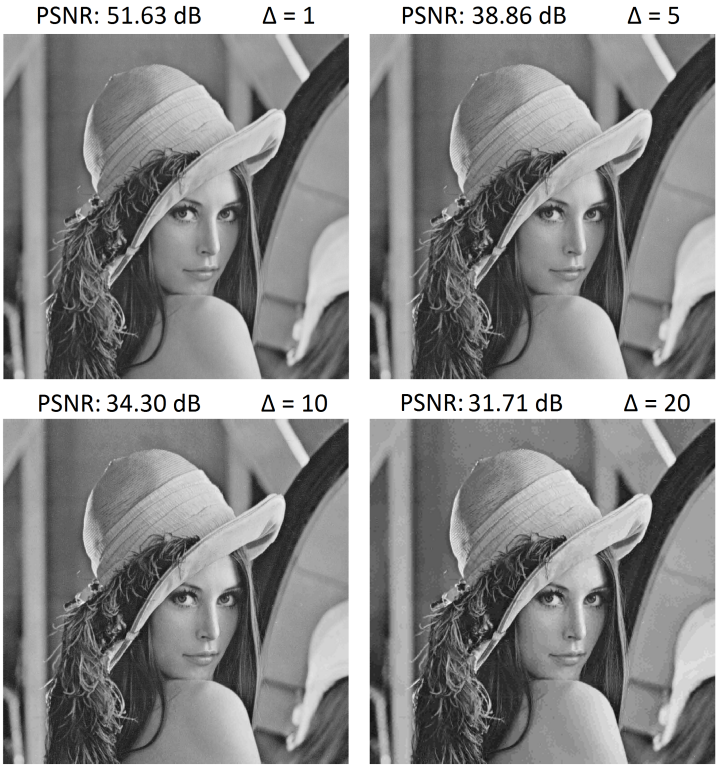

Figure 2.40: The resulting image after dequantization and inverse wavelet transform.

As Figure 2.40 shows, the PSNR of the reconstructed image compared to the original decreases as the step-size increases. The images have been reconstructed with $r = 0.5$. Once the stepsize reaches $\Delta = 20$, the distortion caused by quantization begins to become visible. One of the features of the JPEG2000 standard is that it allows the different subbands of the wavelet decomposition to be quantized with different stepsizes. This means that the LL band can be quantized with a small stepsize, while the HL, LH and HH bands can be quantized with a larger stepsize. Essentially, this means that the more important subbands, the low frequency subbands, can be quantized with a higher fidelity, leading to a higher quality reconstructed image while still reducing the entropy of the wavelet coefficients.

Figure 2.41 shows the wavelet decomposition subbands being quantized with different stepsizes and then dequantized and reconstructed.

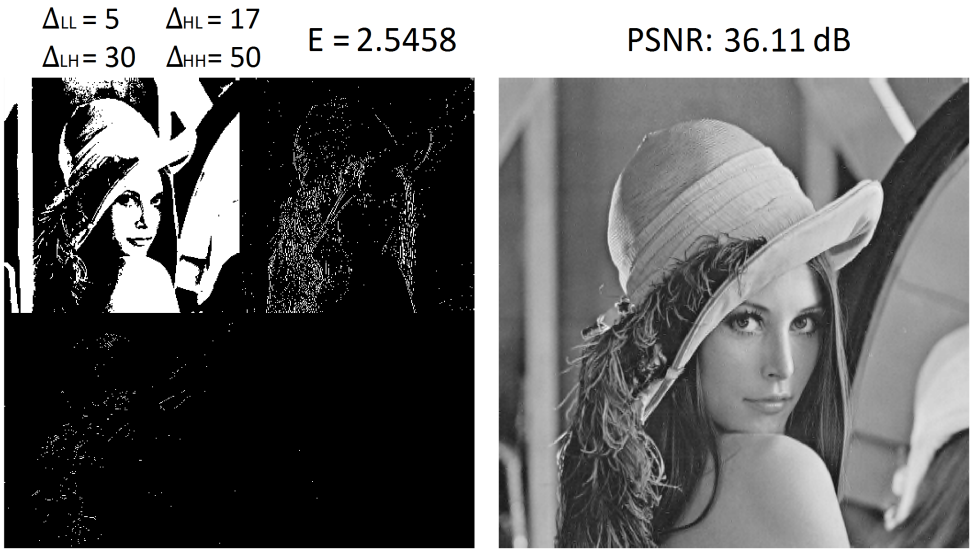

Figure 2.41: Quantization and reconstruction using different stepsizes for the subbands.

As Figure 2.41 shows, by quantizing the LL subband with a lower stepsize and quantizing the other subbands with a higher stepsize, the entropy level is slightly lower compared to when all the subbands are quantized with a fixed stepsize of $\Delta = 10$. As can be seen in Figure 2.40, a constant stepsize of $\Delta = 10$ resulted in a reconstructed image with a PSNR = 34.30 dB. The reconstructed image with varying stepsizes results in a PSNR = 36.11 dB, which is close to a 2 dB increase in the PSNR. The stepsizes here are chosen relatively arbitrarily, and a different picture might yield a different PSNR for the same stepsizes, but it illustrates the merit of separate quantization stepsizes for each sub band. The reconstruction was done with $r = 0.5$. By changing $r = 0.15$, the PSNR of the reconstructed image in Figure 2.41 increases to 37.46 dB. This section has explained how quantization is performed on a single-component grayscale image. If the image has more components, such as for a RGB image, the quantization is performed on each component separately.

2.8 Tier 1 Encoder

After the quantization is complete the quantized wavelet coefficients are then encoded using a bit plane coder (BPC) and a binary arithmetic coder (BAC). For JPEG2000 the BPC chosen is the Embedded Block Coding with Optimized Truncation (EBCOT) coder, which was developed by David Taubman in 2000 and later adopted for the JPEG2000 standard.[30] The BAC used is the Matrix Quantizer coder (MQ), which is a multiplication free, low complexity arithmetic encoder also adopted for the JPEG2000 standard.[31]

The purpose of encoding the image is to reduce the amount of data needed to store the information of the quantized wavelet coefficients even further. Before encoding takes place, the quantized subbands are divided into precincts and code blocks. Figure 2.42 shows how a wavelet decomposition is divided into precincts and code blocks.

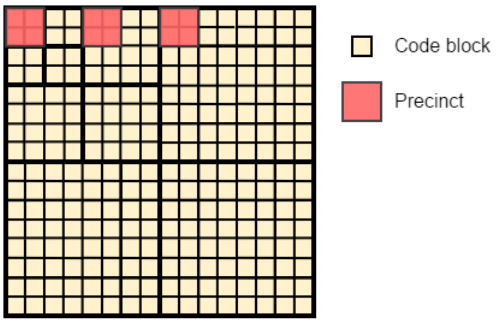

Figure 2.42: The division of a 3 level wavelet decomposition into precincts and code blocks.

The size of the code block has to be a power of two, and also has to be smaller than the size of the precinct. The precincts must also stay within a sub-band, which means that a single precinct can not be present in two sub-bands. The importance of the code blocks and precincts will be further elaborated in the subsequent sections. Figure 2.43 shows the building blocks which makes up the Tier 1 Encoder for JPEG2000. A code block is passed to the EBCOT encoder which generates a symbol and context label that is then transmitted to the MQ coder. The result form the MQ coder is an encoded bitstream which is then packetized by the Tier 2 Encoder as detailed in Section 2.9.

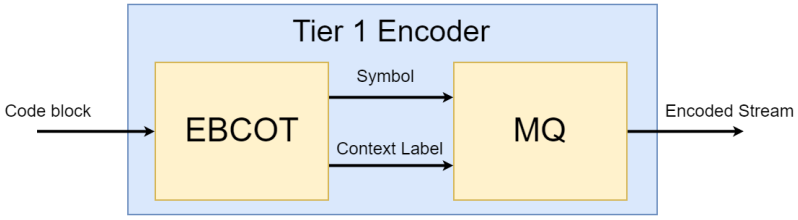

Figure 2.43: Overview of the Tier 1 Encoder containing the EBCOT encoder and MQ encoder.

2.8.1 Embedded Block Coding with Optimized Truncation

The Embedded Block Coding with Optimized Truncation (EBCOT) coder encodes each code block within a precinct independently. As mentioned, it is a bit plane coder (BPC), which means that it operates on the bit planes of the code blocks. Each bit plane is encoded through three coding passes, resulting in an embedded code for each code block which consists of multiple coding passes. To illustrate the concept of bit planes, Figure 2.44 shows the bit planes of a grayscale version of Lena with a bit depth of 8 bits.

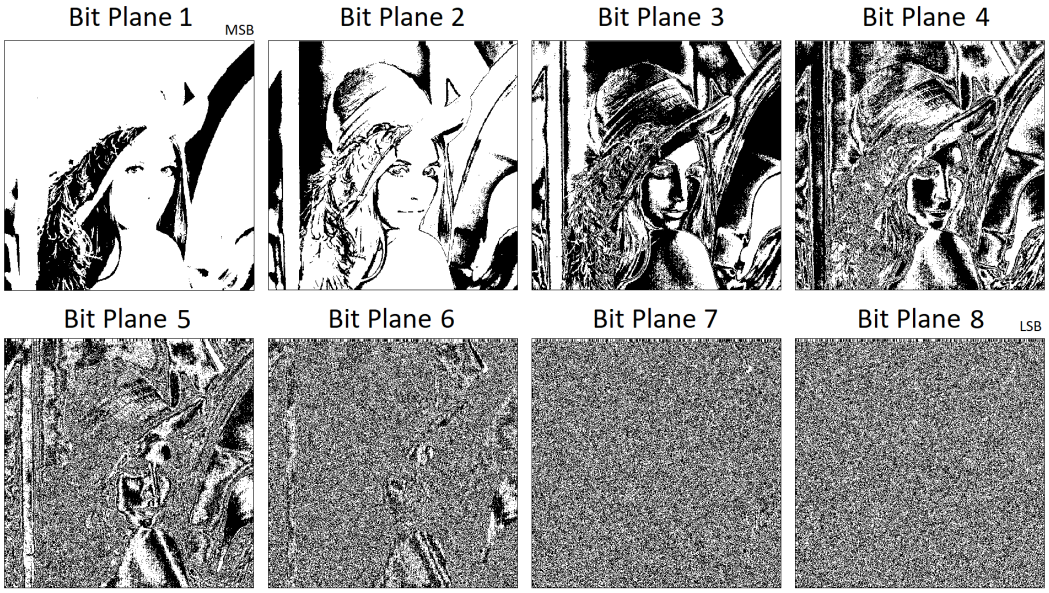

Figure 2.44: A grayscale 8 bit version of Lena split into its constituent bit planes.

From Figure 2.44, Bit Plane 1 is created from the most significant bits (MSBs) of each pixel in the image, which results in a bit plane with the same dimensions as the original image but with each pixel only having a value of either 0 or 1. The same process is repeated for every bit used to represent the pixel, resulting in the final Bit Plane 8, which forms a bit plane consisting of the least significant bits (LSBs) of the image. The same process is used to divide the code blocks into bit planes, with the number of bit planes depending on the number of bits used to represent the samples in the code block. The bit planes are then processed starting with the most significant bit plane and then the second most significant bit plane. This continues until the least significant bit plane is reached. Figure 2.45 shows how samples within a code block are used to construct a bit plane.

A code block of size 8x8 consists of 64 wavelet coefficient samples. The number of bit planes formed by the code block depends on the minimum number of bits needed to represent the samples. If the samples can be represented using 12 bits the result is 12 bit planes each with a size of 8x8 bits. Each bit plane is further divided into

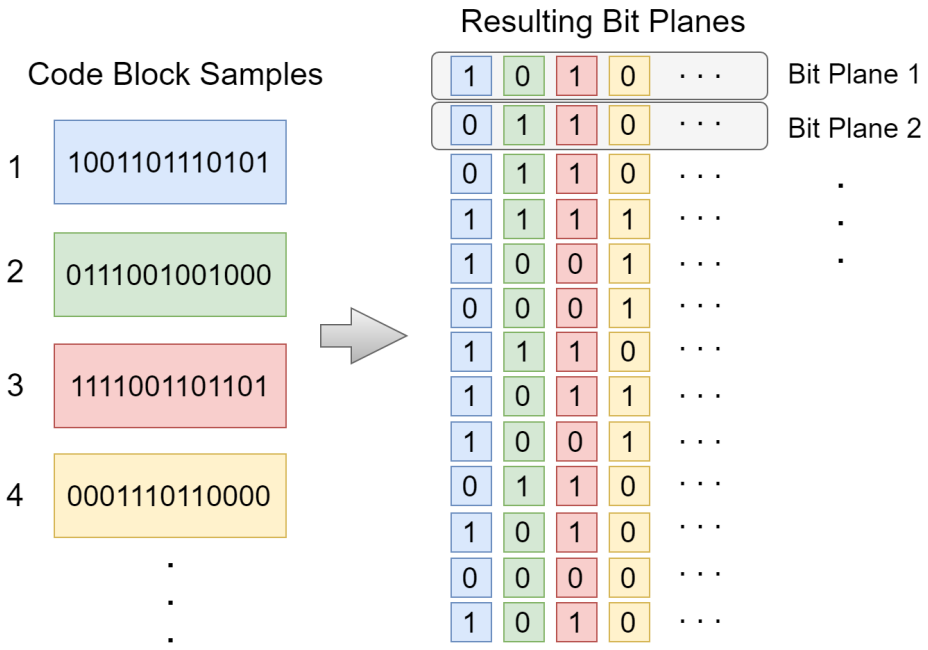

Figure 2.45: Structuring of bit planes from code blocks.

vertical "strips" of four bits each, as shown on the left in Figure 2.46. The numbers represents the order in which the bits are processed. This is not to say that the four bits are processed at once, but simply in what order the bits are processed. In fact, each bit is processed using information about its 8 neighbouring bits, which forms a sliding window as illustrated on the right in Figure 2.46.

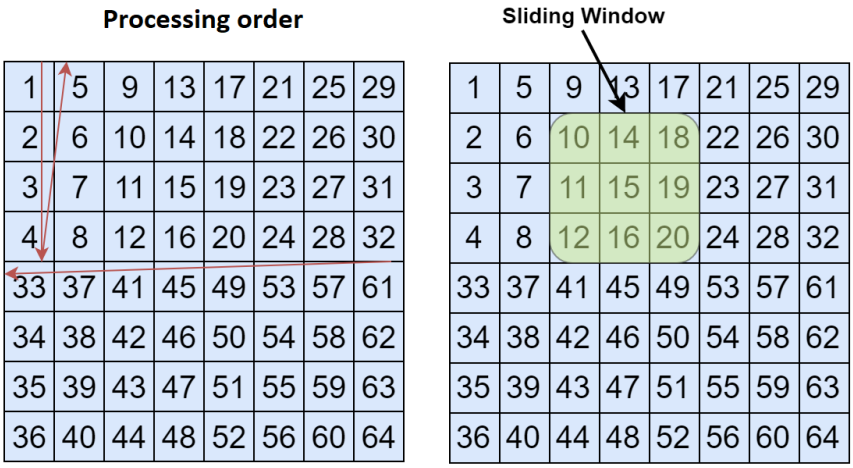

Figure 2.46: Wavelet coefficient sample processing order of a 8x8 code block and the sliding window operation.

The bits obtained from the sliding window is organized as shown in Figure 2.47, with each position in the window labeled from D0 to D8 and D4 being the bit which is currently being processed.

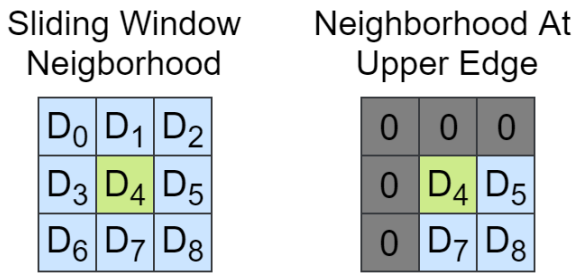

Figure 2.47: The neighborhood of samples formed by the sliding window. Samples outside the code block is set to 0.

At the edges and corners of the bit plane, parts of the sliding window moves outside of the bit plane where there are no valid bits. When this happens, those bits are simply treated as if they are zero. From the 3x3 bit array formed by the sliding window, the value of the neighbouring bits are used to code the middle bit. This is performed during three distinct coding passes performed on each bit plane. One pass refers to when the sliding window has processed all the bits in the bit plane. The three coding passes are referred to as the Significance Pass (SP), the Magnitude Refinement Pass (MRP) and the Cleanup Pass (CP). During each pass only part of the bit plane is coded, with each bit only coded once during the three coding passes. To summarize, the bit planes of a code block is coded using the three coding passes once for each bit plane, except for the most significant bit plane, which is only coded with the Cleanup Pass. This is because the Significance Pass and Magnitude Refinement Pass relies in information from previous bit planes. For the other planes, the order of the coding passes are first SP, MRP and lastly CP.

The 3x3 bit pattern formed by the sliding window forms what is called a context. A total of 19 contexts is used to describe the different bit patterns, with each context belonging to a particular coding method. The definition of the contexts for some of the methods also varies depending on which subband the code block belongs to. The context tables for the various coding primitives is supplied by the JPEG2000 standard.[19] For the three coding passes, each bit in the bit plane can be coded using four different coding primitives: Zero Coding (ZC), Sign Coding (SC), Magnitude Refinement Coding (MRC) or Run Length Coding (RLC). The four different coding primitives are used during one or more of three coding passes, depending on the current coding pass and the context of the sliding window. The three coding passes perform the following:

1. **Significance Pass (SP)** - The Significance Pass encodes all insignificant bits that have one or more significant neighbour in the sliding window.

2. **Magnitude Refinement Pass (MRP)** - The Magnitude Refinement Pass encodes the bits which became significant in the previous bitplane.

3. **Cleanup Pass(CP)** - The Cleanup Pass is the last pass and is used to encode the bits that were not encoded during the other passes. The Cleanup Pass is always the first pass in a new code block.

The resulting encoded symbols and the context formed by the sliding window is then transmitted to the MQ-encoder. The EBCOT coding scheme requires a significant amount of memory as information about the processed bit planes must be stored between the coding passes because coding primitives such as the MRP requires that the significant bits of the previous bit planes are known. It also requires significant processing times, as each bit plane except the first is encoded using three coding passes.

2.8.2 MQ-encoder

The MQ-encoder is an arithmetic encoder which takes the symbol and context label generated by the EBCOT coder and produces an encoded bitstream which is then transmitted to the Tier 2 Encoder of the compression system. The basis of binary arithmetic coding in the MQ-encoder is the recursive probability interval subdivision associated with Elias coding. Using the incoming context and symbol, the encoder relies on a precomputed probability lookup table to establish a more probable symbol (MPS) and less probable symbol (LPS). The precomputed probability lookup table is supplied with the JPEG2000 standard.[19]. The MPS and LPS are then used to create a special partitioned interval denoted as A. The interval A is represented using a fixed precision format and is set to be within the range $0.75 \leq A \leq 1.5$. The incoming symbols and contexts are then used to partition the interval A which results in an partition which can be decoded to reproduce the incoming symbols and contexts. The bits needed to store this partition is less than the incoming symbol and contexts, which is how the information is compressed. Figure 2.48 shows a simplified overview of the MQ encoder architecture.[16]

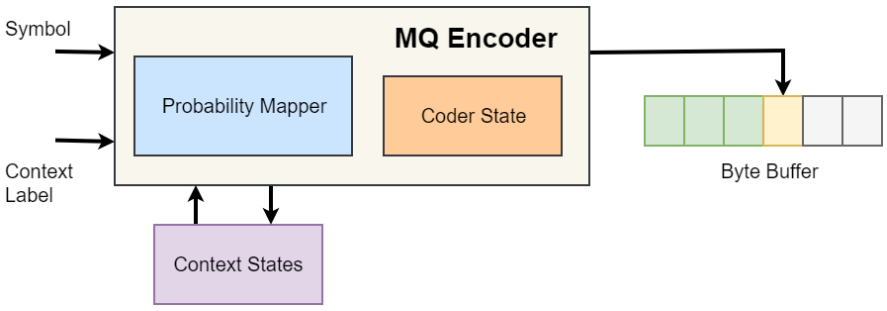

Figure 2.48: An overview of the MQ encoding process.

From the architecture in Figure 2.48, the Context States block contains the lookup table with probabilities which are then mapped to each incoming context. After the arithmetic encoding process the encoded intervals are stored in the byte buffer, which is flushed once all the bit planes in a code block has been processed. The content of the byte buffer is then transmitted to the Tier 2 Encoder.

2.9 Tier 2 Encoder

The Tier 2 Encoder packetizes the resulting encoded samples from the Tier 1 Encoder into properly formatted JPEG2000 packets. This process also packages information about the color transform used, the type of wavelet transform, the stepsize for each subband utilized in the quantization step and all other information relevant to decompressing the compressed image. This information is vital for the JPEG2000 decompressor for it to be able to decode the encoded image properly.

2.9.1 Packetization

The packetization process ensures a universal format for the JPEG2000 compressed images, which makes it possible for any JPEG2000 decompressor to properly decompress a compressed image from any JPEG2000 compression system given that they follow the strict guidelines for how a compressed image is packetized. The JPEG2000 code stream structure is divided into segments as illustrated in Figure 2.49.[19]

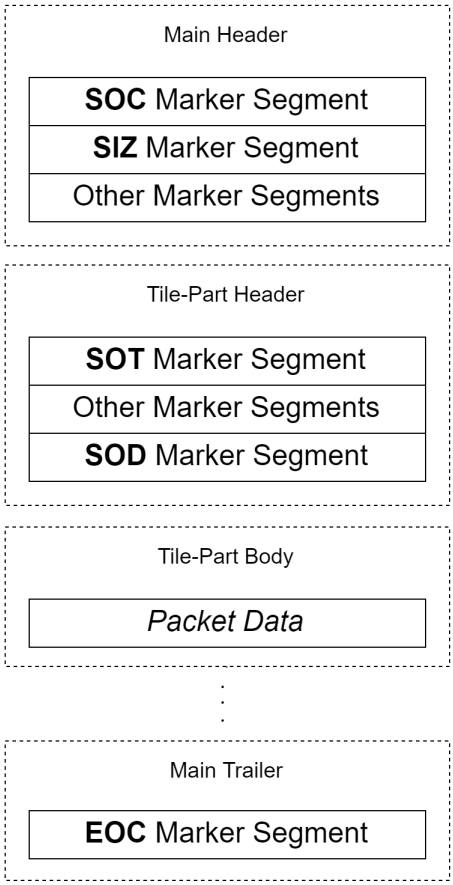

Figure 2.49: The JPEG2000 code stream structure.

