Norwegian University of
Science and Technology

A JPEG2000 Encoder Interpretation

Tore Våland Bø

Master of Science in Electronics
Submission date: June 2018
Supervisor: Bjørn B. Larsen, IES
Co-supervisor: Amund Gjersvik, IES

Norwegian University of Science and Technology
Department of Electronic Systems

Abstract

Students at the Norwegian University of Science and Technology (NTNU) are developing a CubeSat through the Orbit project. The on-board image compression system will be implemented on an FPGA. Continuing from previous work, the Encoder module of the JPEG2000 compression standard is investigated in this thesis. The analysis is the majority of the task, and provides a way for future works.

The Encoder module is sectioned into sub-modules. The MATLAB implementation of the *Partitioner* serves as a bridge between the previous work and the new, by dividing the image into codeblocks. The MATLAB implementation of the *Embedded Block Coding with Optimized Truncation* (*EBCOT*) encodes the codeblocks to produce context labels and decisions. It is connected to the MATLAB implementation of the *Matrix Quantizer* (*MQ*), which produces bitstreams. The output bitstreams are not properly sent, as the unimplemented *Data Orderer* must create and sort the packet locations. Finally, the unimplemented *File Packetizer* inserts file headers and markers to produce a jp2 file. The file codestream will be readable by any decoder.

With an incomplete model of the Encoder module, the VHDL implementation is not yet started. The analysis presented in this thesis should assist future works.

Sammendrag

Studenter ved Norges teknisk- naturvitenskapelige universitet (NTNU) utvikler en CubeSat i Orbit-prosjektet. Bildekomprimeringssystemet ombord i satelliten skal implementeres i en FPGA. Denne fortsettelsen av tidligere arbeid undersøker Encoder-modulen i JPEG2000 standarden for bildekomprimering. Analysen blir hovedparten av oppgaven, og vil vise vei for fremtidige arbeider.

Encoder-modulen er inndelt i undermoduler. MATLAB-implementasjonen av *Partitioner* forbinder tidligere arbeid med dette, og deler bildet i kodeblokker (codeblocks). MATLAB-implementasjonen av *Embedded Block Coding with Optimized Truncation* (*EBCOT*) produserer merkelapper (context labels) og symboler (decisions) for å beskrive situasjonen og avgjørelsen som må tas. *EBCOT* er tilknyttet MATLAB-implementasjonen av *Matrix Quantizer* (*MQ*), som lager bitstrømmer. Bitstrømmene blir ikke sendt riktig, da implementasjonen av *Data Orderer* mangler og følgelig lokasjonen de skal lagres til. Implementasjonen av *File Packetizer*, som også mangler, vil sette inn markørene som utgjør filformatet jp2. Filens kodestrøm vil være leselig for alle dekodere.

Siden modellen av Encoder-modulen er uferdig er heller ikke VHDL-implementasjonen påbegynt. Analysen som presenteres i dette verket vil assistere fremtidige arbeider.

Preface

This work is performed on request for the Orbit project and team at NTNU. They use english as the primary language, to accommodate foreign students, and so this thesis is written in english. Many of the limitations to the scope in this thesis is caused by Orbit's desire to get a functional prototype first, and improve it later.

I had no previous knowledge of Joint Photographic Experts Group 2000 (JPEG2000) prior to this work, which I think have made the progress slower. The main task has been to ”translate” the standard into an understandable extract. I have mostly been working on my own. I would still acknowledge and thank Amund Gjersvik and Bjørn B. Larsen for their guidance and inputs.

Problem Description

Students at the Norwegian University of Science and Technology (NTNU) are developing a CubeSat through the Orbit project, formerly known as NTNU Test Satellite (NUTS). The project gives hands-on experience to students within multi-disciplinary fields of satellite technology. Construction and implementation of the CubeSat should be built from scratch and in-house, as much as possible.

Among other satellite modules, the camera payload module currently needs to be built. Its purpose is to capture, compress and store images while in low earth orbit. The compression system is desired for reducing on-board storage and transmission time to earth. The JPEG2000 compression standard will be used. Both a MATLAB and a VHDL implementation exist for the first few modules of the JPEG2000 compression system, from previous work.

The student's task will concentrate on investigating and implementing the remainder of the compression system. This consists of the Encoder module. Tasks will include completing the MATLAB processing chain, completing the VHDL processing chain, and verifying the complete system. The MATLAB model will be used as reference for creating and validating the VHDL implementation. The Encoder module should then be created in VHDL for the FPGA.