Each of the segments consists of a 16-bit type description, a 16-bit length specifier and a parameter with a length defined by the length specifier. This is shown in Figure 2.50.

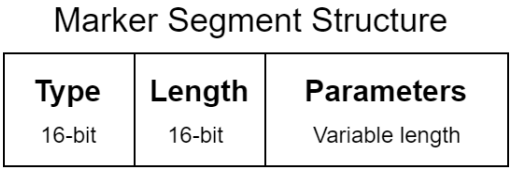

Figure 2.50: The marker segment structure.

The type description in the segment describes the content of the segment, and the most common segments used in the JPEG2000 code stream structure is shown in Table 2.2.[19] The actual encoded bitstream from the Tier 1 Encoder is placed right after the **SOD** marker segment, and is terminated by the **EOC** marker segment. The rest of the segments describe characteristics about the compressed image.

Table 2.2: The the most commonly used marker segments.

Type	**Description**
Start of code stream (**SOC**)	The first marker segment in a code stream. Used to signal the start of a code stream.
End of code stream (**EOC**)	The last marker segment in a code stream. Used to signal the end of a code stream.
Start of tile-part (**SOT**)	Used to indicate the start of a tile-part header. Must always appear first in a tile-part header.
Start of data (**SOD**)	Used to indicate the end of the tile-part header and also indicate the beginning of the tile body.
Image and tile size (**SIZ**)	Conveys the primary image characteristics such as image/tile size, number of components, bit depth of sample values, and other tiling parameters. Must follow immediately after **SOD**.
Coding style default (**COD**)	Conveys the default coding parameters such as color transformation, wavelet transform and encoding parameters.
Coding style component (**COC**)	Conveys coding parameters for a single component only.
Quantization default (**QCD**)	Conveys default quantization parameters such as quantization type used and step-sizes.
Quantization component (**QCC**)	Conveys quantization parameters such as quantization type used and step-sizes for a single component.
Region of interest (**RGN**)	Conveys region of interest coding parameters.

After the JPEG2000 code stream structure consisting of marker segments has been constructed, a final step wraps the code stream into the final JPEG2000 file format structure, as shown in Figure 2.51. [19]

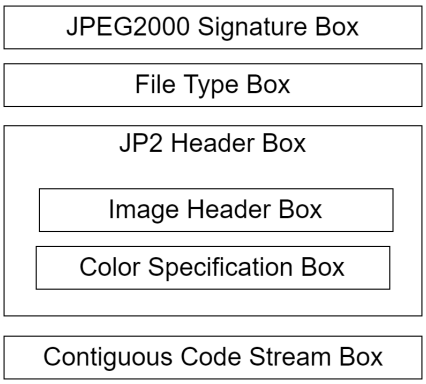

Figure 2.51: The JPEG2000 file format structure.

The file format structure consists of boxes, which like the marker segments contains information about the image and also the code stream structure itself. The box structure is shown in Figure 2.52.

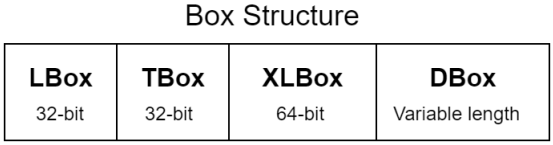

Figure 2.52: The box segment structure.

The **LBox** field specifies the total length of the box in bytes. The **TBox** field specifies the type of box. The **XLBox** field is only used if the length of the box is larger than specified by the **LBox** field. The last field **DBox** contains the data specific to the **TBox** box type. The different box types used are shown in Table 2.3.[19]

Table 2.3: The different box segments.

Type	**Description**
JPEG2000 Signature	Appears first in the file and marks the file as being the JP2 format.
File Type	Defines the version of the JP2 format.
JP2 Header	Provides information about the image besides the coded image itself. This is defined as a "superbox".
Image Header	Defines the size and other characteristics of the image.
Color Specification	Defines the color space which the image belongs to.
Contiguous Code Stream	Contains the code stream.

2.10 Half-precision floating-point

This section will cover the basic theory behind the implementation of the half-precision floating-point format, which is needed in order to perform arithmetic operations on fractional numbers. This is needed for both the wavelet transform and scalar quantization performed in the JPEG2000 image compression system.

2.10.1 Floating-point format

In order to use fractional values in the wavelet transform operation, half-precision floating-point arithmetic is used. The half-precision floating-point representation employed is modeled after the IEEE 754 standard[32], however with some exceptions. The standard defines certain special values which can occur when performing floating-point arithmetic, such as "Infinity" and "NaN", which occurs when dividing by zero or dividing zero by zero. Instead of returning "Infinity" or "NaN", the arithmetic functions in this implementation returns the highest possible integer value for a half-precision floating-point which is 65504 in place for "Infinity", and 0 in place for "NaN". In addition, the strict rounding definitions specified by the IEEE standard is not enforced. Figure 2.53 shows the format for the half-precision floating-point standard.

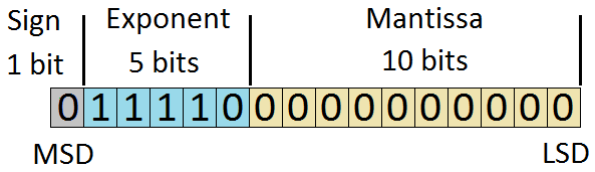

Figure 2.53: The half-precision floating-point format.

Figure 2.54 illustrates the conversion from the floating point to the decimal system. Note the explicit bit and the exponent bias which is equal to 15 being subtracted from the exponent.

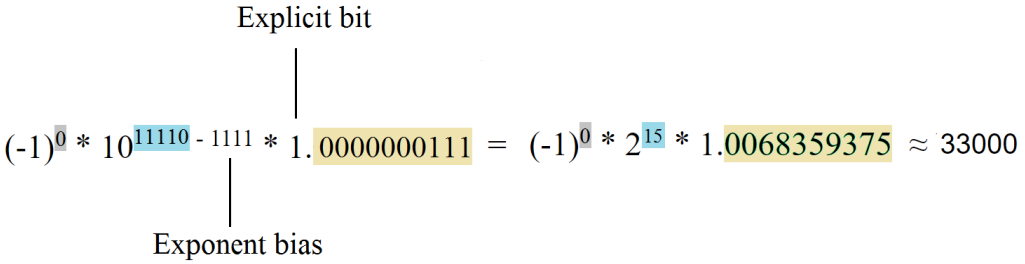

Figure 2.54: Conversion between the half-precision floating-point and the decimal system.

Opting for the half-precision floating-point representation instead of the more popular single or double precision representation has both advantages and disadvantages.

The primary advantage is that half-precision only requires 16 bits to represent the format, while single and double precision requires 32 and 64 bits respectively. The fewer number of bits required, the less hardware resources is needed to perform the floating-point arithmetic. The disadvantage is a loss in precision when using less bits to represent the floating-point number. In addition, the maximum and minimum value that can be represented also changes. For half-precision, the maximum and minimum representable numbers are ±65 504, while its around $\pm\ 3.4 \times 10^{38}$ for single precision and ± 1.8 x 10^{308} for double precision when not accounting for ± infinity.

The main reason for using the half-precision floating-point format to represent fractional numbers in this project is because it makes it possible to implement most of the floating-point arithmetic as purely combinational circuits without requiring too much hardware resources. This means that the arithmetic can be performed in a single clock cycle, rather than using several clock cycles. The downside of using the half-precision floating-point format as opposed to single-precision or double-precision is, as the name implies, a loss in precision. In other words, the end results become more inaccurate because some precision is lost when performing floating-point arithmetic due to rounding errors.

It should be noted that the amount of resources and the timing delay introduced by the floating-point arithmetic depends heavily on which operation that is performed. For instance, performing division on floating-point numbers is far more expensive in terms of resources and time compared to multiplication, addition or subtraction. This is especially true when performing the operation in a single clock cycle. The delay introduced by the combinational circuits is manageable however due to the relatively low clock speed of 50 MHz. If the clock speed was to be increased, it would most likely be better to implement the floating point arithmetic as multi-clock cycle operations. The next sections discuss the basic operation behind floating-point addition, subtraction, multiplication and division. As mentioned, the floating point arithmetic implemented in this project is only loosely based upon the IEEE 754 standard, which makes the arithmetic less complex. This is especially true when the strict rounding operation specified in the IEEE standard is not followed completely.

2.10.2 Floating-point conversion

Converting from normal binary numerical representation to floating-point and vice versa is an important operation in the compression system. This is because some modules operates on normal binary while others are dependent on the floating point format. A 16-bit signed binary integer can be converted to a half-precision floating-point format as shown in Figure 2.53 by performing the following steps in order:

1. If the input is zero, return zero.
2. If input is negative, invert the input and set sign bit to 1, else set sign bit to 0.
3. Set the exponent to log_2 of the input and add the exp bias.

4. If the input is between 1 and $2^{11} - 1$, set mantissa to input - $2^{exponent-exp\ bias}$, and left-shift the mantissa depending on the exponent.
5. If the input is between 2^{11} and $2^{12} - 1$ then subtract 2^{11} from the input, right shift it once and set the result as the mantissa.
6. If the input is between 2^{12} and $2^{13} - 1$ then subtract 2^{12} from the input, right shift it twice and set the result as the mantissa.
7. If the input is between 2^{13} and $2^{14} - 1$ then subtract 2^{13} from the input, right shift it three times and set the result as the mantissa.
8. If the input is between 2^{14} and $2^{15} - 1$ then subtract 2^{14} from the input, right shift it four times and set the result as the mantissa.
9. If the input is between 2^{15} and $2^{16} - 1$ then subtract 2^{15} from the input, right shift it five times and set the result as the mantissa.
10. If the input is higher, set all the bits in the mantissa to 1.
11. Return the sign bit, exponent and mantissa as a 16-bit binary integer.

Where the exp bias for the half-precision floating-point format is 15. The reason for checking the range of the input is because of the limited precision of the format. Lower numbers can be represented more accurately, while higher numbers are lose some of their least significant digits in the process. In order to convert a number back from half precision floating point to signed binary, the following procedure can be performed in order:

1. Separate the sign bit, exponent and mantissa.
2. If the exponent is lower than 14, return zero.
3. If the exponent is 14, return 1.
4. If the exponent is between 15 and 25, right shift the mantissa (25-exponent) number of times and add $2^{exponent-exp\ bias}$.
5. If the exponent is between 26 and 29, right shift the mantissa (exponent-25) number of times and add $2^{exponent-exp\ bias}$.
6. For any higher exponent, set output to $2^{15} - 1$.
7. If the sign bit is 1, invert output.
8. Return output.

Both the conversion to and from floating point is implemented in VHDL, as is shown in Section 3.7.2. Converting from floating point to signed binary will naturally round the result. The IEEE standard has strict definitions for how the rounding is to be performed, but these definitions are not adhered to in this instance and the result is simply rounded down to nearest whole number.

2.10.3 Floating-point addition and subtraction

Floating-point addition and subtraction is, surprisingly, more complex than multiplication or division. Floating-point subtraction can be performed using the same method as for floating-point addition by simply changing the sign bit of the floating-point. This is true because mathematically, $a - b = a + (-b)$. Floating-point addition/subtraction between two floating-point numbers input A and input B into an output C can be done with the following steps performed in order:

1. If input A is zero, return input A for addition and for subtraction negate the sign of input B and return the result.
2. If input B is zero, return input A.
3. Extract sign A, exponent A and mantissa A from input A, and extract sign B, exponent B and mantissa B from input B.
4. If subtraction is performed, invert sign B.
5. Add implicit bit to mantissa A and mantissa B. (Set bit 10 of mantissa to 1.)
6. Shift the mantissa and the exponent of the input with the lowest exponent until it match the higher exponent. Set exponent C to resulting exponent.
7. If sign A is negative, convert mantissa A to two's compliment. If sign B is negative, convert mantissa B to two's compliment.
8. Add mantissa A with mantissa B and form mantissa C. If the result is negative, invert the result and set sign C to 1.
9. Normalize the result, which means shifting the leftmost bit of mantissa C until it reaches the 10th bit position. The shift can be either left or right, depending on the position of the leftmost bit. The exponent C is also adjusted accordingly depending on which direction the mantissa is shifted.
10. Check for exponent C underflow. If an underflow is present, return zero.
11. Check for exponent C overflow. If an overflow is present, return maximum value possible together with sign C.
12. Return sign C, exponent C and mantissa C as a 16 bit binary integer.

The maximum value possible in step 11 is defined as the maximum value representable with a half precision floating point number, which is ±65504.

2.10.4 Floating-point multiplication/division

Floating-point multiplication and division is simpler in terms of complexity compared to addition and subtraction, but generally requires more hardware resources to perform. Floating-point division, just like normal division, is the most expensive and time consuming operation to perform in hardware. The exception for normal division is if the divisor is a power of two, for then the division can be performed by simply right shifting the dividend. For floating-point division, the numerical value of the dividend or divisor has little impact on run-time or hardware complexity. The multiplication or division between two floating point numbers input A and input B, resulting in output C can be achieved by the following steps performed in order:

1. If input A or input B is zero, return zero.
2. Extract sign A, exponent A and mantissa A from input A, and extract sign B, exponent B and mantissa B from input B.
3. Add implicit bit to mantissa A and mantissa B. (Set bit 10 of mantissa to 1.)
4. Sign c is equal to sign A xor sign B.
5. Add exponent A with exponent B and subtract the exp bias. This forms exponent C.
6. *Multiply or divide* mantissa A with mantissa B. This forms mantissa C.
7. Normalize the result, which means shifting mantissa C until the leftmost bit is in 20th bit position. Adjust exponent C as mantissa C is shifted.
8. Check for exponent C underflow. If an underflow is present, return zero.
9. Check for exponent C overflow. If an overflow is present, return maximum value possible together with sign C.
10. Return sign C, exponent C and mantissa C as a 16 bit binary integer.

The only difference in the procedure from floating-point multiplication or division is in step 6. The most hardware intensive part of the floating-point multiplication or division process is the multiplication/division of the 10 bit mantissas. Performing this operation in a single clock cycle would naturally incur a substantial cost in terms of hardware. Changing it to a multi-clock cycle operation would then naturally reduce the hardware overhead required to implement it, although at the cost of increased computation time.

2.11 JPEG vs JPEG2000

As previously mentioned, JPEG2000 offers several advantages over JPEG. One of the main advantages is the superior compression ratio that JPEG2000 offers, especially at low bit rates. Figure 2.55 shows images compressed using JPEG at different bit rates with the standard Y quantization table, standard VLC Huffman encoding and no smoothing effects. The images were compressed using the image compression program VCDemo.[33]

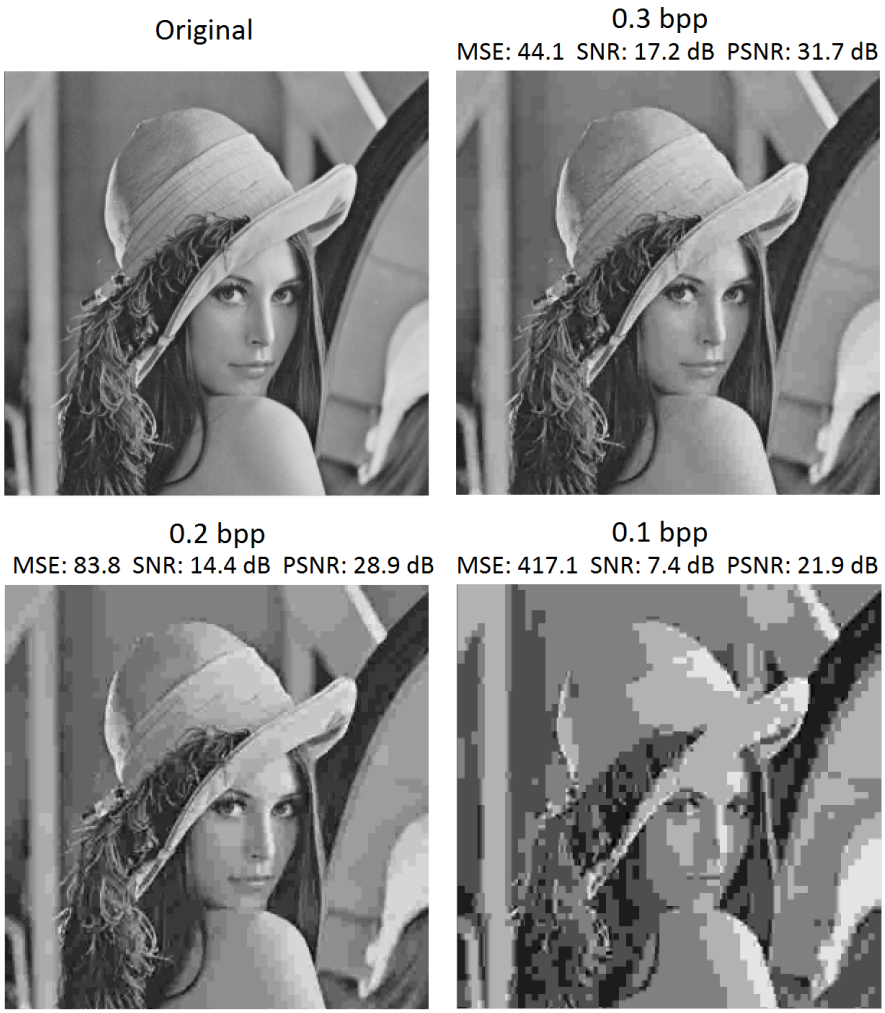

Figure 2.55: Comparison of different JPEG compression rates.

As is seen in Figure 2.55 the quality of the picture deteriorates quickly as the bit rate drops. The blocking artifacts characteristic of JPEG begins to become visible at 0.2 bits per pixel (bpp), and is especially egregious at 0.1 bpp. Clearly the lowest acceptable bit rate for JPEG with this picture in particular is 0.2 bpp.

Figure 2.56 compares different bit rates for JPEG2000. The compression was also done using VCDemo, with a 5-level wavelet transform and no tiling of the image. At 0.3 bpp the compressed image is visually nearly identical to the original. Even at 0.05 bpp it is still possible to discern details in the image.

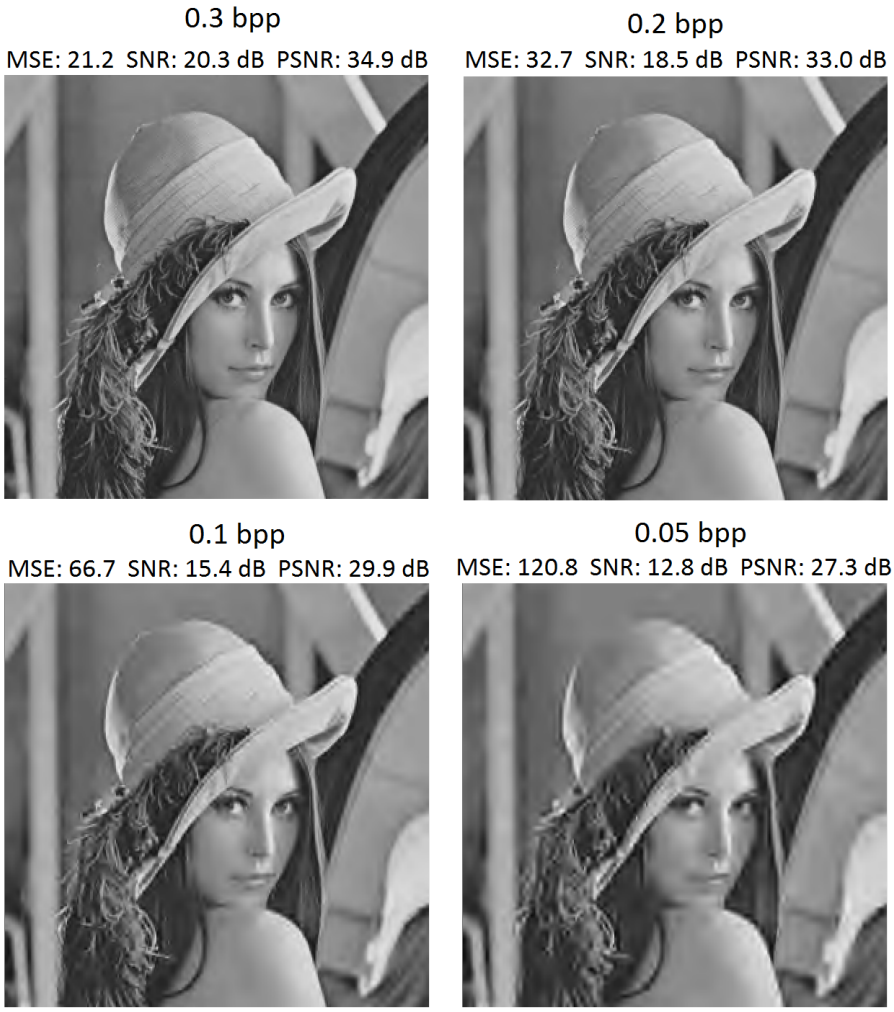

Figure 2.56: Comparison of different JPEG2000 compression rates.

Comparing Figure 2.55 and Figure 2.56 it is clear to see that JPEG2000 offers superior quality over JPEG at low bit rates. When comparing the results at 0.1 bpp it is especially clear that JPEG2000 is superior at low bit-rates compared to JPEG.

2.12 Xilinx Vivado Design Suite

Vivado Design Suite[34] is a program created by Xilinx to aid in the development and testing of HDL designs. It is a complete solution which enables simulation, synthesis and implementation of both Verilog and VHDL based designs in both SoC and purely FPGA based systems. Previous work done for NUTS make use of the MicroBlaze[11] softcore microprocessor developed by Xilinx, as well as an Numato Lab Xilinx Spartan-6 development board[35], making Vivado the natural choice of development tools for this project.

2.12.1 Synthesis and Timing Analysis

Vivado Design Suite offers several features to verify the functionality of a design. As mentioned, designs can be synthesized and thereafter the hardware resources required by the design can be determined. This makes it possible to verify if the design fits certain resource requirements, or if a chosen FPGA has the hardware capacity for it to be implemented. Hardware resources for an FPGA refers to the configurable hardware resources available on an FPGA, such as look-up tables (LUTs), registers (REGs), multiplexers (MUX), block random access memory (BRAM) and digital signal processors (DSP). The features and hardware resources offered by an FPGA varies depending on the manufacturer and model.

Vivado also offers a Timing Analysis function which verifies that a signal is able to propagate through the logic elements of the module in time for the result to become available during the subsequent clock cycle. The analysis also tests various other timing issues that might be present in a design. Failure to meet the timing criteria means that the module is unable to operate at the specified clock frequency. To remedy this, the clock frequency has to be reduced or the implementation redesigned. The Timing Analysis feature can also be used to determine the maximum operating frequency of the system. The Timing Analysis determines the slack of a signal.

The slack of a signal is defined as the difference between the required time and the actual time it takes for a signal to propagate. The Worst Negative Slack (WNS) is the signal path with the highest slack, while the Total Negative Slack (TNS) is the sum of all signal paths with a slack. If the WNS and TNS are negative, the design fails to meet the timing requirements because the signal arrives after the required time. If it is positive, then the design passes the timing requirement. Worst Hold Slack (WHS) and Total Hold Slack (THS) indicate that a signal is not present long enough on an input or output for the module to be able to read it properly during an active clock cycle. An Input and Output Delay is also often specified, which time it takes for a signal to become valid at an input or output port in relation to the clock of the transmitting or receiving module. FE stands for Failing Endpoint, and indicates how many of the total paths a signal can take that fails the timing test. If a design fails the Timing Analysis, the design itself must either be modified or the design clock speed has to be reduced. The Timing Analysis is used extensively in Section 3.

2.13 VCDemo

VCDemo is an interactive video and image compression software developed by Delft University of Technology.[33] It includes several of the most used image and video compression schemes, including JPEG and JPEG2000. It allows for customization of the compression systems, such as changing tile sizes, transform levels, quantization levels and bit-rate. This makes it easy to compare JPEG with JPEG2000, in addition to showing the impact small changes in the image compression system has on the final compressed image. Several of the comparisons done in this report was based on results from the VCDemo software, such as in Section 2.11.

3 Implementation

This section will go into detail about how the JPEG2000 compression system was implemented in hardware. The compression system is split into a number of smaller modules, each with its own purpose such as demosaicing, wavelet transformation, scalar quantization and so forth. The following sections will detail the purpose of each module, its method of operation and how it was implemented in hardware and the design decisions made for each module. Each module is represented using a flowchart unless the module control system is too large or complex to be practically displayed as a flowchart. In this case it is represented using a simplified system overview. In addition, a section is devoted to the previously developed modules and why it was decided to develop new modules. Synthesis and Timing Analysis results are also included in this section due to their importance in the implementation procedure. It should be noted that no proprietary IP cores or external designs were used in this project, meaning that all modules were designed from the ground up using only the the features offered by the VHDL language and the Xilinx Vivado Design Suite. A description of the connection interface of each module is included in Appendix B.

3.1 The previous work

This section will give a brief overview of the previous work done on the JPEG2000 compression system and why it was decided to not implement the previously developed modules and rather create new modules instead. Three relevant modules had been developed which were considered for implementation in this project[1]:

- A demosaicing module
- A color transform module
- A gamma correction module

The following sections will cover each of the previously developed modules in turn. Each section will check the functionality of the module, if the module is synthesizable, the resources required to implement the module and finally a Timing Analysis to verify that the module meets all timing requirements to operate at a minimum of 50 MHz.

3.1.1 Previous demosaicing module

Several versions of the previous demosaicing module exists as part of previous work conducted for NUTS.[1] This section will deal with the most recent version. In order to

test the functionality of the previous demosaicing module, a test image in Bayer format was created. As explained in Section 2.3.2, the purpose of demosaicing is to convert an image from the Bayer format to RGB. Figure 3.1 shows a section of the test image used.

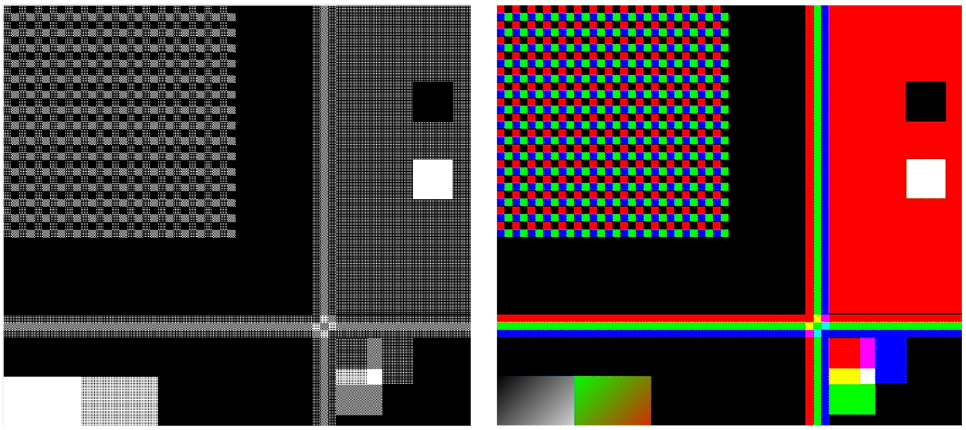

Figure 3.1: Bayer formatted test image (left) and expected demosaiced image (right) using MATLABs demosaicing function.

The image on the left in Figure 3.1 shows the Bayer formatted test image, while the image on the right shows the result expected after the demosaicing process. This resulting demosaiced image was generated in MATLAB using the inbuilt demosaic function. This image will serve as a reference used to verify the output of the previous demosaicing module. It was specified in the report for the old demosaicing module that it had several known faults of unknown origin. By running the supplied testbench for the module, the resulting output is shown in Figure 3.2.

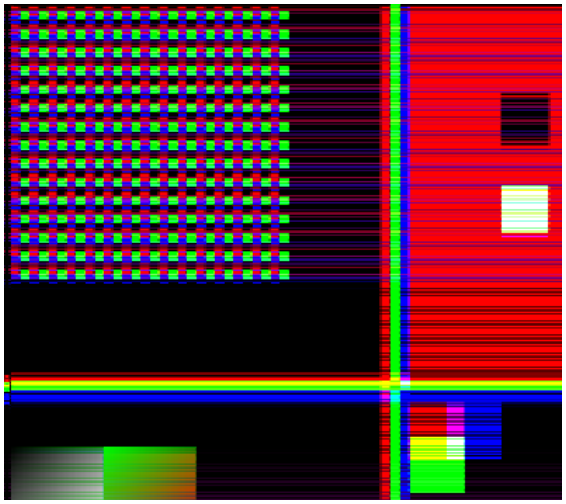

Figure 3.2: The result from the previously developed demosaicing module.

As Figure 3.2 shows, the resulting image from the module contains a high level of

unexpected artifacts in the form of horizontal lines. Some of the colors also appear to be slightly misaligned. These errors were mentioned in the previous report, but the author had been unable to determine the source of the artifacts.

When attempting to synthesize the module, Xilinx Vivado is unable to complete the synthesis, even after attempting to complete the process for several days. The reason behind this is not known definitively, but one hypothesis is the massive memory arrays initialized internally in the module failing to synthesize properly. [36] The module works by first loading in the first five horizontal lines of the entire Bayer image, performing demosaicing on these lines, writing them to the output port, then reading the next five horizontal lines and starting the operation over again. The issue with this method is that it requires one array to store the incoming Bayer image lines and three arrays to store the red, green and blue interpolated values temperately. Because the expected raw image from the camera sensor has a width of 2592 pixels, each array is initialized as a 2592 x 5 2D array, with each data point in the array containing a 16-bit pixel sample. This translates to 207 360 bits, or 25.92 kB per array. For four arrays, this equals 103.68 kB of internal storage needed when implementing the arrays in hardware.

Although this requires a significant amount of resources to store, it can easily be achieved by implementing the array as block RAM (BRAM) in hardware. This means that the demosaicing process has to access the array as if it was a RAM module, meaning that data is accessed one sample at a time. However, the demosaicing process of the previous demosaicing module is implemented in such a way that it accesses multiple values from the arrays simultaneously. This means that the arrays can not be synthesized as BRAM, but has to be synthesized as LUT elements instead. Storing over 100kB as LUT elements would require thousand upon thousand of elements, and far more than the resources available on the FPGA considered for the camera module. Because the module is not synthesizable, it is not possible to accurately determine the total resources the module would use, or if the module would pass the Timing Analysis. The module also makes extensive use of the *mod*-function, which is an expensive function to implement in hardware for *mod n* unless n is a power of two. Because of this, and in the light of the known faults in the demosaiced image produced by the module, it was decided to design a new demosaicing module from scratch rather than attempt to troubleshoot the old module.

3.1.2 Previous color transform module

The previous color transform module implements a reversible color transform (RCT) as described in Section 2.4.1. The RCT equation is reproduced here for convenience

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} \frac{R+2G+B}{4} \\ B-G \\ R-G \end{bmatrix} \tag{3.1}$$

The previous module accepts 16-bit RGB samples in and returns 16-bit YCbCr sam-

ples out. A testbench which was supplied with the module testes a small set RGB samples to verify the output. This small set of samples show that the output of the module is equal to the expected output, as shown in Figure 3.3. The input and output are shown in unsigned integer format, with Db and Dr being used to represent Cb and Cr respectively.

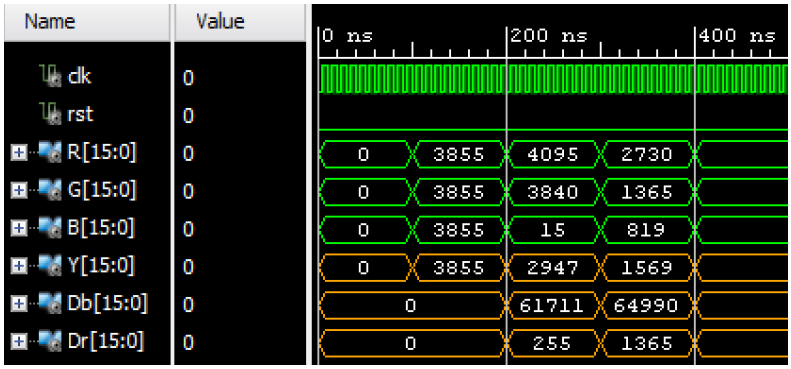

Figure 3.3: Testbench results from the previous color transform module.

As can be seen from the results in Figure 3.3, an input of $R = 4095$, $G = 3840$ and $B = 15$ results in a $Y = 2947$, $Cb = 61711$ and $Cr = 255$. In signed notation, $Cb = -3825$. Mathematically, following the equation for the RCT, this result is correct. However, this is not accounting for the fact that the dynamic range of the output must be equal to the dynamic range of the input. In this case, the input has a dynamic range of [0,65535], which means that the output can not be a negative number. An underflow therefore causes the high value of 61711 for Cb, which should in reality have been 0. This problem is relatively easy to fix, and a simple check which sets negative numbers to zero would have sufficed. The module is synthesizable, with Table 3.1 showing the resources required to implement the module in hardware.

Table 3.1: Previous color transform module resource usage

Slice LUTs	Slice REGs	F7 MUXs	F8 MUXs	Block RAM	DSP
49	0	0	0	0	0

As Table 3.1 shows, the hardware implementation requires relatively few resources. A Timing Analysis was also conducted on the module, as shown in Table 3.2. An Input Delay and Output Delay of 1 ns was added to the analysis.

Table 3.2: Timing analysis of the previous color transform module

Clock	WNS	TNS	FE	WHS	THS	FE
50 MHz	11.676 ns	0.000 ns	0/48	4.104 ns	0.000 ns	0/48

The result of the Timing Analysis shows that the module passes all tests. The bottom line is that the previous color transform module can be utilized in the current design

if the dynamic range issue is addressed. However, as explained in Section 2.4, the most suitable color transform for lossy compression is the irreversible color transform (ICT), and not the reversible color transform (RCT) as is used in this module. Due to the differences between the transforms, it was decided to create a new color transform module with the ICT instead rather than attempt to modify the previous module.

3.1.3 Previous gamma correction module

The previous gamma correction module implemented a fixed gamma correction of the form

$$Z = X^Y \tag{3.2}$$

Where Z is the gamma corrected output rounded to nearest integer, X is the input sample and Y is the gamma correction variable. The gamma correction variable in this case was fixed to 1/2, meaning that the gamma correction is essentially a square root operation of the input X. The square root function is implemented using what appears to be a non-restoring square root algorithm, although this is not fully explained in the original report. A testbench feeding the module with seemingly random values indicates that the module works as intended by squaring the incoming values.

Although a gamma correction variable of 1/2 is close to the recommended gamma correction value for raw images in general (1/2.2)[37], this module does not allow for color specific gamma correction or gamma correction using variables other than 1/2, which might be necessary depending on the camera sensor used. The module is synthesizable, and Table 3.3 shows the resources used by the module.

Table 3.3: Previous gamma correction module resource usage

Slice LUTs	Slice REGs	F7 MUXs	F8 MUXs	Block RAM	DSP
87	0	0	0	0	0

As Table 3.3 shows, this module utilizes relatively few hardware resources. However, the module is implemented in such a way as to perform the square root function in a single clock cycle. It is therefore important to run a Timing Analysis on the module to ensure that the module is able to operate at the specified frequency of 50 MHz. For the tests, an Input and Output Delay of 1 ns was added, which indicates the rise time of signals entering and exiting the module. It should be noted that this is a rather optimistic value for the Input and Output Delays. Table 3.4 shows the result of a Timing Analysis of the module at both 50 MHz and a lower clock frequency of 44.95 MHz.

The results in Table 3.4 shows that the module fails to meet the timing requirements with a clock frequency of 50 MHz. Only by lowering the frequency to 44.95 MHz does the module pass the Timing Analysis. Based on these results, and the fact that the

Table 3.4: Timing analysis of the previous gamma correction module

Clock	WNS	TNS	FE	WHS	THS	FE
50 MHz	-2.425 ns	-3.680 ns	2/8	4.041 ns	0.000 ns	0/8
44.95 MHz	0.000 ns	0.000 ns	0/8	4.041 ns	0.000 ns	0/8

module only utilizes a fixed gamma correction variable, it was decided that the module should not be used in the current design, at least not without significant modifications. The timing issue could possibly been resolved by adding some delays into the squaring process.

3.2 Camera module system overview

It is intended for the JPEG2000 compression module to be implemented as a module in a larger control system on the camera module FPGA. This control system has not yet been fully developed, but the current system consists of a softcore microprocessor which interfaces the camera sensor with an external microcontroller on the camera module circuit board. It will be the task of the softcore processor to interface the external microcontroller, camera sensor and external memory with the compression module. A proposed system overview is shown in Figure 3.4.

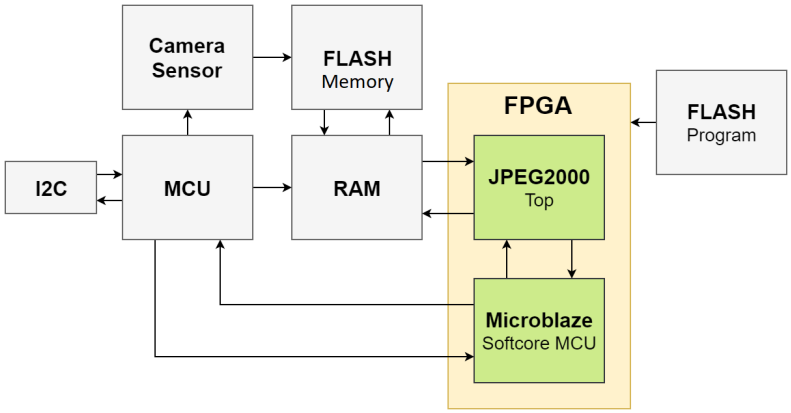

Figure 3.4: System overview of the payload camera module.

From the system overview, the MCU is an external microcontroller which handles the communication with the rest of the satellite via I2C. In order to reduce the power consumption of the camera module, the MCU will power down the FPGA and camera sensor when not in use.

The FLASH Memory is used to store raw Bayer encoded images from the camera sensor, and its capacity has to be large enough to enable storage of multiple images. The FLASH Program is used to store the FPGA configuration program. In order to start the

compression, the MCU moves the picture to be compressed to the RAM and powers up the FPGA. By communicating with the internal Microblaze softcore microprocessor, the MCU can provide the JPEG2000 module with information about the raw image and adjust parameters such as the bit rate. After compression is initiated, the compression module operates autonomously from both the external MCU and internal softcore microprocessor. The raw Bayer encoded image is read from the external RAM one tile at a time, compressed, and then written back to a dedicated part of the external RAM. After compression is complete, the external MCU moves the compressed tiles to the flash memory or transmits them to another storage location on the satellite.

3.3 The JPEG2000 top module

The purpose of the JPEG2000 top module is to control the flow of data through all the other internal modules in the system. It also acts as a communication point with the external systems of the FPGA, or eventually a softcore microprocessor internally on the FPGA itself. Figure 3.5 shows a system overview of the module with the internal modules and the external systems which the top module communicates with.

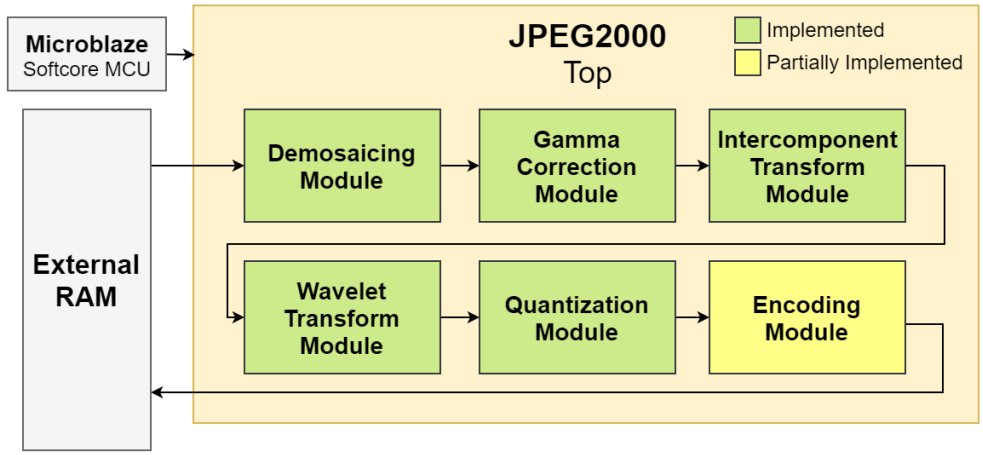

Figure 3.5: The JPEG2000 top module system overview.

The top module instructs the demosaicing module which tile to process. The demosaiced tile then passes through the other modules sequentially. The top module monitors the other modules and initiates the correct module in the right order. As is seen from the top module overview in Figure 3.5, the encoding module is only partly implemented. This was due to time limitations associated with the completion of the thesis, and the partial implementation is further expanded upon in Section 3.9. The compression system is designed to operate on all three color components of the image concurrently.

It should also be noted that the overview looks different from the JPEG 2000 compression overview shown in the theory from Section 2.2. In the VHDL implementation, the different modules perform the following operations:

1. **Demosaicing Module** - Image tiling, demosaicing.
2. **Gamma Correction Module** - Gamma correction, bit depth reduction.
3. **Intercomponent Transform Module** - DC level shifting, color transform.
4. **Wavelet Transform Module** - Wavelet transform.
5. **Quantization Module** - Scalar quantization.
6. **Encoding Module** - Tier 1 Encoding (EBCOT and MQ coder), Tier 2 Encoding (Packetization). *Module is not fully implemented*.

Each of the modules are treated in separate sections in this chapter.

3.3.1 Hardware synthesis and Timing Analysis

The JPEG2000 top module, and subsequently the submodules which constitutes it, synthesizes successfully in Xilinx Vivado. Table 3.5 shows the hardware resources required to implement the newly developed JPEG2000 compression system with gamma correction enabled and including the partly implemented encoding module.

Table 3.5: JPEG2000 top module hardware resource usage.

Slice LUTs	Slice REGs	F7 MUXs	F8 MUXs	Block RAM	DSP
14325	2902	37	72	72	35

A Timing Analysis was also performed on the top module, with an Input and Output Delay of 5ns and a clock frequency of 50 MHz. The results of the Timing Analysis is shown in Table 3.6.

Table 3.6: Timing analysis of the JPEG2000 top module.

Clock	WNS	TNS	FE	WHS	THS	FE
50 MHz	4.257 ns	0.000 ns	0/7922	0.059 ns	0.000 ns	0/7922

As can be seen from the results of the analysis, the top module passes the Timing Analysis. By adjusting the clock frequency until the analysis failed, the top frequency of the JPEG2000 compression system was determined to be 63.5 MHz. Synthesis results and Timing Analysis results from all the submodules in the JPEG2000 compression system is shown in the subsequent sections.

3.4 The demosaicing module

The demosaicing module performs two operations on the raw image from the camera sensor. It divides the image into tiles and performs the demosaicing process on each

tile from the external RAM memory as instructed by the JPEG2000 top module. Figure 3.6 shows the a simplified flowchart for the demosaicing module. The demosaicing process itself is further elaborated upon later in this section.

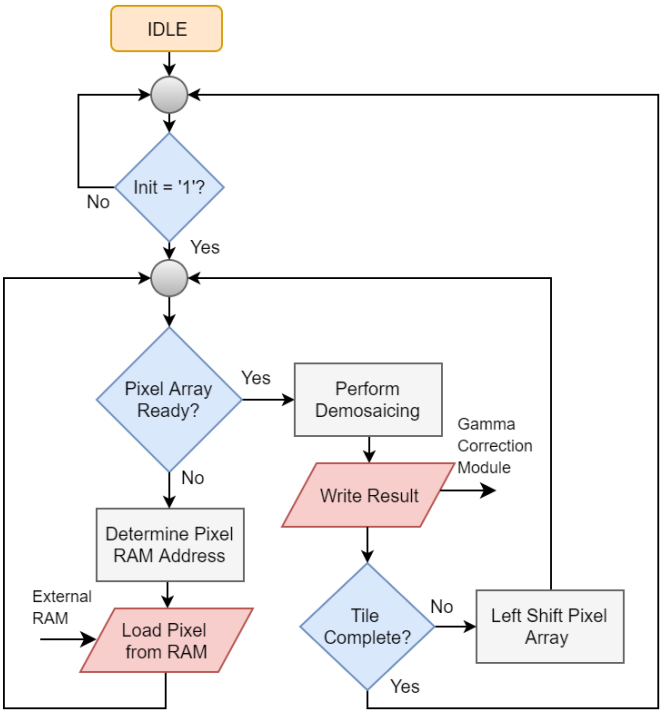

Figure 3.6: Simplified flowchart of the demosaicing module.

3.4.1 Image tiling

The demosaicing module is designed to operate on a tile size of 128x128 Bayer encoded pixels. The demosaicing module itself does not store an entire tile at once, but rather just the pixels required to perform the demosaicing process. The tile size was chosen as a compromise between the resulting image quality of the decompressed image and FPGA resources required to perform the compression. As outlined in Section 2.3.1, the tile size has a direct impact on image quality, with a smaller tile size causing clearly visible blocking artifacts. As will be detailed in the following sections, the tile size also affects the hardware resources used on the FPGA, as the system is designed to not store intermediate data on the external RAM of the camera module. This means that when a tile is read, it will be stored internally on the FPGA until the compression process is finished.