Table of Contents

Abstract **i**

Sammendrag **ii**

Preface **iii**

Task Description **iv**

Abbreviations **viii**

1 Introduction **1**

2 Background **3**
2.1 JPEG2000 . . . 3
2.2 Design Choices from Previous Work . . . 4

3 Theory **5**
3.1 JPEG2000 Encoder Module . . . 5
3.1.1 Tier 1 Encoder . . . 5
3.1.2 Tier 2 Encoder . . . 6
3.2 Image Parts Formed by JPEG2000 . . . 6
3.2.1 Tiles . . . 7
3.2.2 Sub-bands . . . 7
3.2.3 Precincts . . . 8
3.2.4 Codeblocks . . . 8
3.2.5 Bitplanes . . . 9
3.2.6 Coefficients . . . 9
3.2.7 Packets . . . 9
3.2.8 Layers . . . 9
3.3 T1 Partitioner . . . 10
3.4 T1 EBCOT Coder . . . 10
3.4.1 Contexts Labels and Decisions . . . 10
3.4.2 Significance States . . . 11
3.4.3 Scan Pattern . . . 11
3.4.4 Sliding Window . . . 11

3.4.5 Coding Pass Order . . . 12
3.4.6 Significance Propagation Pass . . . 12
3.4.7 Magnitude Refinement Pass . . . 13
3.4.8 Cleanup Pass . . . 13
3.4.9 Sign Coding . . . 14
3.5 T1 MQ Coder . . . 16
3.5.1 Value Explanations and Ranges . . . 16
3.5.2 MQ Functions In Flowcharts . . . 17
3.6 T2 Data Orderer . . . 17
3.7 T2 File Packetizer . . . 18
3.7.1 Headers . . . 18
3.7.2 Markers . . . 18

4 Design Exploration **19**
4.1 Encoder . . . 19
4.2 Image Parts . . . 19
4.3 EBCOT . . . 20
4.3.1 Coding Pass Roles . . . 20
4.3.2 Coding Methods . . . 21
4.3.3 Trivial Parts of Implementation . . . 22
4.3.4 Read and Write Observations . . . 22
4.4 MQ . . . 22
4.5 Investigating the Partitioner and Data Orderer . . . 23
4.5.1 Data Orderer Considerations . . . 23
4.5.2 Memory Considerations . . . 24
4.6 Partitioner . . . 24
4.7 Data Orderer . . . 25
4.8 File Packetizer . . . 25

5 MATLAB Implementation **27**
5.1 EBCOT . . . 27
5.1.1 Coding Passes . . . 29
5.1.2 Scan Pattern . . . 30
5.1.3 Significance Propagation Pass . . . 31
5.1.4 Significance Coding . . . 31
5.1.5 Sign Coding . . . 32
5.1.6 Magnitude Refinement Pass . . . 33
5.1.7 Magnitude Refinement Coding . . . 33
5.1.8 Cleanup Pass . . . 33
5.1.9 Normal Cleanup Coding . . . 35
5.1.10 Run-length Cleanup Coding . . . 35
5.2 MQ Coder . . . 37
5.3 Verification . . . 38

6 Discussion **39**
6.1 Verification . . . 39
6.1.1 Partitioner . . . 40
6.1.2 EBCOT . . . 40
6.1.3 MQ . . . 40
6.2 Unfinished Items . . . 40
6.2.1 Unimplemented Items . . . 41
6.2.2 Partially Implemented Items . . . 41
6.3 Thoughts On JPEG2000 Standard . . . 41
6.4 Improvements . . . 42
6.4.1 EBCOT . . . 42
6.4.2 EBCOT In Parallel . . . 42
6.4.3 Coding Pass Optimizations . . . 43
6.4.4 MQ . . . 44
6.4.5 Stream Sorting . . . 44