For the most part increasing the tile size only increases the amount of block RAM tiles utilized by the design. The total internal memory needed to store the tile specifically for this implementation is given by

$$M = \frac{N^2 \cdot B \cdot C \cdot K}{1000} \tag{3.3}$$

Where M is total memory in Kb, N is the tile size, B is the bit depth of each pixel, C is the number of color components for each tile and K is the total number of memory modules needed in the compression system. The bit-depth B is set to 16-bit because of the need to store half-precision floating point numbers, while C is set to 3 components because the image is RGB. K is set to 3, as two memory modules is needed for the DWT module and one is needed in the encoding module. The equation simplifies to

$$M = 0.144N^2 \tag{3.4}$$

Table 3.7 shows a comparison between tile sizes, memory required and the difference in PSNR between images compressed with the specific tile size as shown in 2.3.

Table 3.7: Tile size impact on internal memory and compressed image PSNR.

N	M	PSNR	Gain
32	148 Kb	27.8 dB	NA
64	590 Kb	32.9 dB	5.1 dB
128	2360 Kb	34.4 dB	1.5dB
256	9438 Kb	ca 34.9 dB	ca 0.5 dB
No tiling	967458 Kb	34.9 dB	ca 0 dB

As shown by the table, there is a considerable gain when increasing the tile size from 32x32 to 64x64, and also when increasing the size further to 128x128. Beyond that point the gain becomes vanishingly small while the memory requirement grows exponentially. From this, the tile size of the implemented solution was set to 128x128. If the amount of internal memory needed becomes a significant issue, the tile size can be lowered to 64x64, but at the cost of a reduction in the PSNR of a decompressed image of around 1.5 dB.

3.4.2 Demosaicing

Each tile read from the external memory is demosaiced using the Malvar et.al algorithm described in Section 2.3.2, which results in three tiles of red, green and blue components. Because the algorithm is based on interpolation of neighboring pixels, to demosaic a tile requires that some of the pixels in neighboring tiles are also read.

As described in the theory, the demosaicing process requires that a 5x5 array of pixels is read to determine the RGB components of a single pixel, as shown in Figure 3.7.

Figure 3.7: The demosaicing process.

The demosaicing module reads pixel samples from the external RAM where the Bayer encoded image from the camera sensor is stored. The module also determines the RAM addresses for the correct pixel samples based on the address of the first pixel in the image, which tile it is processing, and which pixel in the tile it needs to access. The tiles that constitutes the entire image is processed from left to right, and from top to bottom in a similar fashion to the scanning order of pixels in a tile block as shown in Figure 3.8. This means that tile 0 is in the upper left corner, tile 1 is on the right of tile 0 and so forth.

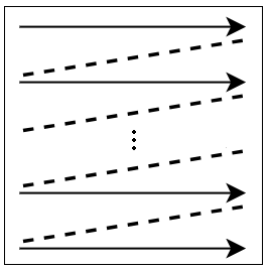

Figure 3.8: The demosaicing scan order within a tile block.

The JPEG2000 top module only has to provide the start RAM address for the first pixel of the image and which tile it wants demosaiced and the demosaicing module will handle all memory accessing and processing by itself. With no storage of intermediate pixel samples, the demosaicing module has to access on average 25 samples for every pixel in the image. For a single tile, this corresponds to over 400,000 external memory accesses. This would constitute a serious bottle neck for the entire JPEG compression system. If assuming that it takes 10 clock cycles to request and receive a sample from the external RAM, and assuming a clock frequency of 50MHz, it would take 80ms to perform demosaicing of one tile. One way to reduce the amount of external memory accesses is to exploit the fact that the interpolation of neighboring pixels make use of some of the same pixel samples. In fact, by simply shifting all the elements in the pixel array one position to the left, and only updating the rightmost column, the adjacent pixel can be interpolated. This process is illustrated in Figure 3.9.

By storing intermediate pixel samples, the processing time of a tile is reduced by

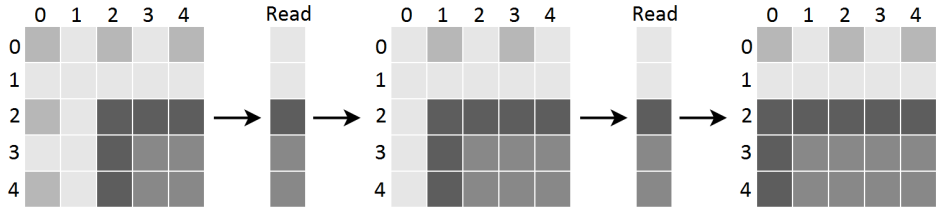

Figure 3.9: Array shifting to reduce external memory access and processing time.

around 80%, but at the cost of a higher complexity control system and an increase in hardware resources utilized by the module. Once the array is filled with the needed pixel samples, the demosaicing process is designed to perform the interpolation in a single clock cycle. This requires more hardware resources, as the multiplication part of the algorithm is done using combinational circuitry instead of sequential circuitry. On the other hand, it reduces the complexity of the control circuit and increases the running speed of the module. The resulting demosaiced pixel samples are then transmitted to the wavelet transform module.

3.4.3 Hardware synthesis and Timing Analysis

The demosaicing module synthesizes successfully in Xilinx Vivado, and the hardware resources required by the module is shown in Table 3.8.

Table 3.8: Demosaicing module hardware resource usage.

Slice LUTs	Slice REGs	F7 MUXs	F8 MUXs	Block RAM	DSP
979	395	0	0	0	8

As is shown, the module requires 8 digital signal processors (DSP) and a noticeable amount of other hardware resources. This is primarily due to the complex control system designed to reduce the number of external memory accesses as much as possible, and additionally the combinational nature of the multiplication logic. A Timing Analysis was also performed on the module, with a clock frequency of 50 MHz and a conservative Input and Output Delay of 5ns, with the result shown in Table 3.9.

Table 3.9: Timing analysis of the demosaicing module.

Clock	WNS	TNS	FE	WHS	THS	FE
50 MHz	5.224 ns	0.000 ns	0/1099	0.118 ns	0.000 ns	0/1099

By adjusting the frequency until the Timing Analysis test fails, it was determined that the maximum operating frequency of the demosaicing module is 67.5 MHz.

3.5 Gamma correction module

The gamma correction module performs bit depth reduction and gamma correction on the demosaiced image samples from the demosaicing module. The module operates on three pixel samples at once because the demosaicing process produces a red, green and blue pixel sample simultaneously. Figure 3.10 shows the operation of the module.

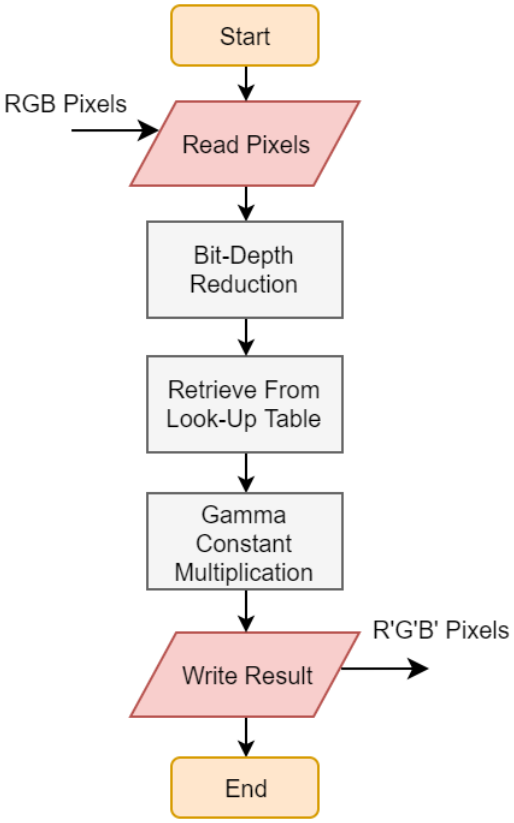

Figure 3.10: Simplified flowchart of the gamma correction module.

3.5.1 Bit depth reduction

The bit depth of the pixels from the demosaicing module is 12 bits. This is reduced to 8 bits in order to reduce the computational complexity associated with the gamma correction step and to save hardware resources. Doing this has a negligible impact on the image quality, but reduces the information content of each pixel by 50%. The reduction is performed by doing a simple right shift operation four times on the incoming pixel samples, which is equivalent to a division by 16. After the depth reduction, the pixel samples are gamma corrected.

3.5.2 Gamma correction

The gamma correction process exponentiates the incoming pixel sample by a constant exponent $0 < \gamma \leq 1$ and multiplies the result by a constant $0 < A \leq 1$, as outlined in Section 2.3.3. The exponent γ and constant A is considered constant as they are intended to correct for artifacts introduced by the camera sensor, which means that there is no need to change them during the operation of the camera module. By considering them constant, the computational complexity of the gamma correction module can be reduced considerably, as exponentiation by an arbitrary fraction is a non-trivial operation to do in hardware. Certain limitations have been imposed on the gamma correction constants to simplify its implementation in hardware. That is,

$$A = 0.1n, \quad n = 1, 2, 3 \ldots 10 \tag{3.5}$$

$$\gamma = 0.1n, \quad n = 1, 2, 3 \ldots 10 \tag{3.6}$$

In other words, the constants are limited to 10 different values each, which greatly simplifies the mathematical operations. The multiplication of the constant A is performed through a shift-multiplication procedure. This entails multiplying the pixel sample with another constant $A^{'}$ which is then right shifted k number of times to produce an approximate result. By performing the shift operation 8 times, the operation is equivalent to a division by 256.

$$A \cdot X \approx \frac{A^{'} \cdot X}{2^k} = \frac{A^{'} \cdot X}{2^8} \tag{3.7}$$

$$A^{'} = 10 \cdot A \cdot 2^k = 10 \cdot A \cdot 2^8 \approx 260 \cdot A \tag{3.8}$$

By multiplying A by 260, it results in $A^{'}$ being a natural number for the whole range of A. This is crucial to avoid fractional multiplications, which is the point of performing the shift-multiplication procedure. The multiplication by a fraction A has been reduced to a multiplication of the natural number $A^{'}$ and a 8 right shift operations. The result is only approximate, with an average error of 8.6% in the resulting value. The simplest way to perform the fractional exponentiation in hardware is to use look-up tables. This is a simple method which consists of a look-up table of precalculated values which are then mapped to the value of X. The downside of using look-up tables is that the size of the look-up table scales as a function of the dynamic range of the pixel input X. Because of the bit-depth reduction performed before the gamma correction operation, the dynamic range of the pixel values are [0,255], or 256 levels. With γ remaining constant, the look-up table has to store 256 values. If no bit-depth reduction had been performed, the dynamic range of X would be 4096 levels, which corresponds to a table with 4096 values, which naturally requires more hardware resources. Because X is always a natural number, the look-up table can be initialized as an array with X corresponding to the index in the array as shown partially in in Table 3.10. The resulting values from the

Table 3.10: Part of the look-up arrays for the fractional exponentiation in the gamma correction process.

Index (X)	$\gamma = 0.1$	$\gamma = 0.2$	$\gamma = 0.3$	...	$\gamma = 0.8$	$\gamma = 0.9$	$\gamma = 1$
0	0	0	0	...	0	0	0
1	1	1	1	...	1	1	1
2	1	1	1	...	2	2	2
3	1	1	1	...	2	3	3
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
252	2	3	5	...	84	145	252
253	2	3	5	...	84	145	253
254	2	3	5	...	84	146	254
255	2	3	5	...	84	146	255

exponentiation is rounded so that they can be represented as natural numbers. This also introduces some error in the exponentiation process.

One way to reduce the error between the computer and expected result would be to implement floating- or fixed-point arithmetic. However, to do so would increase the hardware cost of the gamma correction module considerably, and was decided that the introduced error is acceptable.

3.5.3 Hardware synthesis and Timing Analysis

The gamma correction module is fully synthesizable, and the hardware resources required by the module when performing gamma correction on all incoming pixel samples is shown in Table 3.11.

Table 3.11: Gamma correction module resource with gamma correction.

Slice LUTs	Slice REGs	F7 MUXs	F8 MUXs	Block RAM	DSP
159	24	35	12	0	0

If gamma correction is disabled, meaning that the constants $A = 1$ and $\gamma = 1$, then the look-up table is not synthesized, and samples merely pass through the module without the need for arithmetic operations. Table 3.12 shows the resulting resource use in this configuration.

Table 3.12: Gamma correction module resource without gamma correction.

Slice LUTs	Slice REGs	F7 MUXs	F8 MUXs	Block RAM	DSP
0	24	0	0	0	0

Hardware resources can be saved by disabling the gamma correction feature and instead perform it when the image is received back on earth.

Table 3.13 shows the result of a Timing Analysis performed on the module. The clock speed is set to 50 MHz, and a conservative 5 ns Input and Output Delay is added to the ports of the module.

Table 3.13: Timing analysis of the gamma correction module.

Clock	WNS	TNS	FE	WHS	THS	FE
50 MHz	9.306 ns	0.000 ns	0/72	4.121 ns	0.000 ns	0/72

By adjusting the clock speed until the Timing Analysis fails, it was found that the maximum operational clock frequency of the module is 93.4 MHz. The synthesis report and Timing Analysis indicates that the module can in all likelihood be implemented without issues.

3.6 Intercomponent transform module

The intercomponent transform module performs two operations on the pixel samples from the gamma correction module. First the samples are DC level shifted before being color transformed using the ICT as described in Section 2.4.2. Just as with the gamma correction module, the module performs the operation on three pixel samples simultaneously and performs it in a single clock cycle. Figure 3.11 illustrates how the module operates.

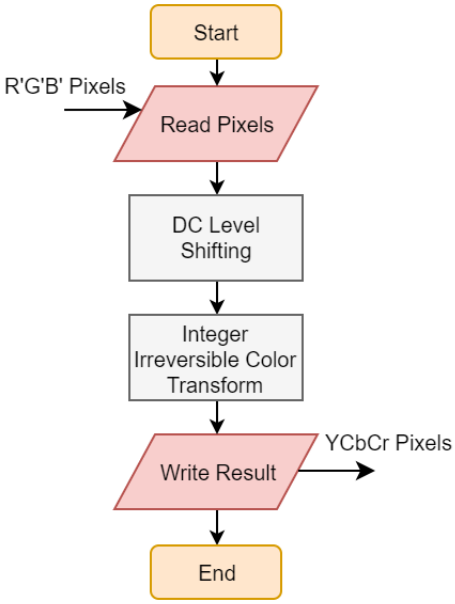

Figure 3.11: Simplified flowchart of the intercomponent transform module.

As seen from the flowchart, the flow of the module is relatively simple and similar to the gamma correction module.

3.6.1 DC level shifting

The DC level shifting process on the intercomponent module simply shifts the middle value of the pixel sample to zero. For a pixel sample with a dynamic range [0, 255], the shift results in a dynamic range of [-128, 127]. Negative numbers are represented in the two's compliment format, and the DC level shifting is performed by subtracting 128 from each pixel sample, as is described in Section 2.3.4. After being level shifted, the pixel samples are color transformed.

3.6.2 Color transform

The color transform process uses the integer based irreversible color transform as outlined at the end of Section 2.4.2. This reduces the transform to a simple multiplication and shift operation rather than fractional multiplication, although some error is introduced by this approximation similarly to in the gamma correction module. After the color transform, the pixel samples are transmitted to the wavelet transform module.

3.6.3 Hardware synthesis and Timing Analysis

The intercomponent transform module is fully synthesizable using Xilinx Vivado, with the hardware resources required to implement the module shown in Table 3.14.

Table 3.14: Intercomponent transform module hardware resource usage.

Slice LUTs	Slice REGs	F7 MUXs	F8 MUXs	Block RAM	DSP
573	24	0	0	0	0

As seen in Table 3.14, the module requires a noticeably large amount of resources. Similarly to the demosaicing module, this is due to the module performing the operation in a single clock cycle, making the hardware logic combinational. A Timing Analysis was also performed on the module, with a clock frequency of 50 MHz and an Input and Output Delay of 5ns. The results of the analysis is shown in Table 3.15.

Table 3.15: Timing analysis of the intercomponent transform module.

Clock	WNS	TNS	FE	WHS	THS	FE
50 MHz	4.610 ns	0.000 ns	0/72	4.329 ns	9.500 ns	0/72

The maximum operating frequency of the module is determined to be 64.9 MHz based on Timing Analysis results.

3.7 Wavelet transform module

The wavelet transform module is one of the more complex parts of the compression system and the module requiring the most hardware resources due to heavy usage of floating point arithmetic. The wavelet transform module performs the CDF 9/7 DWT as described in Section 2.5. Figure 3.12 shows a basic overview of the module.

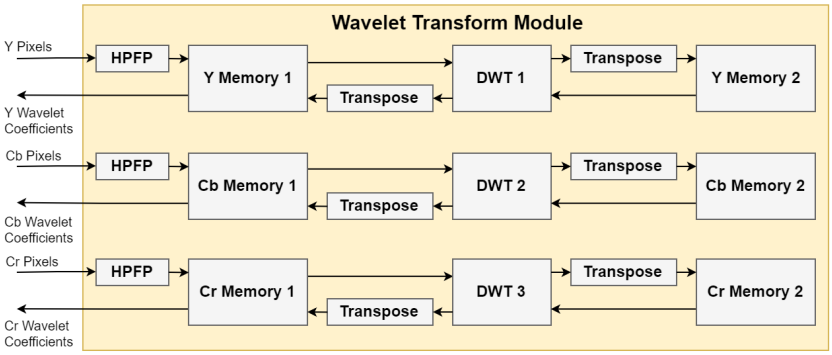

Figure 3.12: Overview of the wavelet transform module.

The input pixels are converted from signed binary to the half-precision floating-point in the module before being written to the Memory 1 memory. The pixels are then transformed in the DWT architecture, which performs a 1D transformation. Thereafter the wavelet coefficients are transposed, stored in Memory 2 and then transformed again before a second transposition occurs. The content of Memory 1 is now a 1 level 2D DWT of the incoming tile. To perform a multi-level transform, the lowest frequency subband in Memory 1 is run through the same process several times depending on the level wanted.

3.7.1 The lifting scheme architecture

The components that make up the lifting scheme architecture as described in Section 2.5.2 is shown in Figure 3.13. The lifting scheme architecture is the component marked as DWT in the system overview shown in Figure 3.12.

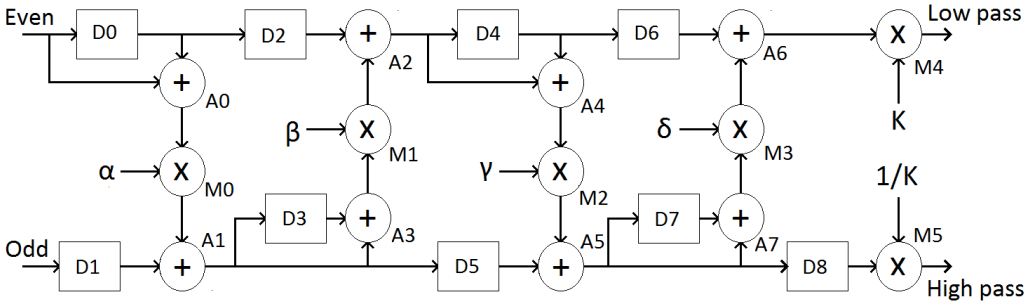

Figure 3.13: The lifting scheme hardware architecture.

The architecture consists of a total of 6 floating-point multipliers, 8 floating-point adders and 9 delay registers, which are described in more detail in the subsequent section. Odd and even samples from the tile components stored internally inside the wavelet transform module is fed into the architecture at one sample per clock cycle. The resulting low pass and high pass components are stored in another part of the internal memory of the module. The delay registers in the architecture ensures that the pipelining of the pixel sample are correctly synchronized through the entire process. The floating point arithmetic operations requires a single clock cycle to complete an calculation, which is also compensated for by the delay register. Table 3.16 shows the length of the delay registers.

Table 3.16: The delay register lengths.

D(n)	Delay (clock cycles)
0	1
1	3
2	5
3	1
4	1
5	6
6	5
7	1
8	3

Table 3.17 shows the value of the constants used in the architecture as was described in Section 2.5. Some additional rounding of these constants takes place as they are converted to the half-precision floating-point format.

Table 3.17: The constants used in the lifting scheme architecture for the 9/7 CDF DWT.

Constant	Value
α	-1.5859375
β	-0.0529785
γ	0.8828125
ϵ	0.443359375
k	1.154455289473
$\frac{1}{k}$	0.86962890

3.7.2 Floating-point arithmetic

The floating-point multipliers and adders used in the lifting scheme architecture is based on the theory discussed in Section 2.10.1. These are the most important building blocks of the architecture, and also the parts requiring the most resources, especially because the operations are designed to be performed in a single clock cycle. In addition, units converting to floating point from signed binary and vice versa also constitutes a part of the wavelet transform module. A simplified flowchart in Figure 3.14 shows how floating point conversion is achieved in accordance with theory from Section 2.10.1.

When converting from floating point to signed binary, the output range is from -32,768 to 32,767. Any floating point number outside this range is rounded up or down to the minimum or maximum output. In addition, fractional numbers are rounded down to the nearest whole number. Since only floating point multiplication and addition/subtraction were needed in the architecture, floating-point division is not illustrated with a flowchart. Floating-point division was implemented and tested, but it was determined that the division can not be performed in a single clock cycle at 50 MHz while still passing Timing Analysis tests. A simplified flowchart for floating-point addition/subtraction is shown in Figure 3.15.

From the flowchart in Figure 3.15, the most expensive operation in terms of hardware resources is the addition between the mantissas and the normalization of the result. By changing the addition and normalization to a multi-clock cycle operation, a significant amount of hardware resources could be saved, and the operation could have been performed at a higher clock speed. On the downside, this would also have increased the number of clock cycles to perform the operation. As can be seen, the only difference between addition and subtraction is a negation of the sign of input B. In similar fashion, a simplified flowchart for the floating point multiplication procedure is shown in Figure 3.16.

In the floating point multiplication flowchart, the mantissa multiplication and normalization of the result requires the most hardware resources of the procedure. Just as with the addition/subtraction procedure, by changing the multiplication and normalization operation to a multi-clock cycle operation, a significant reduction in hardware resources utilized for the operation would have been achieved.

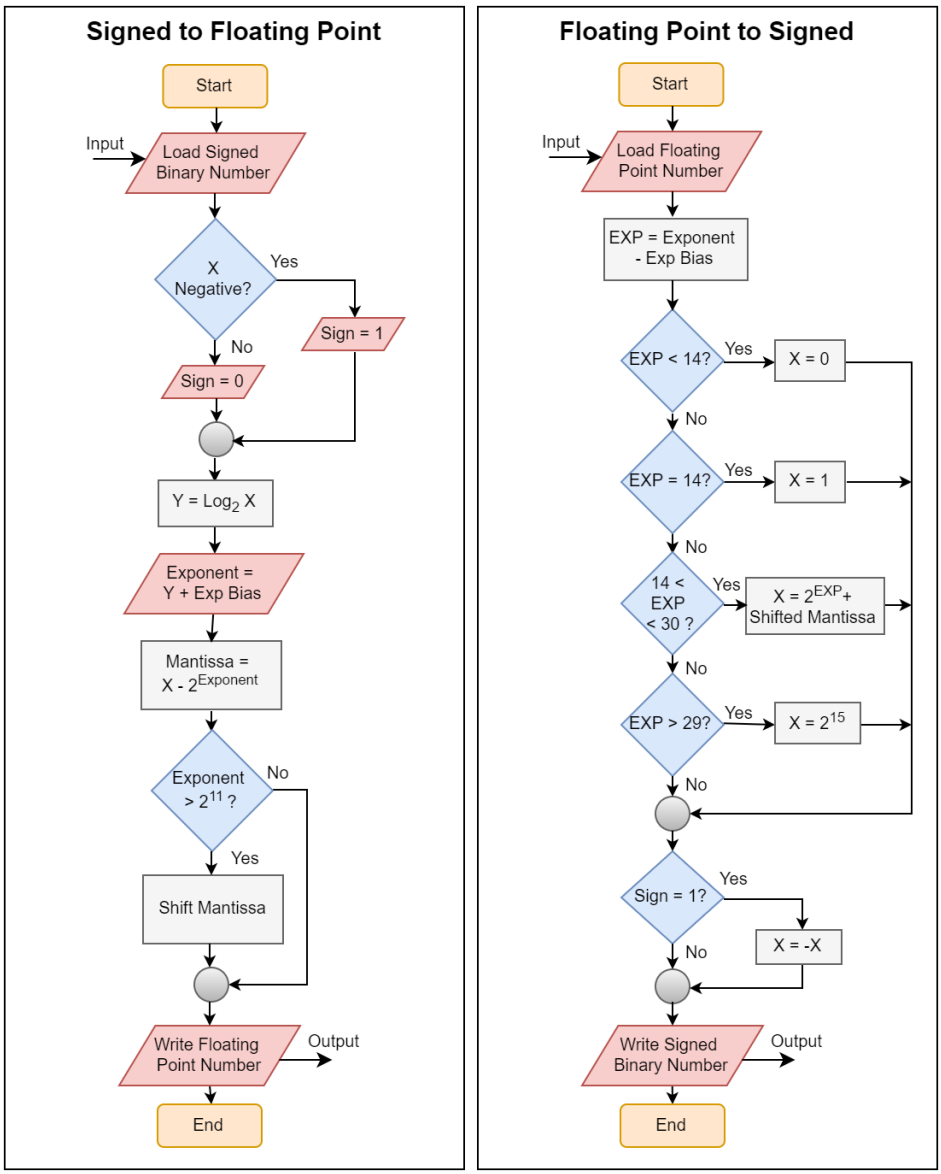

Figure 3.14: Simplified flowchart of the floating point conversion in the wavelet transform module.

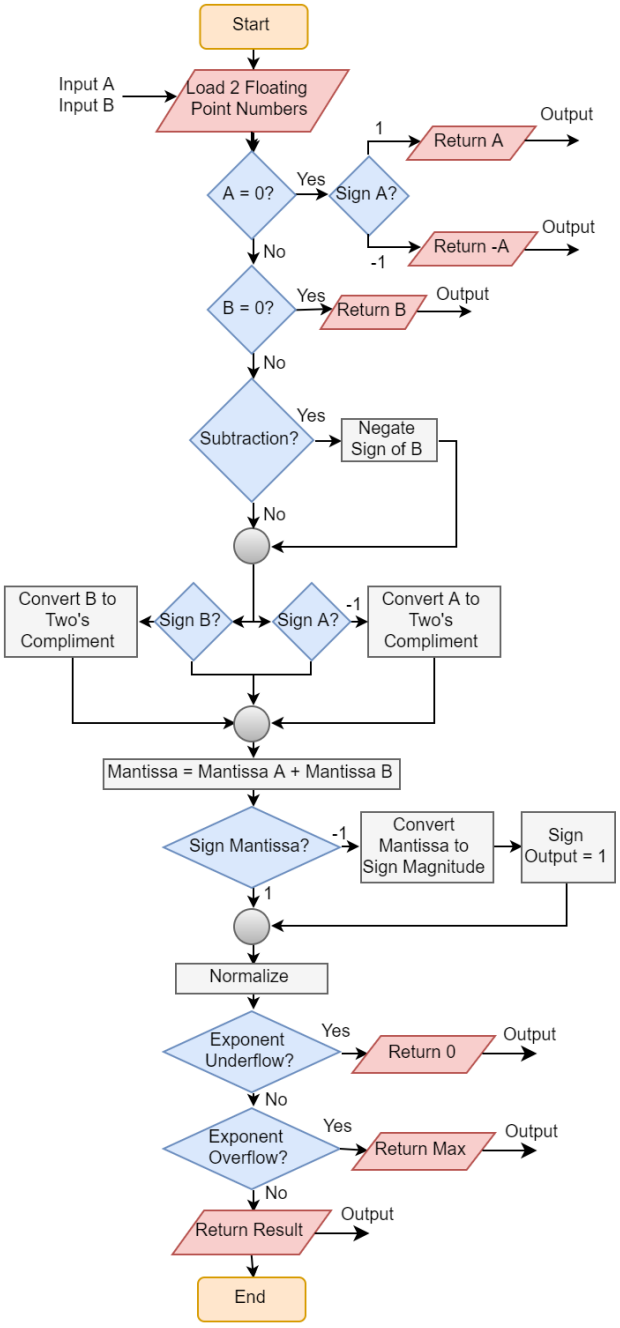

Figure 3.15: Simplified flowchart of the floating-point addition/subtraction procedure in the wavelet transform module.

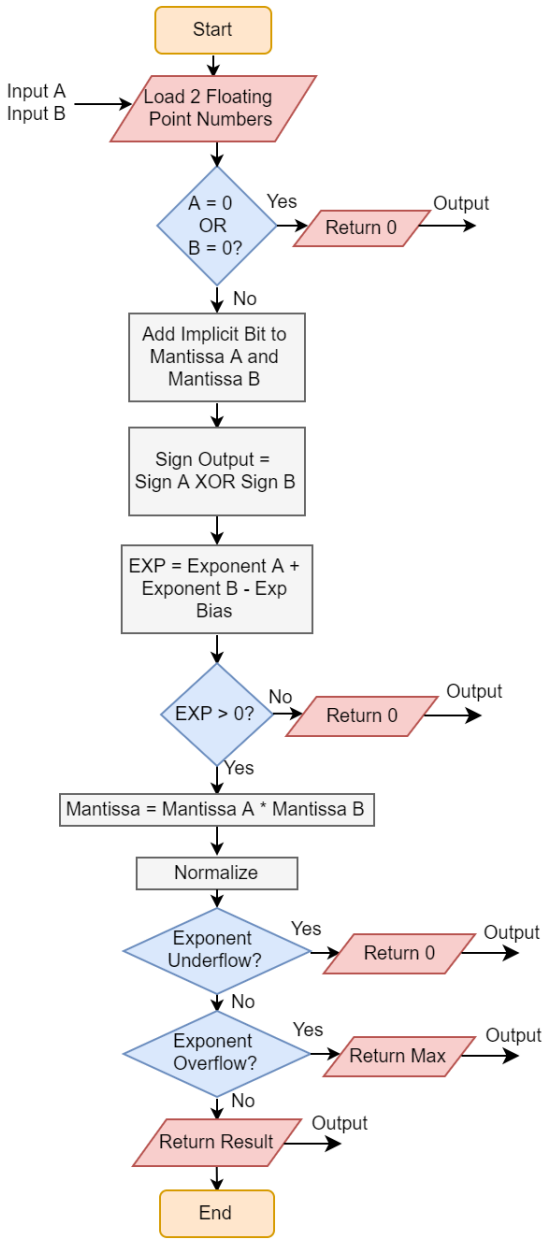

Figure 3.16: Simplified flowchart of the floating point multiplication procedure in the wavelet transform module.

3.7.3 Hardware synthesis and Timing Analysis

Both the lifting scheme architecture and the rest of the wavelet transform module is synthesizable in hardware. Table 3.18 shows the hardware resources required by the lifting scheme architecture.

Table 3.18: The lifting scheme architecture hardware resource usage.

Slice LUTs	Slice REGs	F7 MUXs	F8 MUXs	Block RAM	DSP
3458	400	0	0	0	8

As Table 3.18 shows, the architecture requires a substantial amount of hardware resources to implement. This can be attributed to the architecture containing 6 floating-point multipliers and 8 floating-point adders, all of which perform their operation in a single clock cycle. The resource usage could have been reduced significantly if the floating point arithmetic were changed to single clock cycle operations to a multi-clock cycle operation, though at the cost of significantly longer processing time and control complexity. As mentioned, the wavelet transform module is designed so that the three components that make up an RGB tile are processed simultaneously, meaning that three such lifting architectures are run in parallel. The total hardware resource use for the module is shown in Table 3.19.

Table 3.19: The wavelet transform module hardware resource usage.

Slice LUTs	Slice REGs	F7 MUXs	F8 MUXs	Block RAM	DSP
11702	1990	0	0	48	24

The large amount of resources from Table 3.19 is primarily from the three lifting scheme architectures and the internal memory modules used to store the intermediate results of the wavelet transform. Both block RAM and other hardware resources could have been reduced substantially by only processing one tile component at once, meaning that the red, green and blue tile component would have been processed sequentially rather than in parallel. This would naturally have increased the processing time of the wavelet transform module, but at the same time would have reduced the hardware resource usage by around two thirds.

A Timing Analysis was performed on the module, with a clock frequency of 50 MHz and an Input and Output Delay of 5ns. The results of the analysis is shown in Table 3.20.

Table 3.20: Timing analysis of the wavelet transform module.

Clock	WNS	TNS	FE	WHS	THS	FE
50 MHz	4.257 ns	0.000 ns	0/5140	0.059 ns	0.000 ns	0/5140

The maximum operating frequency of the wavelet transform module was determined to be 63.49 MHz.

3.8 Quantization module

The quantization module is implemented based on the theory described in Section 2.7. Two distinct versions of the module were developed, with the second version deemed superior to the first. While both versions perform the same task of quantizing the wavelet coefficients, the second version is faster, requires less hardware resources to implement and is far less complex. On the other hand, the first version can operate independently of the second version, which makes it possible to perform the wavelet transform of the next tile while still quantizing the previous tile. The scalar quantization itself in both versions is performed through half-precision floating-point multiplication of the wavelet coefficient with the reciprocal of the selected stepsize for the current subband. By using the reciprocal of the stepsize, floating-point multiplication can be used instead of floating-point division. Both versions are capable of quantizing each subband of a wavelet decomposition with a different stepsize.

3.8.1 Version 1

Version 1 of the quantization module is constructed in a similar fashion to the wavelet transform module. Pixel samples from the wavelet transform module are first loaded into internal memory modules before the quantization process starts. The quantization is performed by reading a sample from the internal memory, quantizing the sample, and then writing the sample back to the internal memory. After all the samples have been quantized, they are converted from half precision floating point to signed binary. Figure 3.17 shows a system overview of version 1 of the module.

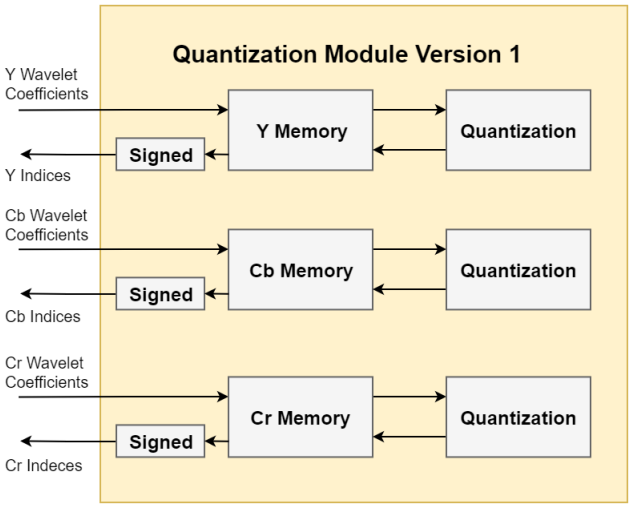

Figure 3.17: Overview of version 1 of the quantization module.

Further explanation of the quantization process itself is provided in Section 3.8.3.

3.8.2 Version 1 hardware synthesis and Timing Analysis

Version 1 of the quantization module is fully synthesizable, and the resulting hardware resources required to implement the module is shown in Table 3.21.

Table 3.21: Quantization module version 1 hardware resource usage.

Slice LUTs	Slice REGs	F7 MUXs	F8 MUXs	Block RAM	DSP
948	346	0	0	24	3

As can be seen, because of the storage of tile components internally in the module, a total of 24 block RAM units are required in the implementation. The primary reason for the relatively high resource requirement for the module is the floating-point arithmetic performed when quantizing the pixel samples. A Timing Analysis was also performed on the module, with the results shown in Table 3.22. The clock frequency was set to 50 MHz, with the Input and Output Delay set to 5 ns.

Table 3.22: Timing analysis of the version 1 of the quantization module.

Clock	WNS	TNS	FE	WHS	THS	FE
50 MHz	4.785 ns	0.000 ns	0/1185	0.118 ns	0.000 ns	0/1185

The maximum frequency of version 1 of the module was found to be 65.7 MHz by increasing the clock frequency until the analysis failed. One significant disadvantage of version 1 of the module is its impact on the processing speed of the compression system. By having to store the wavelet coefficients from the wavelet transform module into a separate memory in the quantization module, perform quantization on the content in the memory, and then write the quantized indices to the encoding module introduces significant delay in the system.

3.8.3 Version 2

Version 2 of the quantization module performs the quantization on each sample "on-the-fly" rather than first store them internally in the module, perform quantization on them and then write the quantized coefficients to the encoding module. Figure 3.18 shows a simplified flowchart for version 2 of the module.

Wavelet coefficients are received from the wavelet transform modules. Using an internal counter, the quantization module determines which subband the current wavelet coefficient pertains to. From Section 2.7, the stepsize δ has here been limited to an integer between 1 and 50. In order to quantize the result, the wavelet coefficient is divided by the stepsize. Performing division of a variable natural number which is not a power of 2 is a relatively hardware expensive operation, especially when performing it in a single clock cycle. In the wavelet transform module, the wavelet coefficients were processed as half-precision floating point-numbers. When transferring the coefficients from the wavelet transform module to the quantization module, they were on

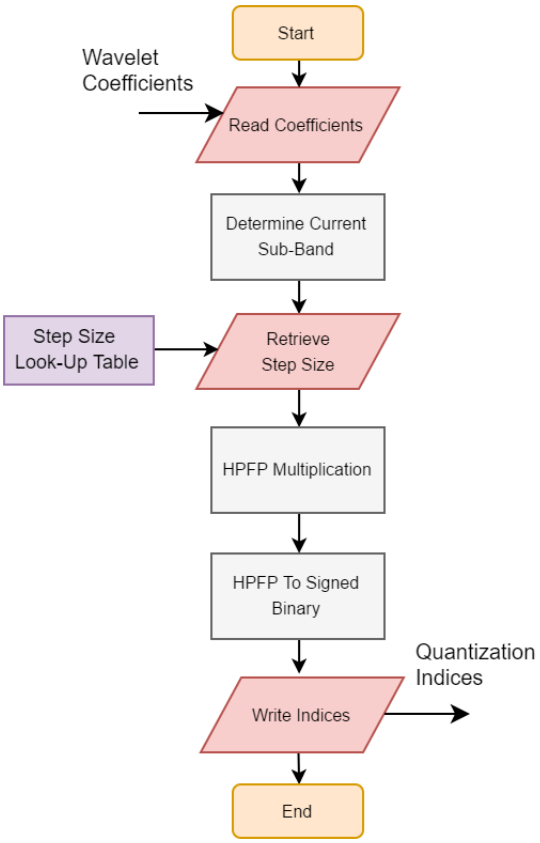

Figure 3.18: Simplified flowchart of version 2 of the quantization module.

purpose not converted back to signed integers. We can therefore perform floating-point arithmetic on the incoming coefficients without having to convert them first. Since the stepsizes are fixed in a range between 1 to 50, we can turn the division into a multiplication by simply changing the step size *delta* to a range between 1/1 to 1/50. By using a look-up table containing the new stepsizes in the floating-point format, the floating-point multiplication can be performed efficiently. This also enables variable step sizes for the different subbands. After determining which subband the coefficients belong to, a second array containing the appropriate look-up table indexes allows individual step sizes for the different subbands in the wavelet decomposition.

Once the appropriate stepsize has been retrieved from the look-up table it is floating-point multiplied with the the wavelet coefficient before being converted back to signed binary. This naturally rounds down the numerical value, as is required by the scalar quantization equation specified in Section 2.7. The resulting quantized wavelet coefficients are then transmitted to the encoding module.

3.8.4 Version 2 hardware synthesis and Timing Analysis

Version 2 of the quantization module is also fully synthesizable, and the results of the synthesis is shown in Table 3.23.

Table 3.23: Quantization module version 2 hardware resource usage.

Slice LUTs	Slice REGs	F7 MUXs	F8 MUXs	Block RAM	DSP
910	288	0	0	0	3

Version 2 requires less hardware resources, especially with regards to block RAM because the samples are not stored internally in the quantization module in this version. A Timing Analysis was also performed on the module, with the result shown in Table 3.24. The clock frequency was set to 50 MHz, with an Input and Output Delay of 5 ns.

Table 3.24: Timing analysis of the version 2 of the quantization module.

Clock	WNS	TNS	FE	WHS	THS	FE
50 MHz	4.525 ns	0.000 ns	0/646	0.124 ns	0.000 ns	0/646

It was shown that the maximum operating frequency of version 2 of the module is 64.6 MHz by increasing the frequency until the Timing Analysis fails.

3.9 Encoding module

The purpose of the encoding module was to perform the Tier 1 Encoding and Tier 2 Encoding of the JPEG2000 compression system as was outlined in Section 2.8 and Section 2.9. However, due to time constraints the EBCOT and MQ coder were not in a state were it could be implemented in the encoding module. Figure 3.19 shows the both the implemented and planned components which would make up the encoding module.

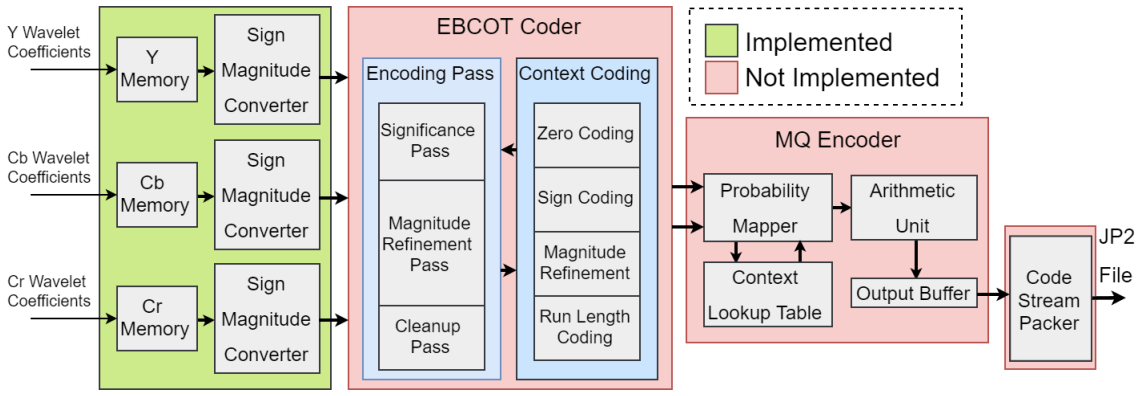

Figure 3.19: System overview of the partially implemented encoding module.

The encoding module would have received quantized wavelet coefficients from the quantization module and stored them in internal RAM modules. Once a code block is needed, the required samples are read from the memory and converted from two's compliment to sign magnitude representation before being passed to the EBCOT coder. Here, the code block would have been split into its constituent bit planes, with each bit plane encoded using the three encoding passes in order to generate a symbol and context label which would then be passed to the MQ coder which produces a probability from a lookup table based on the incoming data which are then used to perform the binary arithmetic encoding. The end result would have been an encoded bitstream. This would have constituted the Tier 1 Encoder. The last step would be to perform the packetization step of the Tier 2 Encoder, which would have taken place right after the MQ coder. The result from the packetization would have been a compressed JPEG2000 image with the JP2 file extension.

3.9.1 Hardware synthesis and Timing Analysis

Although the module was not complete, both synthesis and Timing Analysis tests were performed on the incomplete module. The module is synthesizable, and the hardware resources required are shown in Table 3.25.

Table 3.25: The incomplete encoding module hardware resource usage.

Slice LUTs	Slice REGs	F7 MUXs	F8 MUXs	Block RAM	DSP
198	217	0	0	24	0

As the synthesis results show, the current module requires few resources. This is expected to grow significantly as the EBCOT and MQ encoders are implemented. The Timing Analysis were performed with an Input and Output Delay of 5ns, and a clock frequency of 50 MHz. The results of of the analysis is shown in Table 3.26.

Table 3.26: Timing analysis of the incomplete encoding module.

Clock	WNS	TNS	FE	WHS	THS	FE
50 MHz	9.306 ns	0.000 ns	0/1105	0.118 ns	0.000 ns	0/1105

The analysis succeeded at 50MHz. By increasing the clock frequency until the analysis failed, the maximum operating frequency for the module was determined to be 93.4 MHz.

4 Testing and results

This chapter contains the testing procedures and results for the individual modules in the JPEG2000 compression system as well as for the system as a whole. It also measures the performance of the individual modules and compares them to similar implementations in MATLAB. By examining the comparison, the functionality of the module can be verified and evaluated.

4.1 Testing procedure

All the modules of the JPEG2000 compression system were tested either individually or in combination with other modules. The results of the tests were either visually inspected or compared against results acquired from MATLAB scripts designed to perform the same operation as the modules. We therefore designate the MATLAB results as a "gold standard/reference" model upon which we compare the results of the modules. This allows us to verify individual modules and track down potential errors in the operation of the compression system to a reasonable degree. The test scheme is outlined in Figure 4.1.