7 Conclusion **47**

8 Future Work **49**

Bibliography **50**

A Source Files **i**

Abbreviations

1D	= One-Dimensional.
2D	= Two-Dimensional.
COC	= Coding style Component.
COD	= Coding style Default.
CubeSat	= Cube Satellite.
DWT	= Discrete Wavelet Transform.
EBCOT	= Embedded Block Coding with Optimized Truncation.
EOC	= End Of Codestream.
FPGA	= Field-Programmable Gate Array.
FSM	= Finite State Machine.
HH	= High-pass High-pass frequency.
HL	= High-pass Low-pass frequency.
JPEG2000	= Joint Photographic Experts Group (year 2000).
LH	= Low-pass High-pass frequency.
LL	= Low-pass Low-pass frequency.
LPS	= Less Probable Symbol.
LSB	= Least Significant Bit(s).
MPS	= More Probable Symbol.
MQ	= Matrix Quantizer coder.
MSB	= Most Significant Bit(s).
NTNU	= Norwegian University of Science and Technology.
NUTS	= NTNU Test Satellite.
QCD	= Quantization Default.
RGB	= Red, Green and Blue.
SIZ	= Image and tile Size.
SOC	= Start Of Codestream.
T1	= Tier 1 Encoder.
T2	= Tier 2 Encoder.
VHDL	= Very high speed integrated circuit Hardware Description Language.

Chapter 1

Introduction

This thesis regards the encoder of a JPEG2000 compression system. The system is part of a satellite payload whose purpose is to capture and compress images while in low earth orbit. The satellite, a CubeSat, is under development by Orbit, a student organization at the Norwegian University of Science and Technology (NTNU).

The payload consists of an image capture system and an image compression system. The former is performed by an on-board camera that produces raw RGB format images. The latter is performed by an FPGA implementation of the JPEG2000 compression standard [1].

The images are compressed to reduce storage size in-flight, as well as transmission time during flyover near the ground station. In general, the satellite has excess time available for such processing, as it is not in transmission range at all times. The processing must however not exceed the energy budget, as determined by the solar panels and battery capacity. It must have a low enough instantaneous power draw to avoid overheating. These considerations must be inspected during the later prototype stages.

An FPGA implementation of JPEG2000 is partially developed [2]. The remaining parts are the Encoder module and the Rate Control module. It also needs an interface for communicating with the satellite and initiating compression execution. The Encoder module is the topic of this report.

The following chapters 2 and 3 investigates and analyzes the previous work and the JPEG2000 standard. They will provide an overview of the tasks of the Encoder module, as well as the types of image pieces used in it. The specific sub-modules are then elaborated in detail.

The approach and analysis is performed in Chapter 4. It describes parts of the MATLAB implementation, both as separate sub-modules and as system parts of the Encoder module. The highlights from the implementation is then shown and commented in Chapter 5. The complete code is found in Appendix A. A discussion of the situation at the end of this work is presented in Chapter 6, along with possible improvements both for current software solution and future hardware solution.

Chapter 2

Background

2.1 JPEG2000

Joint Photographic Experts Group 2000 (JPEG2000) [1] is an ISO/IEC standard for image compression. A typical compression can convert an input image to an output jp2 file. The compression execution has several stages, which can be divided into modules [1]. The modules [2] are shown in Figure 2.1.

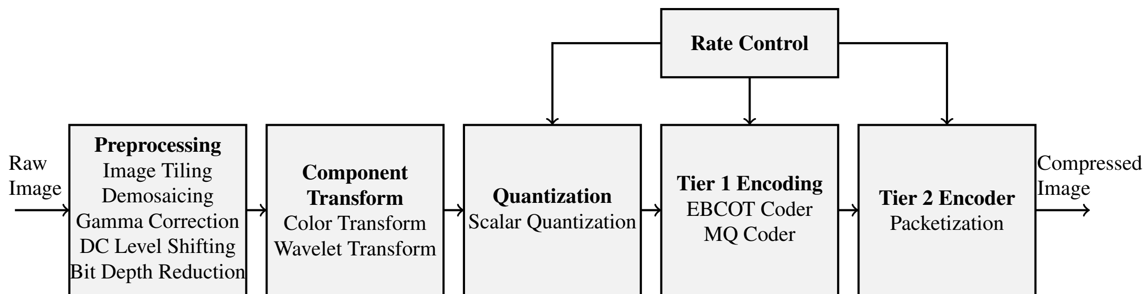

Figure 2.1: Overview of the modules of the JPEG2000 compression system. [2]

The Preprocessing module performs image adjusting operations such as image tiling, demosaicing, gamma correction, DC level shifting and bit depth reduction [2]. The Component Transform module executes colour transform and Discrete Wavelet Transform (DWT) on the image [2]. The Quantization module collects coefficient values in bins through a scalar quantization process [2]. The Preprocessing, Component Transform and Quantization modules have been implemented and tested in VHDL codein a previous work [2]. The operations used in these modules will not be elaborated in this thesis, but some details on their resulting image pieces are shown in the first subsections of Section 3.2. The resulting image pieces after these three modules are quantized sub-band coefficients.

The Tier 1 Encoder (T1) and Tier 2 Encoder (T2) modules are the focus of this thesis, collectively referred to as the Encoder. Here the Embedded Block Coding with Optimized Truncation (EBCOT), Matrix Quantizer coder (MQ) and Packetizer operations execute. The Encoder is described further in Section 3.1.

The Rate Control module adjusts the compression rate used in an execution [2]. The Rate Control module, as well as an overall interface module, remain for future work.

2.2 Design Choices from Previous Work

Some design choices from the previous [2] implementation of JPEG2000 must be accounted for in this and future work.

The processing of tile-components diverge [2] from the norm of the JPEG2000 standard. They are processed in another order from normal JPEG2000 processing. In normal processing order the tile components of a single colour component are processed first. For an RGB image, all red tile components are processed, then all green tile components and finally the blue. [2]

Because of the VHDL implementation [2] of the Demosaicing module, all tile components of a single tile are processed first. For an RGB image, a red tile component, then a green and finally a blue component is processed. This is repeated for each tile. This is a measure to reduce memory traffic between external memory and the Demosaicing module. The consequence is one of two alternatives. The first option is for the Encoder module to reorder tile components into the normal order. The second option requires building a specialized decoder for this system, thus not truly conforming to the JPEG2000 standard. [2]

Chapter 3

Theory

3.1 JPEG2000 Encoder Module

The Encoder continues the JPEG2000 compression execution of the image data from the previous modules (see Figure 2.1). The incoming image data is quantized and DWT-ed [2]. The output is a file of the jp2 format. The file can be used to reconstruct the image.

JPEG2000 specifies an encoder algorithm [1]. It can be categorized in T1 and T2 [2], covered in Section 3.1.1 and Section 3.1.2. T1 compresses the intermediate image data into a bitstream. T2 sorts and marks the compressed data, turning it into a meaningful structure; the jp2 file format.

3.1.1 Tier 1 Encoder

The Tier 1 Encoder (T1) has three tasks: Dividing image parts, executing the EBCOT and executing the MQ [2] [1]. The image parts are explained in Section 3.2. The explanation of the dividing procedure is in Section 3.3. The details of the EBCOT are in Section 3.4. Section 3.5 explains the MQ. The following paragraph provides an overview of these items.

The first task of the encoder is to divide the image data into codeblocks. The image data is the incoming quantized, DWT-ed image parts [2]. The EBCOT reads the codeblocks, one bitplane at a time, and creates a sequence of context labels and decisions. The MQ consumes the context labels and decisions, and produces bitstreams of compressed image data. The bitstreams are used in T2. Figure 3.1 and Figure 3.2 show block diagrams for the EBCOT and MQ. [1]

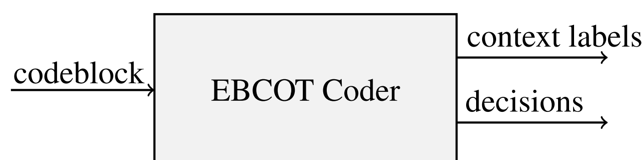

Figure 3.1: Outside view of the EBCOT. [2]

Figure 3.2: Outside view of the MQ. [1]

3.1.2 Tier 2 Encoder

The Tier 2 Encoder (T2) is the final stage of the JPEG2000 compression. T2 has two tasks: Sorting and packetization of the bitstreams [2] [1]. The sorting is covered in Section 3.6. The packetization is explained in Section 3.7. The following paragraph provides a brief overview.

The first task is to sort the bitstreams of compressed image data. These are ordered into packets and layers based on the precincts, the sub-bands and the sorting scheme used. The packetization inserts markers and headers in between the ordered bitstreams. They contain information about sizes and other metadata. This creates a filestream of the jp2 format. [1] [2]

3.2 Image Parts Formed by JPEG2000

The JPEG2000 standard defines several pieces that the original image is split into. The image parts are used in the modules in Figure 2.1. Some of these pieces can be processed separately, such as the tiles [1]. The output jp2 file contains information about all the image parts [1].