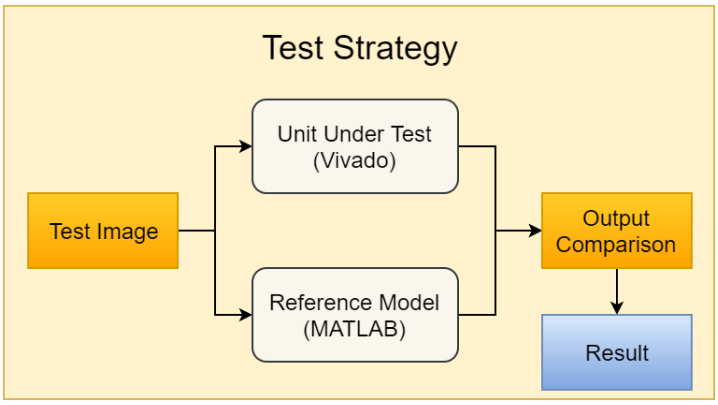

Figure 4.1: The test strategy for the modules.

Four pictures of different size and content were used to test the modules, each with three components R, G and B and with each component having a bit depth of 8 bits. This is to test the effects of tiling and how the compression ratio changes based on the content of the image, as well as to verify the functionality of the system. It also enables certain parameters such as the quantization step sizes to be adjusted to better suit the environment the camera module will be operating in.

Figure 4.2 shows two images of Lena with different resolution used as test images.

Figure 4.2: The 512x512 LenaBig and 128x128 LenaSmall test images.

The purpose of LenaSmall is to verify that the compression system can handle a single tile, as any image fed into the system is divided into equally sized tiles. The LenaBig image tests how the system handles an image consisting of several tiles. When it is processed, it will be divided into 16 tiles as illustrated in Section 2.3.1. These two test images illustrates more natural and relatively low frequency images and is not expected to be encountered in the environment the camera module will operate. Figure 4.3 shows the third test image used.

The HFBIG test image in Figure 4.3 is a computer generated image containing several high frequency regions with abrupt changes from one color to another. This tests the modules performance on high frequency images, although not the type of image expected to be encountered in the normal application of the camera module. The last test image used is shown in Figure 4.4.

The NASA image in Figure 4.4 is used to measure the performance of the system in a similar environment to the one that the camera module is expected to operate in. The four test images ought to give a sufficient foundation to determine the performance of the system.

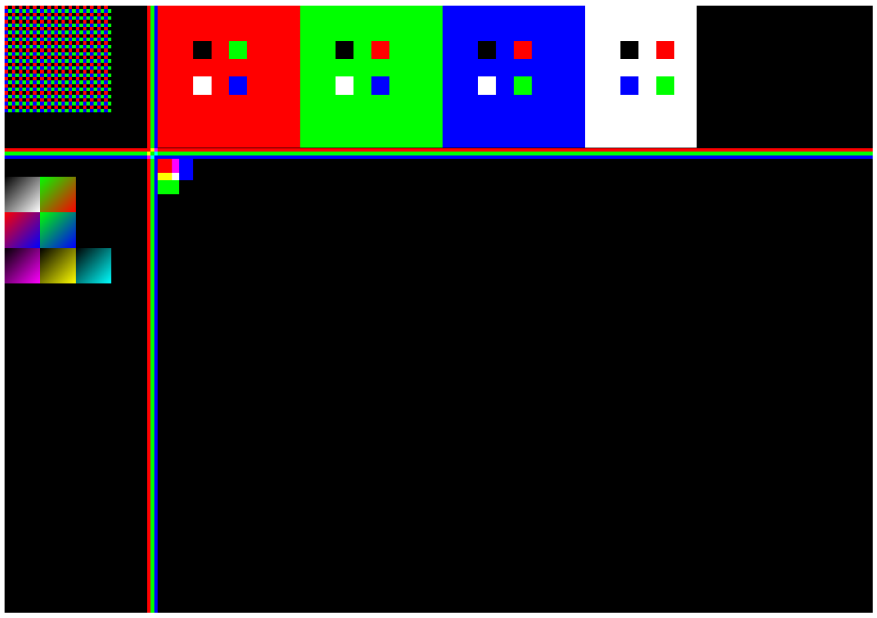

Figure 4.3: The 2592x1944 HFBIG test image containing high frequency areas developed as part of a previous project.[1]

Figure 4.4: Cropped 2592x1944 NASA picture showing the release of Cubesats over earth from outer space.[2]

4.2 Testbench

A testbench developed in Xilinx Vivado is used to simulate an external memory module containing a Bayer encoded test image. The external memory module is instantiated as RAM, and the device under test (DUT) communicates with the memory in the same way as real RAM memory. The testbench also provides the DUT with a start signal to indicate that the DUT is to begin its operation. Parameters such as bitrate are also supplied by the testbench. The output from the DUT is stored in a text file which is later analyzed. The functionality of the testbench is illustrated in Figure 4.5.

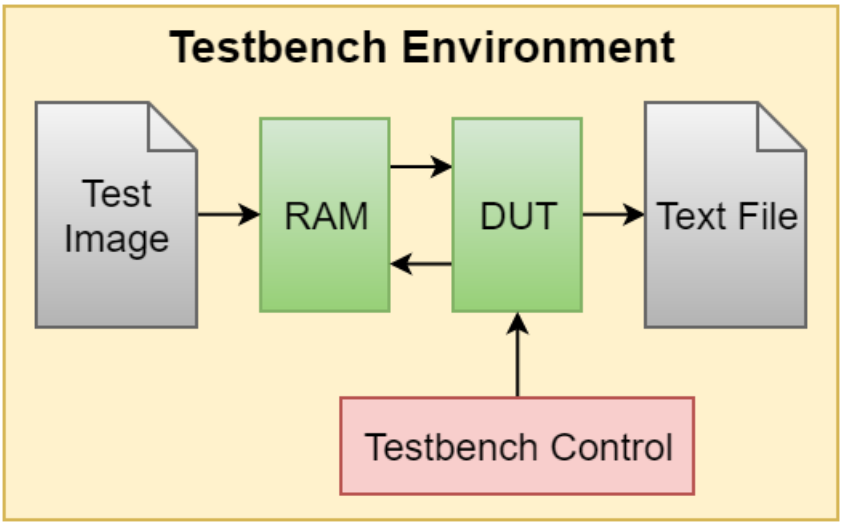

Figure 4.5: The testbench functionality.

For all the modules in the compression system, the output are integer values indicating pixel intensities, wavelet coefficients or quantized coefficients. As these are stored in a text file, a MATLAB script can analyze or construct an image from the output of the module. This result is then used to verify the functionality of the module as outlined in the previous section.

4.3 Demosaicing module

In order to test the demosaicing module, the test images are first converted to Bayer encoded images using MATLAB. This is done by simulating a 'GBRG' CFA over the test images and recording the resulting pixel intensities in a new image as outlined in Section 2.3.2. The result are 3 color component RGB test images turned into single component Bayer encoded images, as was shown previously in Figure 2.6. As was also shown in Section 2.3.2, seeing the result of the demosaicing process on a natural low frequency image is difficult. Therefore, the HFBIG test image was the image primarily used to verify the module. A cropped view of the demosaiced image is seen in Figure 4.6.

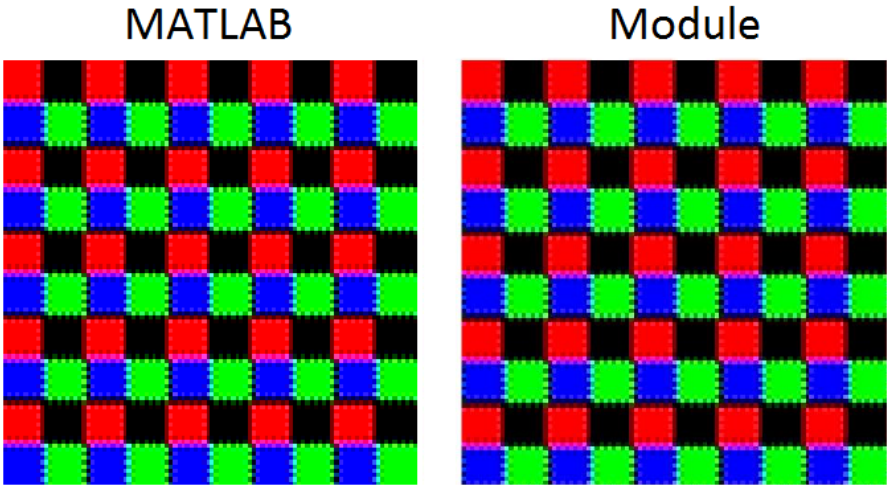

Figure 4.6: Image from the demosaicing module vs the MATLAB result.

The PSNR of the demosaiced image produced by the module compared to the image produced in MATLAB was 51.02 dB. This indicates that the module successfully performs the demosaicing process. Figure 4.7 shows the result of processing the LenaBig test image.

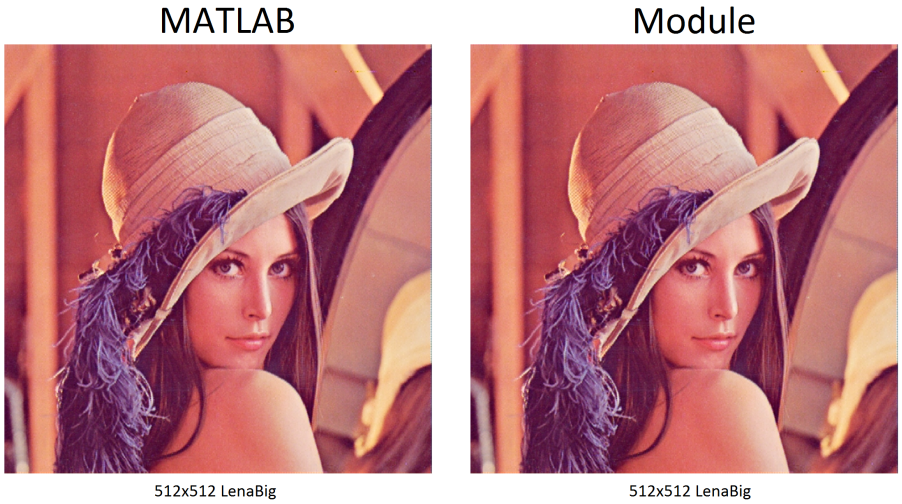

Figure 4.7: The test image LenaBig demosaiced with MATLAB and with the demosaicing module.

Visually, there is no perceivable difference between the demosaiced image from MATLAB and from the module. The other test images were also processed by both MATLAB and the module, with the results shown in Table 4.1. Also shown is the time it takes the module to process the test image. It is assumed that the external RAM can be accessed in one clock cycle.

Table 4.1: PSNR between MATLAB and the module for the demosaiced test images.

Image	MATLAB vs Original	Module vs Original	MATLAB vs Module	DIFF	Module run time
LenaSmall	28.06 dB	26.71 dB	49.25 dB	1.35 dB	1.98 ms
LenaBig	34.89 dB	33.37 dB	51.77 dB	1.53 dB	28.2 ms
HFBIG	31.73 dB	28.62 dB	47.93 dB	3.11 dB	596.2 ms
NASA	42.15 dB	41 dB	53 dB	1.15 dB	596.2 ms

As the results show, the demosaicing module produces images which are reasonably close to the images produced by the MATLAB script. The results are slightly different primarily because the module rounds the results from the demosaicing process in order to represent them using unsigned binary, which is different from the MATLAB implementation which does not round the result. The largest difference occurs in the HFBIG test image, which contains a high proportion of high frequency areas, which most demosaicing processes handles poorly.

4.4 Gamma correction module

Due to the time needed to run simulations in Xilinx Vivado, only LenaBig was used to test the gamma correction module. The module was tested in sequence with the demosaicing module, meaning that the Bayer encoded image is first demosaiced and then gamma corrected. Only a few combinations of gamma correction constants were used, but they are deemed sufficient to verify the functionality of the module. Figure 4.8 shows gamma correction performed by MATLAB and by the module.

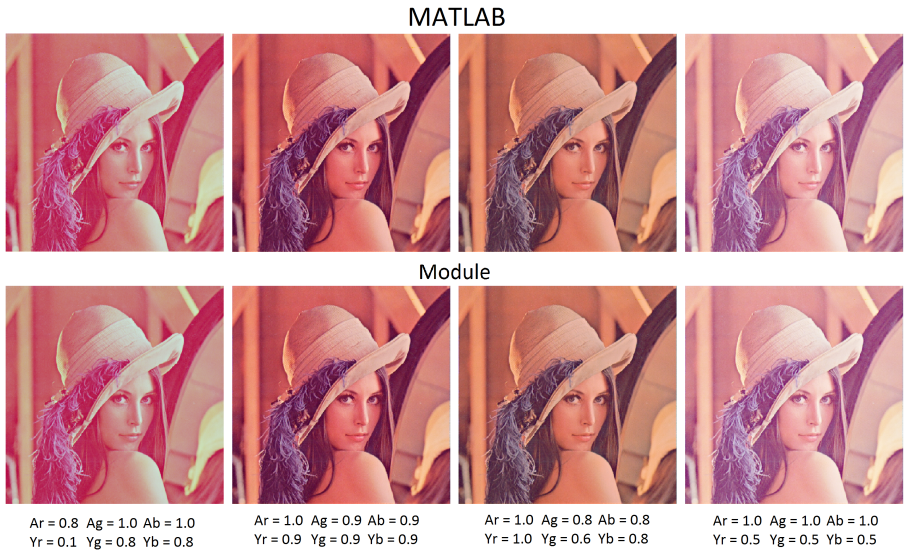

Figure 4.8: Gamma correction performed by MATLAB and the module.

Visually, the module appears to perform just as well as the MATLAB implementation. Table 4.2 show the resulting PSNR between the gamma corrections performed by the module and in MATLAB for the test image LenaBig.

Table 4.2: PSNR between MATLAB and module for gamma correction of LenaBig.

$[A_r, A_g, A_b]$	$[0.8, 1.0, 1.0]$	$[1.0, 0.9, 0.9]$	$[1.0, 0.8, 0.8]$	$[1.0, 1.0, 1.0]$
$[\gamma_r, \gamma_g, \gamma_b]$	$[0.1, 0.8, 0.8]$	$[0.9, 0.9, 0.9]$	$[1.0, 0.6, 0.8]$	$[0.5, 0.5, 0.5]$
Module vs MATLAB	44.08 dB	46.22 dB	46.54 dB	52.78 dB

As the results show, there are some differences in the PSNR between the MATLAB implementation and the module. This is attributed to the rounded look-up table for the gamma correction constant and the shift-multiplication described in Section 3.5.2, which are not used in the MATLAB implementation. In addition, since the gamma correction is done in together with demosaicing, the PSNR is also lower due to the errors incurred in the demosaicing process. Because the gamma correction is performed mostly to appeal visually to a human observer, the results are deemed acceptable.

4.5 Intercomponent transform module

Just as with the gamma correction module, the test image LenaBig were used to test the intercomponent transform module. The module was also tested in sequence with the demosaicing module, but with no gamma correction enabled. The MATLAB implementation of the color transform employs the irreversible color transform, while the module employs an integer version of the irreversible color transform as outlined in Section 2.4. Since the "irreversible" color transform has to be reversed when an image is decompressed, errors are introduced in the reconstructed image. This error increases somewhat when the reversing of the color transform is not done using the exact same process, which is the case when the MATLAB implementation reverses the color transformation performed by the intercomponent transform module. Figure 4.9 shows the ICT performed by the module and its reverse compared to the MATLAB implementation. Table 4.3 show the PSNR between the MATLAB implementation and the module. The reduction in PSNR can be attributed to the same shift-multiply approximations as were used in the gamma correction module.

Table 4.3: PSNR between MATLAB and module for the intercomponent transform of LenaBig.

DUT	PSNR
Module vs MATLAB	50.02 dB
Reverse of MATLAB vs original	109.34 dB
Reverse of module vs original	42.15 dB

Figure 4.9: Irreversible color transform performed by the module and MATLAB. Reverse color transform performed in MATLAB for both the module and MATLAB implementation.

4.6 Wavelet transform module

The initial testing of the wavelet transform module was done using a grayscale variant of the LenaBig test image. The initial tests were performed with the module separated from the demosaicing, gamma correction and intercomponent transform module. In addition, no tiling of the image was used during the initial tests, meaning that the whole image was processed as a single tile. Figure 4.10 shows the result from the module with a 1 level 2D DWT of a grayscale variant of the test image LenaBig.

Figure 4.10: The output of a 1 level 2D DWT from the wavelet transform module.

When visually comparing the result from the module with the expected result from Section 2.5 they appear very similar. Figure 4.11 shows the result when performing the inverse DWT on the image using MATLAB.

Figure 4.11: The resulting inverse DWT of the output of the wavelet transform module.

The resulting image after a 1 level 2D DWT followed by a inverse DWT now matches closely to the result generated in MATLAB. The next test was to verify that the module could perform multi-level 2D wavelet transform while producing a result which, when inverse transformed, will yield a good result compared to the original. To investigate this, the test image LenaBig was used, which consists of 16 tiles. Figure 4.12 compares a 1 level 2D DWT performed by the MATLAB implementation vs the module.

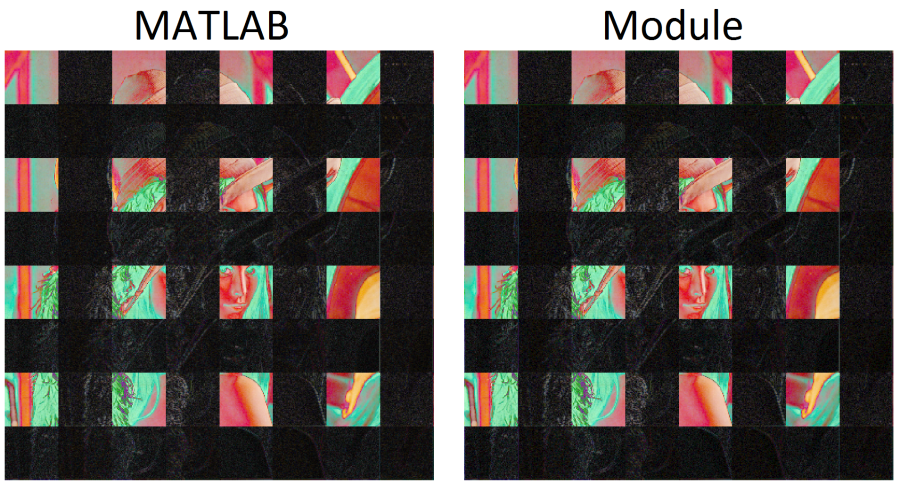

Figure 4.12: The 1 level 2D DWT of LenaBig by the MATLAB implementation and the module with a tile size of 128x128.

When performing the inverse wavelet transform on the result using MATLAB, the result is as shown in Figure 4.13. As can be seen, visually the result from the MATLAB implementation is identical to the result from the module.

Figure 4.13: The inverse transform of the 1 level 2D wavelet decomposition shown in Figure 4.12.

Table 4.4 shows the PSNR between the MATLAB implementation and module for a 1 level 2D DWT. In addition, the run time of the module to perform the transform is also shown.

Table 4.4: PSNR between MATLAB and the module for the 1 level 2D wavelet transformed test images.

Image	IDWT MATLAB vs Original	IDWT Module vs Original	DWT MATLAB vs Module	Module run time
LenaSmall	113.64 dB	44.29 dB	48.04 dB	1.1 ms
LenaBig	107.12 dB	48.52 dB	53.47 dB	20.8 ms
HFBIG	115.2 dB	47.02 dB	50.94 dB	442.8 ms
NASA	117.2 dB	53.1 dB	58.58 dB	442.8 ms

As can be seen from Table 4.4, the the module performs well when comparing the inverse of the module with the original image. A single tile is processed in 1.1ms, and a full 2592x1944 image can be transformed in 442.8ms. The NASA test image has the highest PSNR of 53.1 dB when compared to the original. It is expected that the MATLAB wavelet transform will yield a far higher PSNR because of its use of double-precision arithmetic.

For a 3 level 2D DWT transformation, the PSNR between the MATLAB implementation and the module is shown in Table 4.5. As is seen, the PSNR between the inverse transform of the module result and the original has decreased somewhat for the module. The PSNR of the NASA image has dropped around 3 dB compared to the 1 level 2D DWT. Again as expected, the MATLAB implementation retains its high PSNR due to almost no rounding during the transformation.

Table 4.5: PSNR between MATLAB and the module for the 3 level 2D wavelet transformed test images.

Image	IDWT MATLAB vs Original	IDWT Module vs Original	DWT MATLAB vs Module	Module run time
LenaSmall	113.89 dB	43.73 dB	56.6 dB	1.53 ms
LenaBig	116.89 dB	47.17 dB	63.16 dB	24.49 ms
HFBIG	113.3 dB	46.32 dB	62.83 dB	514.3 ms
NASA	115.4 dB	50.15 dB	67.63 dB	514.3 ms

Tests were also conducted for a 5 level 2D DWT, with the results shown in Table 4.6. This is the highest level achievable by the module with a tile size of 128x128. The LL5 sub-band in this instance is only 4x4 pixels.

Table 4.6: PSNR between MATLAB and the module for the 5 level 2D wavelet transformed test images.

Image	IDWT MATLAB vs Original	IDWT Module vs Original	DWT MATLAB vs Module	Module run time
LenaSmall	112.64 dB	43.07 dB	62.24 dB	1.56 ms
LenaBig	113.92 dB	45.57 dB	72.94 dB	24.94 ms
HFBIG	103.2 dB	44.44 dB	74.76 dB	523.8 ms
NASA	112.6 dB	45.2 dB	72.4 dB	523.8 ms

To illustrate the multi-level transform ability of the module, a 1 to 5 level transform of LenaSmall is shown in Figure 4.14. The result appear to match what was expected from the theory in Section 2.5.