The overview descriptions in this paragraph are shown in Figure 3.3. The original, uncompressed image is split into rectangular tiles. The tiles are non-overlapping. The tiles are further divided into their colour components. Tile components are the basic blocks of the compression execution. An RGB image would yield three grayscale tile components, for each tile. The tile components are decomposed in a DWT. This extracts different frequencies into sub-bands. The sub-bands are quantized. Each sub-band can have separate stepsizes in the quantization process [2]. The pieces in this paragraph are used in the other modules of JPEG2000, prior to the Encoder module. [1]

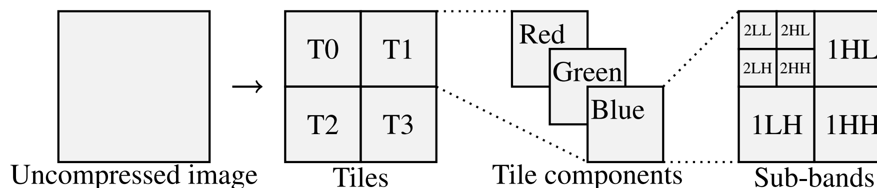

Figure 3.3: The figure illustrates zooming in on the many image parts defined in JPEG2000.

Figure 3.4 shows the pieces in this paragraph, which are used in the Encoder module. The sub-bands are organized into precincts. The precincts are split into codeblocks. The codeblocks are separated into bitplanes. A codeblock consists of coefficients. These are referred to as coefficient bits in the bitplanes. [1]

The bitstreams of encoded codeblock data are collected in packets. Packets then form layers. Markers and headers are inserted, yielding a codestream of the jp2 file format. [1]

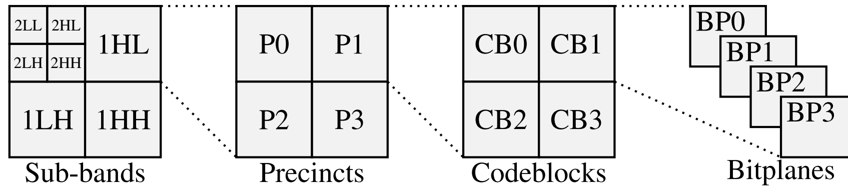

Figure 3.4: The figure illustrates zooming in on the many image parts defined in JPEG2000.

3.2.1 Tiles

Tiles are formed from the original, uncompressed image [2]. They are non-overlapping rectangular shapes of the same size [1]. Tile sizes are in the range from 1 to $2^{32} - 1$, based on the *SIZ* marker [1]. A power of two simplifies the image splitting in later stages of JPEG2000, because rounding of values are avoided. The sub-bands, precincts and codeblocks use powers of two when dividing the image parts. The number of tiles can range from 1, containing the entire image, to as many as desired [1]. The tile number is of course related to the tile size. Tile sizes of 128×128 are recommended and implemented in VHDL [2].

The colour components are extracted from the tiles, forming a tile component [1]. These have the same sizes as the tiles. The tile components are the basic unit of the JPEG2000 standard, and can be processed individually [1]. For example, an RGB image will have three times as many tile components as tiles, because of the three colour components R, G and B.

3.2.2 Sub-bands

A Discrete Wavelet Transform (DWT) decomposes an input into sub-bands. Figure 3.5 illustrates the concept for a 1-level and 3-level 2D DWT[2]. The sub-bands contains either high-pass or low-pass frequency coefficients, labelled the H and L sub-bands. The sub-bands have half the original size for each dimension [2]

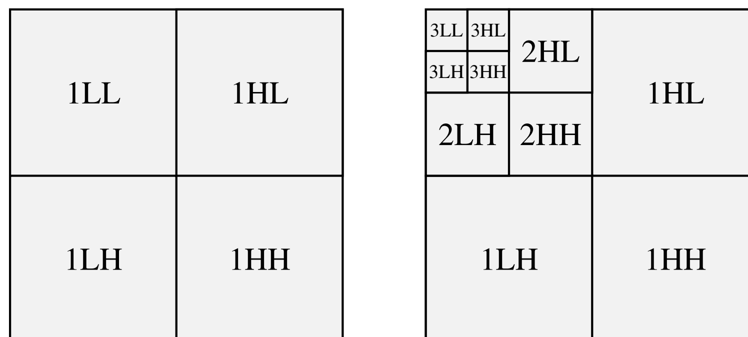

Figure 3.5: The DWT can be repeated on the LL sub-band, creating several levels. This example figure show a 1-level 2D DWT and a 3-level 2D DWT. [2]

A 1D DWT will divide the image into two sub-bands, L and H. A 2D DWT divides this again to produce four sub-bands of $\frac{1}{4}$ the area. The sub-band labels are appended to the 1D labels, resulting in LL, HL, LH and HH. The decomposition can be repeated on the LL sub-band, referred to as the decomposition levels of DWT. JPEG2000 does not specify a preferred amount of decomposition levels, but 4 to 6 is often used. [2]

Every decomposition level contains one of each of the HL, LH and HH sub-bands. The sub-bands are accompanied with a number representing the decomposition level. The amount of decomposition levels in a DWT image is represented by a value N_L. There is always just one LL sub-band. It is referred to as the N_LLL sub-band, because it is in the innermost decomposition level. The decomposition levels begin at 1 in the outer sub-bands and increment inwards. The N_L level contains all four sub-band types, LL, HL, LH and HH. [1]

Resolution levels are similar to decomposition levels. The order is reversed, so the inner HL, LH and HH sub-bands have resolution level 1. The outer have resolution level N_L. The LL sub-band is in a separate level at resolution level 0. This means there is one more resolution level than decomposition levels. [1]

A particular sub-band's dimension sizes are a power of 2 smaller than the tile component's. The *n*-th decomposition level sub-bands have the sizes divided by 2^n, where *n* ranges from 1 to N_L. An image of $x \times y$ will have second decomposition level sub-bands of $x/4 \times y/4$. [1]

3.2.3 Precincts

A precinct is a non-overlapping rectangular piece of a sub-band. The precinct size descriptor *PP* is in the range given in Equation 3.1 [1]. It determines the precinct size, given by Equation 3.2 [1]. The value *PP* is chosen based on preferences such as compression rate and image quality. The choice must be indicated in the *COD* or *COC* markers. *PP* may only be 0 in the $N_L LL$ band. It is limited by the sub-band size and the resolution level. *PP* can be declared in the local markers or the global markers. This means the precincts can have different sizes in different sub-bands. The "default" value for *PP* is 15. [1]

$$0 \leq \begin{matrix} PP_X \\ PP_Y \end{matrix} \leq 15 \tag{3.1}$$

$$(precinct\ width, precinct\ height) = \left(2^{PP_X}, 2^{PP_Y}\right) \tag{3.2}$$

3.2.4 Codeblocks

The codeblock's size descriptor *CB* is given by equation Equation 3.3 [1]. The relation between *CB* and *CB*$'$ (unmarked versus marked) is given by Equation 3.4 [1], where *r* is the resolution level. This essentially means that the codeblocks are limited by the precinct sizes, which are limited by sub-band sizes. Codeblock sizes are given by Equation 3.5 [1]. *CB* ranges from 2 to 10, by choice in *COC* or *COD* markers. The size is restricted so $(CB_X + CB_Y) \leq 12$. [1]

$$2 \leq \begin{matrix} CB_X \\ CB_Y \end{matrix} \leq 10 \tag{3.3}$$

$$\begin{aligned} CB'_X &= \begin{cases} \min(CB_X,\ PP_X - 1) & \text{for } r > 0 \\ \min(CB_X,\ PP_X) & \text{for } r = 0 \end{cases} \\ CB'_Y &= \begin{cases} \min(CB_Y,\ PP_Y - 1) & \text{for } r > 0 \\ \min(CB_Y,\ PP_Y) & \text{for } r = 0 \end{cases} \end{aligned} \tag{3.4}$$

$$(codeblock\ width, codeblock\ height) = \left(2^{CB'_X}, 2^{CB'_Y}\right) \tag{3.5}$$

3.2.5 Bitplanes

The bitplanes are formed from the codeblocks. They are essentially one-bit codeblocks, where every bit comes from the same bit significance position. This means the width and height are the same as the codeblock's. A codeblock with eight bits per coefficient will have eight bitplanes. [1]

3.2.6 Coefficients

A coefficient is essentially a pixel, but has been modified by the processing operations [1]. For instance, the codeblock coefficients can alternatively be referred to as "one quantized and discrete wavelet transformed colour component of a tile's pixel". Within a bitplane they are referred to as coefficient bits [1].