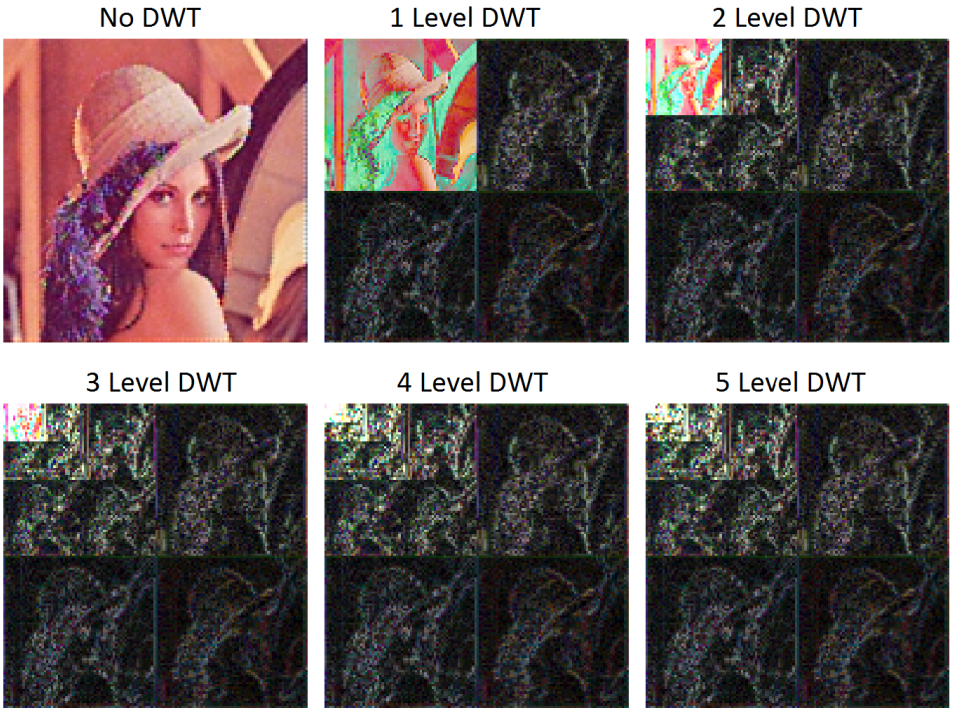

Figure 4.14: Test image LenaSmall transformed at different levels.

4.7 Quantization module

The quantization module were tested in connection with the other modules in the system. This means that when testing the module, the test images were first demosaiced and then wavelet transformed before quantization took place. Gamma correction and the intercomponent transformation was disabled for the test. Because the quantization module is tested in combination with the demosaicing module, the errors measured will be an accumulation of the errors in the demosaicing process as well.

Quantization can be performed with a uniform stepsize for all subbands, or with subband dependent stepsizes. Figure 4.15 shows the result of performing demosaicing, a 5 level 2D DWT and quantization with an uniform step size using the implemented modules, before reversing the process in MATLAB through dequantization and the inverse wavelet transform.

Figure 4.15: Reconstructed image after inverse wavelet transformation and dequantization with R=0.1.

In Figure 4.15, the PSNR between the reconstructed test image and a demosaiced version of the test image is shown. The comparison is done between a demosaiced version of the test image because the quantized image was demosaiced before it was quantized. The reconstruction was done with the constant R=0.1, and also shown is the calculated entropy E of the 5 level wavelet decomposition of the image as described in Section 2.7. Table 4.7 compiles the results of testing with multiple step sizes for the same test image LenaBig.

Table 4.7: PSNR and entropy E for uniform sub-band step size Δ. Dequantized with R=0.1 and the PSNR is between the reconstructed image and a demosaiced version of the original test image.

Step-size Δ	Reconstructed image vs demosaiced original	Entropy
1	42.27 dB	3.2137
5	40.06 dB	1.5125
10	37.53 dB	1.0364
20	34.56 dB	0.5775
35	33.30 dB	0.3377
50	32.83 dB	0.2491

From Table 4.7 it is clear to see that increasing the stepsize causes a reduction in the overall entropy level of the wavelet decomposition, which is in agreement with the theory outlined in Section 2.7.

4.8 Encoding module

Because parts of the encoding module were not implemented due to time constraints, and because no MATLAB implementation of the EBCOT or MQ encoder were developed either there were little that could be tested in with regards to the encoding module itself. The general control structure of the encoding module was tested, which transferred wavelet coefficients form the wavelet module to the internal memory of the encoding module. The sign magnitude conversion function was also tested, which converts from two's compliment format to sign magnitude format. Both were found to be operating as expected. Beyond this there was nothing else that could be tested for the encoding module.

5 Discussion

This section will discuss the more significant results gathered during the testing described in Section 4 and relate them to the design choices made when implementing the design in as outlined in Section 3. Suggestions for future work is also included in this section.

5.1 Preprocessing and intercomponent transform modules

All the modules involved in preprocessing and the intercomponent transform module were implemented without the use of floating-point or fixed-point arithmetic, despite in many cases having to perform arithmetic involving fractional numbers as described in Section 2.3 and Section 2.4. In order to accomplish this, approximations were used to reduce the fractional arithmetic to whole number arithmetic. Naturally, these approximations introduce some errors in the result, depending on the degree of the approximation made. This was seen in Section 4 when the results from the modules were compared to the results from a MATLAB implementation which did not rely on the same approximations. The effects the approximations had on the results varied from module to module.

The advantage of whole number arithmetic is a reduction in hardware complexity. To reduce the errors introduced in these modules, either floating- or fixed-point arithmetic would have to be implemented instead. This is a decision which would have to be made once a final decision on the camera module FPGA is made. If sufficient resources are available on the FPGA then improving the accuracy of the modules could lead to a noticeable gain in the PSNR of the reconstructed images.

5.2 Wavelet transform module

The wavelet transform module in the most complex and resource intensive module implemented, primarily attributed to the floating-point arithmetic needed to perform the lifting scheme based wavelet transform.

5.2.1 Using half precision floating point numbers

Choosing to implement half-precision floating point numbers instead of either single-precision or fixed-point numbers ultimately impacted both the hardware resources required and the accuracy of the results of the module. Since no literature could be

found on similar implementations of the lifting scheme using the half-precision floating point format there was no definitive way of knowing beforehand if the implementation would be successful or not.

From the testing performed on the module in Section 4.6, the graph shown in Figure 5.1 contains the result from performing the wavelet transform on the four test images at different transform levels. The PSNR shown is between the original test image and an inverse wavelet transform of the result from the wavelet transform module.

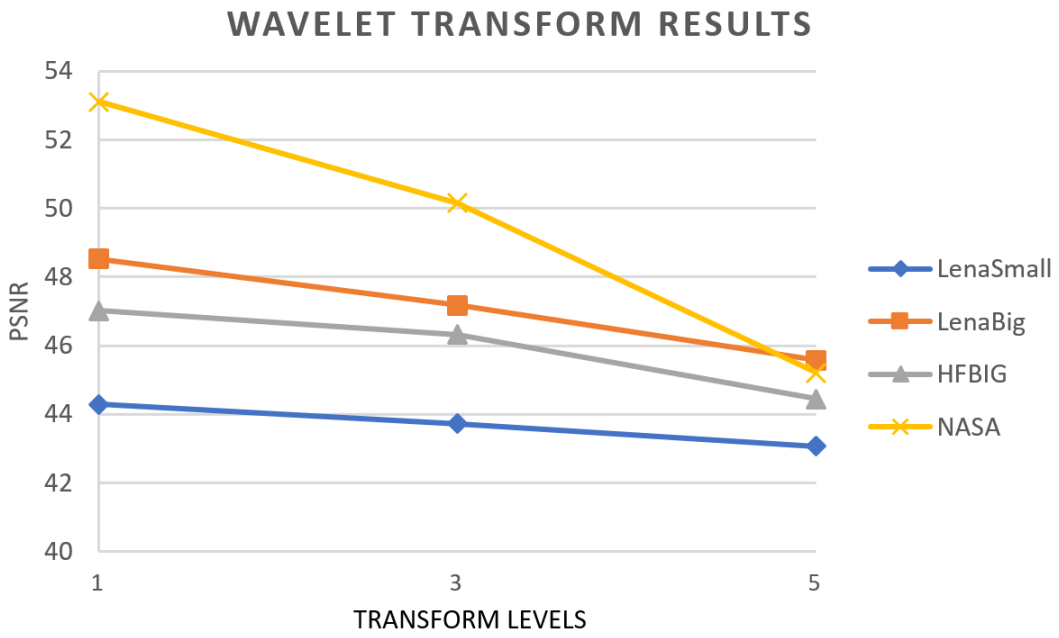

Figure 5.1: The PSNR of the reconstructed images after being wavelet transformed by the wavelet transform module at different levels.

As Figure 5.1 shows, the PSNR is shown to decrease as a function of the transform level for all the test images. This result can be attributed to the nature of the half-precision floating point format. From Section 2.5, it was shown that the dynamic range of the wavelet coefficients also increases as a function of the transform levels. As was explained in Section 2.10.1, the accuracy of the floating point number decreases as the numerical value of the number increases. Following this, an increase in the dynamic range of the wavelet coefficients result in a loss of accuracy of the same coefficients. It is this loss of accuracy that most likely accounts for the loss of PSNR seen as the transformation level increase. This has an impact on the rate of which an image can be compressed, as some PSNR is "wasted" on the wavelet transform process instead of on reducing the information content of the image.

To reduce the losses incurred during the wavelet transform process, the transformation level would either have to be reduced or a different floating-point of fixed-point number scheme would have to be implemented. For instance, the single-precision

floating-point format has a far higher accuracy over a larger dynamic range than the half-precision format. The disadvantage of the single-point format is a greater requirement of hardware resources, and in all likelihood no effective way to perform the floating-point arithmetic in a single clock cycle. The more probable candidate to improve the module would be to implement the fixed-point format, where the precision can be chosen arbitrarily by simply allocating more bits to the format. Either change would require a major revamp of the wavelet transform module.

5.2.2 The lifting scheme architecture

The lifting scheme as outlined in Section 2.5.2 is implemented as explained in Section 3.7.1. It uses 6 floating point multipliers, 8 floating point adders and 9 delay registers. From the way floating point arithmetic is implemented, this leads to a rather hardware costly implementation of the architecture, but with the benefit of a relatively low complexity control logic and fast processing. An alternative implementation aimed at reducing the hardware cost would be to modify the architecture by exploiting the fact that the predict and update stages performed in the 9/7 CDF lifting scheme are identical except for a difference in constants. This means that the current architecture could be "cut in half", by running the samples through the first half twice with the appropriate constants. This would reduce the number of floating point multipliers to 4 and floating point adders to 4, but at the expense of increased processing speed and control logic complexity. Figure 5.2 shows a concept architecture for a folded architecture variant for the lifting scheme.

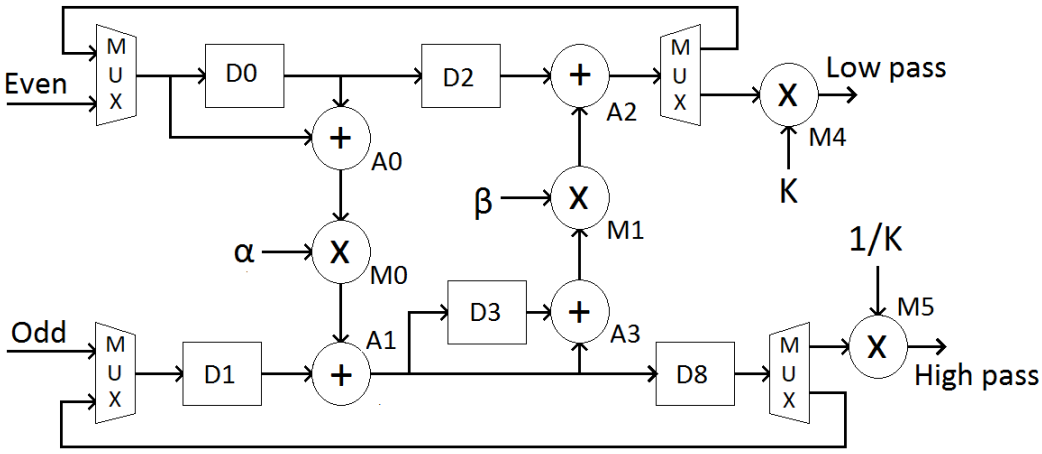

Figure 5.2: Concept folded lifting scheme architecture for reduced hardware cost.

5.3 Quantization module

The quantization module performs scalar quantization using half-precision floating-point multiplication of the wavelet coefficients with a reciprocal of the stepsize chosen for that particular subband. This means that one is free to select individual stepsizes for all the subbands in the wavelet decomposition for all three color components of the tiles. The JPEG2000 standard does not define or recommend any specific stepsizes to use in the quantization step, which makes selecting the appropriate stepsizes up to the designer of the compression system. As was seen in both the theory from Section 2.7 and the implementation from Section 3.8.3, a larger step size leads to a potential higher rate of compression at the cost of quality. The problem then becomes selecting an appropriate stepsize which maximizes the quality of the reconstructed image and at the same time maximizes the potential compression ratio. The optimal stepsizes might also depend on the motive of the image captured by the image sensor, which means that the when determining the best suited step sizes images from the expected operating environment of the camera should be used. This translates to pictures of the earth from Low Earth Orbit (LEO) for the NUTS CubeSat satellite. Although there could be multiple ways to determine the optimal stepsizes, one method would be to use a simple brute force approach where an image is quantized with all possible combinations of stepsizes while the PSNR of the reconstructed image and the compression rate is noted for each combination. This approach would naturally take a considerable time to simulate, but would in all likelihood yield the most suitable stepsizes to achieve a chosen compression rate.

5.4 Utilized hardware resources

From the implementation of the JPEG2000 compression system in VHDL from Section 3 it was found that the system required a significant amount of hardware resources. In total, 35 DSPs, 72 BRAMs and over 14,000 slice LUTs is required to implement this solution on an FPGA. As has been mentioned previously, the reasons for this large expenditure of resources is attributed to several factors:

- Performing floating-point arithmetic in one clock cycle.
- Operating on tile sizes of 128x128 pixels.
- Processing all three color components of the tile simultaneously.
- No intermediate storage of data on the external RAM.

It is the floating-point arithmetic that requires the most amount of DSPs and slice LUTs in the entire implementation. Changing the arithmetic to a multi-clock cycle operation would massively reduce the amount of hardware resources required. Also, since all three color components are processed at once, the wavelet transform module requires three identical copies of the lifting architecture which contains most of the

floating-point arithmetic. By modifying the entire compression system to operate on just one tile component at a time instead of all three, the overall hardware resources needed could in all likelihood be reduced to one third of the current cost. On the downside, this would cause the compression process to take three times longer. It would also require the other color components from the demosaicing module to be stored on the external RAM as they are not used immediately.

The use of internal BRAM modules can also be greatly reduced by relying more heavily on the external RAM instead. By storing intermediate data on the external RAM, the need for the massive memory modules in both the wavelet transform module and the quantization module is removed. The downside is an increase in latency to retrieve the data from the external RAM when compared to storing it internally. In addition, this might make it hard to process the three color components at once, as the current implementation requires that samples from each color component is read concurrently from the internal memory modules. If the samples are stored on a single external RAM module, it will most likely not be possible to read three memory locations at once. This could possibly be remedied by having several small external RAMs on the camera module, allowing the FPGA to access all the RAMs at once.

5.5 Future Work

This thesis has seen the VHDL implementation and testing of the JPEG2000 compression system up to the encoding section of the system. The next natural step would then be to implement the encoding part and test the system as a whole. This would also involve making a MATLAB or similar implementation of the encoder to act as a reference to verify the functionality of the VHDL implementation.

Methods to improve the wavelet transform module can also be investigated, both in terms of hardware resources needed and the accuracy of the transformation. To achieve this, a folded lifting scheme architecture as explored in Section 5.2.2 or an improved floating-point/fixed-point number format as detailed in Section 5.2.1 can be used.

In order to implement a proper compression rate control, optimal stepsizes for the quantization module also has to be determined. It is also yet to be determined if the stepsizes and subsequent compression rate is dependent on the motif of the images, which should also be explored.

Once the complete JPEG2000 compression system is implemented it also has to be tested together with the other systems on the camera module for the NUTS CubeSat. This involves developing the Xilinx MicroBlaze softcore microprocessor to control the compression system and the external MCU which serves as the communication link between the camera module and the rest of the satellite.

6 Conclusion

This thesis presents a partial VHDL implementation of the JPEG2000 compression system which is intended to be used on the camera module for the NUTS CubeSat satellite. In total the system is comprised of over 5000 lines of VHDL code. Due to time constraints, the encoding part of the compression system was only partially implemented. Figure 6.1 shows the current status of the compression system. Here, the green modules represents the parts of the system that has been implemented and tested, while the yellow modules represents the parts which have only been partially implemented.

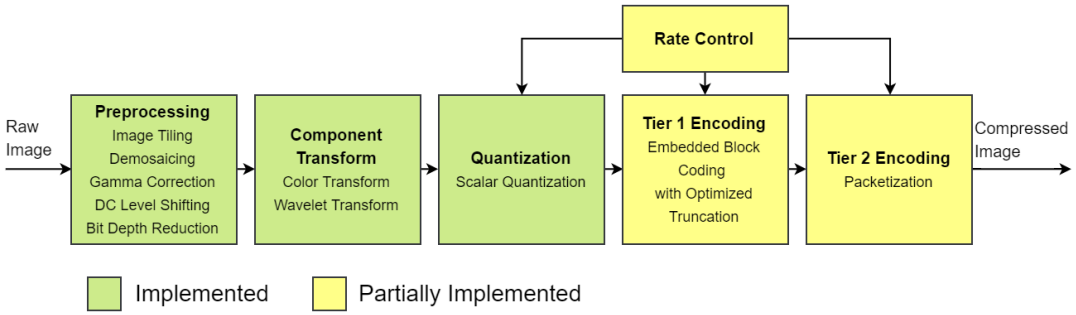

Figure 6.1: The implemented parts of the JPEG2000 compression system.

The VHDL implementation of the compression system is divided into 6 modules: the demosaicing module, the gamma correction module, the intercomponent transform module, the wavelet transform module, the quantization module and the encoding module. The demosaicing module converts the raw Bayer encoded image captured by the image sensor on the camera module into an RGB image. It operates on one 128x128 pixel tile of the raw image at a time to reduce the internal memory overhead. The gamma correction module performs gamma correction and bit depth reduction on the RGB image tile from the demosaicing module. Then the intercomponent transform module performs an irreversible color transform and DC level shift on the image tile. The tile is then wavelet transformed using the CDF 9/7 irreversible discrete wavelet transform in the wavelet transform module, before it is scalar quantized with a variable stepsize in the quantization module. The quantized wavelet coefficients is then transmitted to the encoding module, which was intended to perform the function of the JPEG2000 Tier 1 and Tier 2 Encoder, which performs an EBCOT and MQ encoding of the tile as well as a final packetization of the encoded bit stream into the JP2 file format. The encoding module is only partially implemented due to time constraints.

Because of the scale of a project involving the implementation of a complete compression system the focus was not on developing the most efficient implementation, but

rather on establishing a baseline modular compression system in which each module could be improved upon in subsequent work. As was discussed in Section 5 there are some areas of the system which could be improved, one being the hardware resources required by the implementation and the other being the accuracy of the wavelet transformation.

Table 6.1 shows the required hardware resources to implement the developed compression system on an FPGA using Xilinx Vivado. As is seen, the wavelet transform module is responsible for the majority of the hardware resources required by the system primarily due to the need for floating-point arithmetic operations.

Table 6.1: The hardware resources utilized by the implemented modules.

Module	**Slice LUTs**	**Slice REGs**	**F7 MUXs**	**BRAM**	**DSP**
Demosaicing module	891	365	0	0	8
Gamma correction module	214	24	37	0	0
Intercomponent transform module	576	24	0	0	0
Wavelet transform module	11539	1966	0	48	24
Quantization module	898	275	0	0	3
Encoding module	198	217	0	24	0
Total	14325	2902	37	72	35

From the testing performed in Section 4 it was determined that all the modules, with the exception of the incomplete encoding module, performed as expected. The decision to use half-precision floating-point arithmetic in the wavelet transform module caused some loss of precision in the resulting wavelet coefficients, which in turn leads to a slight loss of image quality depending on the transformation level used. This loss can be avoided by choosing a format with higher precision than the half-precision floating-point, such as a 20-bit fixed-precision or higher format. Table 6.2 shows the processing time for a tile and for a 2592x1944 pixel image.

Table 6.2: The clock cycles and processing time required to process a tile and full image at 50 MHz.

	Number of tiles	**Clock cycles**	**Total time**
Single tile (128x128)	1 (per color component)	163,800	3.276 ms
Raw image (2592x1944)	336 (per color component)	55,036,800	1096 ms

From the test results gathered in Section 4 it was found that the maximum operating speed of the compression system is currently around 63.5 MHz. At this speed, a full image of 2592x1944 pixels could be processed in around 866.7 ms. With the encoder completed the system will in all likelihood be able to compress the raw images from the image sensor on the camera module effectively and reliably.

Bibliography

[1] P. A. Rønning. A study of hardware compression of images. *NTNU Master Thesis*, 2016.

[2] NASA. *CubeSats Deployed From the International Space Station*. 2016. URL: https://www.nasa.gov/image-feature/cubesats-deployed-from-the-international-space-station.

[3] Cal Poly SLO "The CubeSat Program". Cubesat design specification rev. 13, Accessed: 21.06.2017. URL:http://www.cubesat.org/resources/.

[4] NUTS. Ntnu test satellite: A norwegian cubesat project, Accessed: 21.06.2017. URL: http://nuts.cubesat.no/.

[5] Snorre Stavik Rønning. Optimizing an infrared camera for observing atmospheric gravity waves from a cubesat platform, 2012.

[6] ON Semiconductor. Mt9p031: 5 mp 1/3" cmos image sensor, 2017. URL: http://www.onsemi.com/PowerSolutions/product.do?id=MT9P031.

[7] Martin Gammelsæter Håkon Sveinssønn Mork Antoine F.X. Pignède Stian Solberg Nicolas Oppheim Bakkebø, Eirik Lund Flogard. Prosjektrapport – bildekomprimering. *TFE4850 Eksperter i team Romteknologi*, 2014.

[8] Jon Kalevi Oltedal. Development of a prototype of a candidate camera payload. *NTNU Master Thesis*, 2016.

[9] Jon Kalevi Oltedal. Review of the hardware description of the camera module prototype for ntnu test satellite (nuts). *Specialization Project*, 2015.

[10] Xilinx. Spartan-6 fpga family, 2017. URL: https://www.xilinx.com/products/silicon-devices/fpga/spartan-6.html.

[11] Xilinx. Microblaze soft processor core, 2017. URL: https://www.xilinx.com/products/design-tools/microblaze.html.

[12] Andreas Bertheussen. Digital processing system for a cubesat camera. *Specialization Project*, 2014.

[13] Dwight Hooker. Playmate of the month. *Playboy Magazine*, November 1972.

[14] Michael D. Adams. The jpeg-2000 still image compression standard, 2005.

[15] C. Christopoulos, A. Skodras, and T. Ebrahimi. The jpeg2000 still image coding system: an overview. *IEEE Transactions on Consumer Electronics*, 46(4):1103–1127, 2000.

[16] David S Taubman and Michael W Marcellin. *JPEG2000 Image Compression Fundamentals, Standards And Practice*. Springer US, 1 edition, 2002.