3.2.7 Packets

A packet consists of a packet header and compressed image data. The data consists of parts of bitstreams from encoded codeblocks. The bitstream parts are encoded results from certain bitplanes. There can be none or several bitplanes, and the amount may be different for each codeblock. The packet contains such data from all codeblocks within a precinct. The precinct's position is relative to the sub-band. The same relative position is used in all sub-bands of the current resolution level. Packets are 8-bit aligned. [1]

For example, the packet will contain certain bitplanes' encoded results from all of the following: All codeblocks, in raster order, within *precinct1* in the *2LH* sub-band. Then all codeblocks within *precinct1* in *2HL*, and finally all codeblocks within *precinct1* in *2HH*. This continues for *precinct2* and so on, in raster order, and for all resolution levels.

The packet header contains the information required for decoding the packet. Its first bit describes if the packet is of zero length. The next pieces of information describes which codeblocks are included, zero bitplane information, number of coding passes included, and length of packet data from a given codeblock. The Encoder module may choose to not include compressed data, for higher compression rates, but must still include the packet marked as empty. [1]

3.2.8 Layers

A layer consists of compressed image data from coding passes of codeblocks. A bitstream is distributed across one or more layers. Essentially the layers consist of packets. Layers are structured in such a way as to successively and monotonically increase the reconstructed image's quality. [1]

3.3 T1 Partitioner

The Partitioner sub-module is not directly derived from theory, but is implicitly described [1]. It divides the image parts further, from sub-bands to precincts and codeblocks. The input sub-bands come from an image that has been quantized and DWT-ed. These sub-bands are split into precincts and codeblocks, using the sizes in Equation 3.2 and Equation 3.5. The codeblock outputs are then processed in the EBCOT. The metadata information output is sent to the Data Orderer and Packetizer. This involves the sizes, amount of image pieces created and other things required for reconstructing the image.

3.4 T1 EBCOT Coder

Figure 3.1 shows the block diagram for the Embedded Block Coding with Optimized Truncation (EBCOT). It performs coefficient bit modelling through coding passes on a codeblock. This consists of generating a sequence of context labels and decisions. [1] [2]

The codeblock is encoded one bitplane at a time, in order MSB to LSB. Each bitplane is traversed by the coding passes. The two main contributors in the coding passes are the coefficient bits and the significance states. [1] [2]

Coefficient bits are elements of bitplanes in codeblocks. The currently-pointed-to coefficient bit is encoded in exactly one of the coding passes. Significance states are created for the current codeblock. The significance states that are part of the sliding window affects the encoding results. [1]

Each bitplane is traversed by three types of coding passes: the significance propagation pass, the magnitude refinement pass and the cleanup pass. The coding passes encode or skip coefficient bits according to their rules, as defined in sections 3.4.6, 3.4.7 and 3.4.8. They execute in the order presented in Section 3.4.5. The scan pattern is used once for each pass. The significance propagation pass encodes those predicted to become significance state 1 in this bitplane. The magnitude refinement pass encodes those that have significant state 1 from a previous bitplane. The cleanup pass encodes those that remain. [1]

3.4.1 Contexts Labels and Decisions

The coding passes of the EBCOT produce context labels and decisions that are used by the MQ. The context label is formed based on the contribution from the neighbours in the sliding window. Thus it describes the coefficient bit's relation to the surrounding significance states. [1]

JPEG2000 provides 19 context labels, some not explicitly numbered. The context label formation rules are different for each coding pass. Specific values are detailed in Section 3.4.6, Section 3.4.7 and Section 3.4.8. The values used for the context labels are not important, as long as they are unique and recognizable by the MQ. [1]

The decisions are binary values. In most cases they are the coefficient bits. Exceptions are the run-length and the sign coding, which are explained in Section 3.4.8 and Section 3.4.9. [1]

3.4.2 Significance States

An array of significance states is created when the EBCOT begins execution of a codeblock. The binary array has the same dimension sizes as the codeblock, and is used for this codeblock only. It is a temporary array that describes a property of the codeblock. After the initialization to 0, a significance state can only be changed to 1. It is changed when the first non-zero MSB of a coefficient is found. This update can only occur in the significance propagation pass or the cleanup pass. [1]

At any given point in the coding passes, the significance states indicate which coefficient bits have currently been found to be 1 in any bitplane of the codeblock. The currently-pointed-to significance state is the only value that can be updated. The neighbouring significance states of the coefficient bit affects the context label creation in each coding pass. The neighbours are part of the sliding window. [1]