[17] Rico Malvar, Li-wei He, and Ross Cutler. High-quality linear interpolation for demosaicing of bayer-patterned color images. *International Conference of Acoustic, Speech and Signal Processing*, May 2004.

[18] Deane Brewster Judd and Gunter Wyszecki. *Color in business, science, and industry*. Wiley, 1 edition, 1975. p.388.

[19] ISO/IEC JTC 1/SC 29. Information technology – jpeg 2000 image coding system: Core coding system. URL: https://www.iso.org/standard/70018.html.

[20] ITU-T. T.871: Information technology – digital compression and coding of continuous-tone still images: Jpeg file interchange format (jfif). *SERIES T: TERMINALS FOR TELEMATIC SERVICES, T.850–T.899*, 2012.

[21] Alfred Haar. Zur theorie der orthogonalen funktionensysteme. *Mathematische Annalen*, 69(3):331–371, 1910.

[22] I. Daubechies and C. Heil. Ten lectures on wavelets. *Computers in Physics*, 6(6):697, 1992.

[23] A. Cohen, Ingrid Daubechies, and J.-C. Feauveau. Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, 45(5):485–560, 1992.

[24] Pascal Getreuer. Wavelet cdf 9/7 implementation, 2006. URL: http://www.getreuer.info/home/waveletcdf97.

[25] S. G. Mallat. Multifrequency channel decompositions of images and wavelet models. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(12):2091–2110, Dec 1989.

[26] S.G. Mallat. A theory for multiresolution signal decomposition: the wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, 1989.

[27] Wim Sweldens. The lifting scheme: A construction of second generation wavelets. *SIAM Journal on Mathematical Analysis*, 29(2):511–546, 1998.

[28] I. Daubechies A. R. Calderbank and W. Sweldens. Wavelet transforms that map integers to integers. *Applied and Computational Harmonic Analysis*, 5(3):332–369, 1998.

[29] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. *Journal on Fourier Analysis and Applications*, 4(3):247–269, 1998.

[30] D. Taubman. High performance scalable image compression with ebcot. *IEEE TRANSACTIONS ON IMAGE PROCESSING*, 9(7):1158–1170, 2000.

[31] JPEG. Proposal of the arithmetic coder for jpeg2000. *ISO/IEC JTC1/SC29/WG1 N762*, 1998.

[32] IEEE. Ieee standard for floating-point arithmetic, 2008. URL: http://ieeexplore.ieee.org/document/4610935/.

[33] Peter Westerink, Inald Lagendijk, and Jan Biemond. *VCDemo*. TU-Delft/MSP, 2004. URL: http://insy.ewi.tudelft.nl/content/image-and-video-compression-learning-tool-vcdemo.

[34] Xilinx. Xilinx vivado design suite, 2016. URL: https://www.xilinx.com/products/design-tools/vivado.html.

[35] Numato Lab. Mimas – spartan 6 fpga development board, 2017. URL: https://docs.numato.com/doc/mimas-spartan-6-fpga-development-board/.

[36] Avrumw. Re: Synthesis taking too long ... URL: https://forums.xilinx.com/t5/Welcome-Join/Synthesis-taking-too-long/td-p/653146 Accessed: 20.05.2017.

[37] Cambridge in Color. Understanding gamma correction. URL: http://www.cambridgeincolour.com/tutorials/gamma-correction.htm.

Appendix A
Code repository

All the relevant code used in this thesis can be downloaded in ZIP format from the online repository: https://bitbucket.org/Okhs/jpeg2000/downloads/JPEG2000.rar. Included is the MATLAB scripts used to verify the functionality of the VHDL implementation, as well as the VHDL implementation itself. The VHDL implementation is included as a Xilinx Vivado project named JPEG2000.

Appendix B
Module Interface Reference

This reference document aims to document the connection interface of each module in the VHDL implementation of the JPEG2000 compression system and describe the inputs and outputs of each module.

A.1 JPEG2000 top module

Figure 6.2 shows the connection diagram for the JPEG2000 top module.

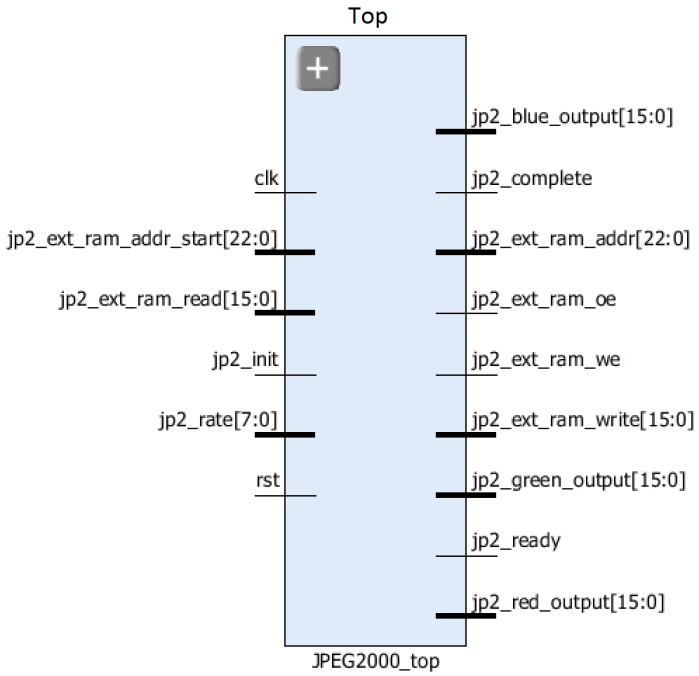

Figure 6.2: Connection interface of the JPEG2000 top module.

Table 6.3 describes the inputs and outputs of the JPEG2000 top module.

Table 6.3: JPEG2000 top module port description.

Port	**Direction**	**Description**
clk	Input	Module clock input.
rst	Input	Module reset. Resets the top module and all internal modules to a default state.
jp2_ext_ram_addr_start	Input	Indicates the start RAM address of the first pixel of the image stored in the external RAM. Supplied either by the MicroBlaze microcontroller or the external MCU.
jp2_ext_ram_read	Input	Data input from the external RAM.
jp2_init	Input	Initiates the compression process. Once initiated, the module will process an image stored in the external RAM.
jp2_rate	Input	Specifies the compression rate. *Currently not implemented.*
jp2_red_output	Output	Outputs compressed tiles from the red color component to the external RAM.
jp2_green_output	Output	Outputs compressed tiles from the green color component to the external RAM.
jp2_blue_output	Output	Outputs compressed tiles from the blue color component to the external RAM.
jp2_complete	Output	Indicates that the compression process has finished.
jp2_ext_ram_addr	Output	Address selection for the external RAM.
jp2_ext_ram_oe	Output	Output enable for the external RAM.
jp2_ext_ram_we	Output	Write enable for the external RAM.
jp2_ext_ram_write	Output	Data output for the external RAM.
jp2_ready	Output	Debugging output. *Currently not used.*

Figure 6.3 shows how the internal modules of the top module are connected. The yellow boxes with "Top module control" indicates signals which are controlled through the state machine in the top module.

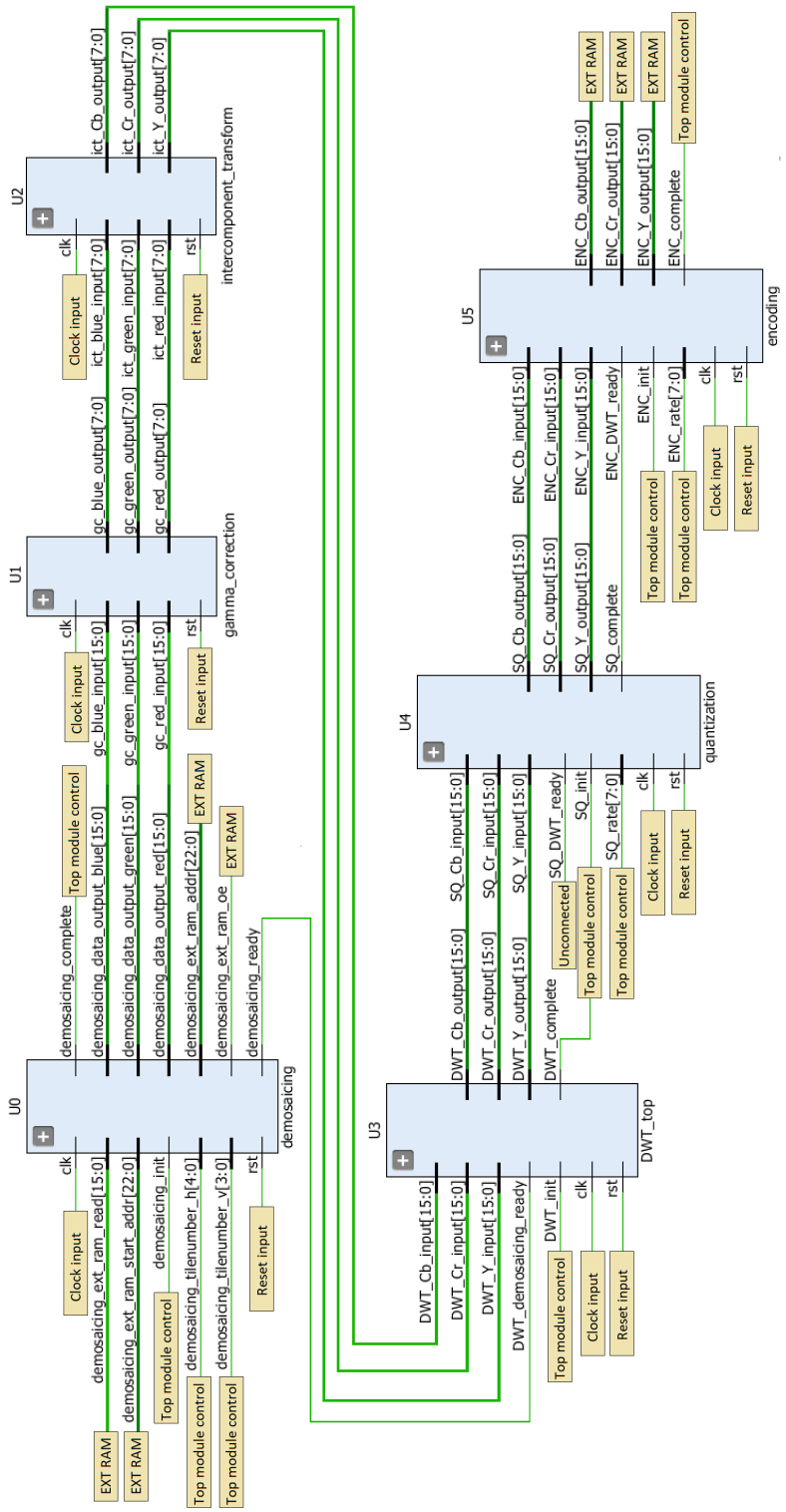

Figure 6.3: Internal module interconnection of the JPEG2000 top module.

A.2 Demosaicing module

Figure 6.4 shows the connection diagram for the demosaicing module.

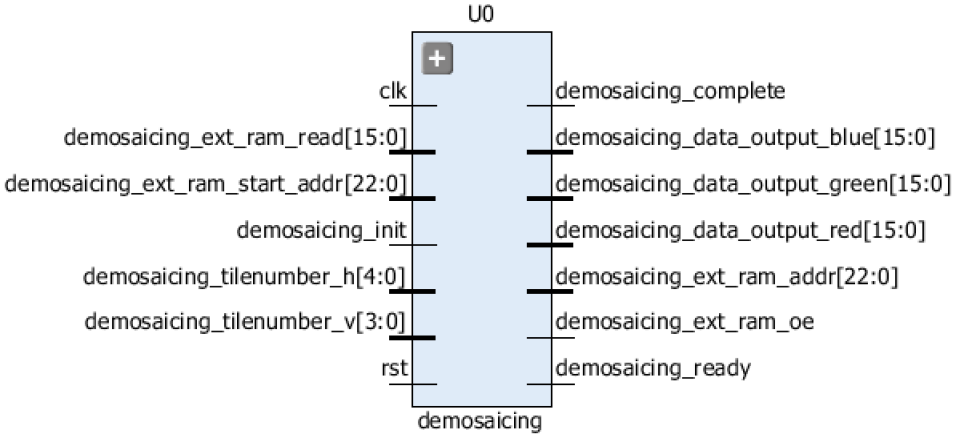

Figure 6.4: Connection interface of the demosaicing module.

Table 6.4 describes the inputs and outputs of the demosaicing module.

Table 6.4: Demosaicing module port description.

Port	Direction	Description
clk	Input	Module clock input.
rst	Input	Module reset. Resets the module to a default state.
demosaicing_ext_ram_addr_start	Input	Indicates the start RAM address of the first pixel of the image stored in the external RAM. Supplied either by the MicroBlaze microcontroller or the external MCU.
demosaicing_ext_ram_read	Input	Data input from the external RAM.
demosaicing_init	Input	Initiates the demosaicing process. Once initiated, the module will process an a tile stored in the external RAM.
demosaicing_tilenumber_h	Input	Specifies the horizontal number of a tile. Used to control which tile is to be processed. Supplied by the top module.
demosaicing_tilenumber_v	Input	Specifies the vertical number of a tile. Used to control which tile is to be processed. Supplied by the top module.
demosaicing_complete	Output	Indicates that the demosaicing of a tile has finished.
demosaicing_data_output_red	Output	Outputs the demosaiced red color component pixels.
demosaicing_data_output_green	Output	Outputs the demosaiced green color component pixels.
demosaicing_data_output_blue	Output	Outputs the demosaiced blue color component pixels.
demosaicing_ext_ram_oe	Output	Output enable for the external RAM.
demosaicing_ext_ram_addr	Output	Address selection for the external RAM.
demosaicing_ready	Output	Indicates that a single pixel has been demosaiced and will be transmitted to the color component outputs.

A.3 Gamma correction module

Figure 6.5 shows the connection diagram for the gamma correction module.

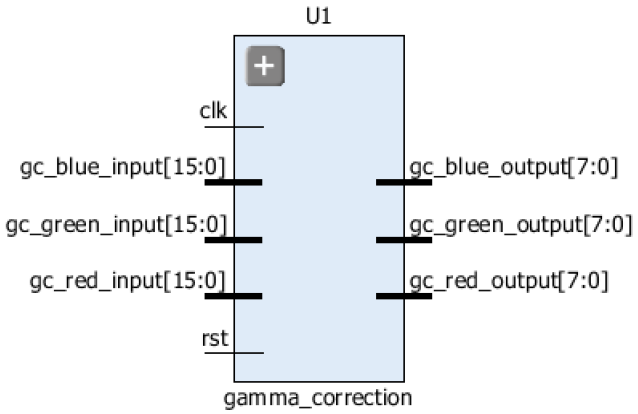

Figure 6.5: Connection interface of the gamma correction module.

Table 6.5 describes the inputs and outputs of the gamma correction module.

Table 6.5: Gamma correction module port description.

Port	**Direction**	**Description**
clk	Input	Module clock input.
rst	Input	Module reset. Resets the module to a default state.
gc_red_input	Input	Pixels from the red color component to be gamma corrected.
gc_green_input	Input	Pixels from the green color component to be gamma corrected.
gc_blue_input	Input	Pixels from the blue color component to be gamma corrected.
gc_red_output	Output	Gamma corrected pixels from the red color component.
gc_green_output	Output	Gamma corrected pixels from the green color component.
gc_blue_output	Output	Gamma corrected pixels from the blue color component.

A.4 Intercomponent transform module

Figure 6.6 shows the connection diagram for the intercomponent transform module.

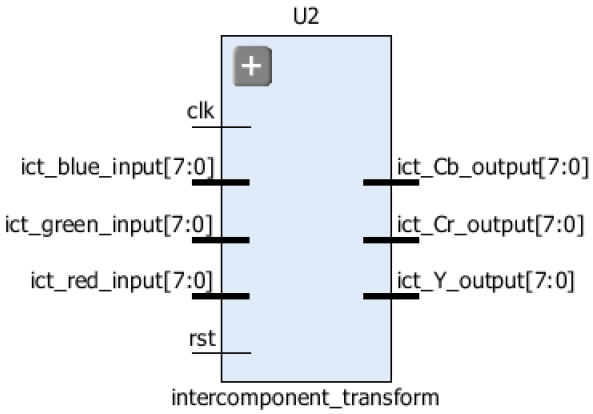

Figure 6.6: Connection interface of the intercomponent transform module.

Table 6.6 describes the inputs and outputs of the intercomponent transform module.

Table 6.6: Intercomponent transform module port description.

Port	**Direction**	**Description**
clk	Input	Module clock input.
rst	Input	Module reset. Resets the module to a default state.
ict_red_input	Input	Pixels from the red color component to be intercomponent transformed.
ict_green_input	Input	Pixels from the green color component to be intercomponent transformed.
ict_blue_input	Input	Pixels from the blue color component to be intercomponent transformed.
ict_Y_output	Output	Intercomponent transformed pixels from the Y color component.
ict_Cb_output	Output	Intercomponent transformed pixels from the Cb color component.
ict_Cr_output	Output	Intercomponent transformed pixels from the Cr color component.

A.5 Wavelet transform module

Figure 6.7 shows the connection diagram for the wavelet transform module.

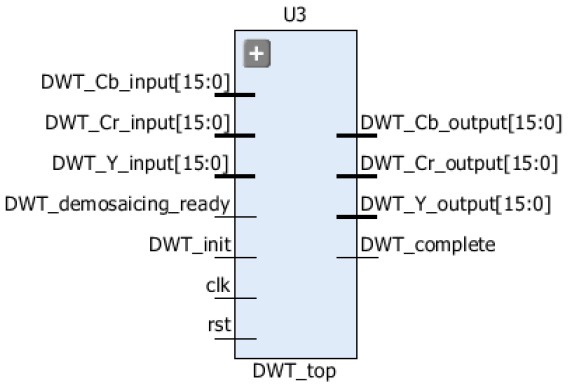

Figure 6.7: Connection interface of the wavelet transform module.

Table 6.7 describes the inputs and outputs of the wavelet transform module.

Table 6.7: Wavelet transform module port description.

Port	**Direction**	**Description**
clk	Input	Module clock input.
rst	Input	Resets the module to a default state.
DWT_Y_input	Input	Pixels from the Y color component to be wavelet transformed.
DWT_Cb_input	Input	Pixels from the Cb color component to be wavelet transformed.
DWT_Cr_input	Input	Pixels from the Cr color component to be wavelet transformed.
DWT_demosaicing_ready	Input	Indicates that a sample has been demosaiced and transmitted.
DWT_init	Input	Initiates the wavelet transform process.
DWT_Y_output	Output	Wavelet transformed pixels from the Y color component.
DWT_Cb_output	Output	Wavelet transformed pixels from the Cb color component.
DWT_Cr_output	Output	Wavelet transformed pixels from the Cr color component.
DWT_complete	Output	Indicates that the wavelet transform process has finished.

A.6 Quantization module

Figure 6.8 shows the connection diagram for the quantization module.

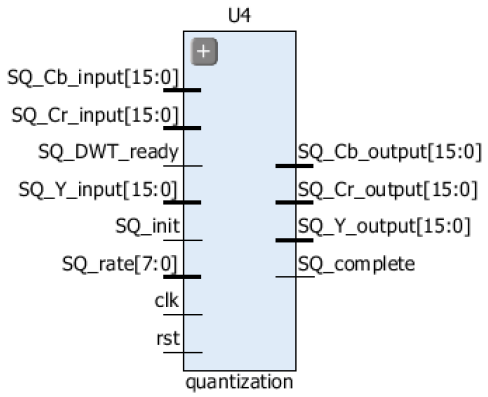

Figure 6.8: Connection interface of the quantization transform module.

Table 6.8 describes the inputs and outputs of the quantization module.

Table 6.8: Quantization module port description.

Port	**Direction**	**Description**
clk	Input	Module clock input.
rst	Input	Resets the module to a default state.
SQ_Y_input	Input	Pixels from the Y color component to be quantized.
SQ_Cb_input	Input	Pixels from the Cb color component to be quantized.
SQ_Cr_input	Input	Pixels from the Cr color component to be quantized.
SQ_DWT_ready	Input	Indicates that the wavelet transform module is transmitting samples. *Currently not used*.
SQ_init	Input	Initiates the quantization process.
SQ_rate	Input	Indicates the compression rate. This affects the quantization step-sizes. *Currently not implemented*.
SQ_Y_output	Output	Quantized pixels from the Y color component.
SQ_Cb_output	Output	Quantized pixels from the Cb color component.
SQ_Cr_output	Output	Quantized pixels from the Cr color component.
SQ_complete	Output	Indicates that the quantization process has finished.

A.7 Encoding module

Figure 6.9 shows the connection diagram for the encoding module.

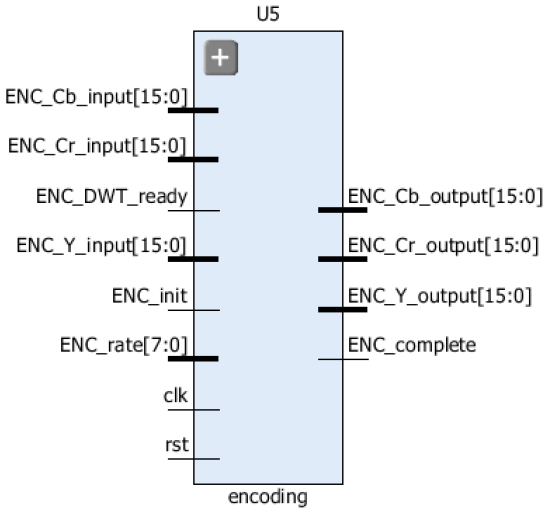

Figure 6.9: Connection interface of the encoding module.

Table 6.9 describes the inputs and outputs of the encoding module.

Table 6.9: Encoding module port description.

Port	**Direction**	**Description**
clk	Input	Module clock input.
rst	Input	Resets the module to a default state.
ENC_Y_input	Input	Pixels from the Y color component to be encoded.
ENC_Cb_input	Input	Pixels from the Cb color component to be encoded.
ENC_Cr_input	Input	Pixels from the Cr color component to be encoded.
ENC_DWT_ready	Input	Indicates that the wavelet transform module is transmitting samples. *Currently not used*.
ENC_init	Input	Initiates the encoding process.
ENC_rate	Input	Indicates the compression rate. This affects the encoding process. *Currently not implemented*.
ENC_Y_output	Output	Encoded tiles from the Y color component.
ENC_Cb_output	Output	Encoded tiles from the Cb color component.
ENC_Cr_output	Output	Encoded tiles from the Cr color component.
ENC_complete	Output	Indicates that the encoding process has finished.