3.4.3 Scan Pattern

During the coding passes each bitplane is traversed in a specific scan pattern. The scan zigzags through the bitplane in the order shown by the numbering in Figure 3.6. The scan goes in raster order, meaning left-to-right and top-to-bottom; the same as reading this text. However, four vertical elements are scanned for every horizontal step. No elements are scanned twice, so the fifth row is the next line after reaching the width of the array. This continues until the bottom right corner is reached. If the last line does not have four vertical elements, the ”missing” rows are skipped. The 7th and 8th row is ”missing” in Figure 3.6. [1]

0	4	8	12	16	20
1	5	9	13	17	21
2	6	10	14	18	22
3	7	11	15	19	23
24	26	28	30	32	34
25	27	29	31	33	35

Figure 3.6: This example shows the scan pattern in a 6×6 array. The numbers represent the traversal order, starting with zero. [1]

3.4.4 Sliding Window

A sliding window [2], alternatively called a context window [1], is used during the scan pattern in the coding passes. The sliding window consists of a 3×3 array, as shown in Figure 3.7. It is always centered on the current position of the scan pattern. The window slides along, moving as the scan pattern progresses. [2] [1]

The sliding window is only used as a means of conveniently pointing to relative positions. The different labels from Figure 3.7 mark the positions relative to the current position X. The adjacent positions are identified by horizontal H, Vertical V and Diagonal D directions and a unique number. These are also called the neighbours of X. The unique number is only used during sign coding. [1]

D_0	V_0	D_1
H_0	X	H_1
D_2	V_1	D_3

Figure 3.7: The sliding window used during the scan pattern. [1]

The positions are used in three types of arrays, depending on which coding pass is executing. The most common case uses *X* to describe the coefficient bit from the bitplane, and the neighbours to describe the significance state bits. The other case occurs during the sign coding in Section 3.4.9, where each sliding window position describe the sign bits. Thus, the three are the bitplanes, the significance states array and the sign bit array. [1]

When the center *X* is at the edges of the array, the neighbours will attempt to access values outside the border. These are treated in an unobtrusive way. The possible values are significance states and sign bits. Coefficient bits are not accessed, as they are only read from the center position. Significance states are treated as a value of 0. Sign bits are not used, because of the rules for sign coding. [1]

Figure 3.8 shows an example when the scan pattern has reached the sixth element. Again, the numbers represent order of traversal. The sliding window is formed around every position like this. The sliding window contains 1, 2, 3, 5, 6, 7, 9, 10 and 11. [1]

0	4	8	12	16	20
1 D_0	5 V_0	9 D_1	13	17	21
2 H_0	6 X	10 H_1	14	18	22
3 D_2	7 V_1	11 D_3	15	19	23
24	26	28	30	32	34
25	27	29	31	33	35

Figure 3.8: An example of how the sliding window is positioned at a particular position.

3.4.5 Coding Pass Order

The coding passes run in a certain order of appearance. The significance propagation pass is first, the magnitude refinement pass follows and finally the cleanup pass runs. This is repeated for all bitplanes in a codeblock. A coefficient bit is encoded in only one coding pass. [1]

An exception to the order occurs in the first bitplanes. The first empty bitplanes are skipped, empty meaning all elements are 0. The first MSB-plane with a non-zero coefficient bit is scanned only by the cleanup pass. Any empty bitplanes after the first non-zero bitplane are scanned as normal. [1]

3.4.6 Significance Propagation Pass

Prior to the initial cleanup pass the neighbour contribution will be 0. The significance propagation pass will therefore not encode any coefficient bits until after the first MSB-plane.

The significance propagation pass encodes the coefficient bits that are predicted to become significance state 1 in a bitplane. The prediction to become significance state 1 is determined by the neighbour contribution. If any neighbour's significant state is 1, the coefficient bit is encoded in this coding pass. [1]

The coefficient bit is the decision. The context label is formed based on the contribution of the neighbours from the sliding window. Table 3.1 shows the context label formation rules. The most current significance states are used, meaning changes made during this significance propagation pass are included. [1]

LL and LH sub-bands (vertical high-pass)			HL sub-band (horizontal high-pass)			HH sub-band (diagonal high-pass)		Context label
$\sum H_i$	$\sum V_i$	$\sum D_i$	$\sum H_i$	$\sum V_i$	$\sum D_i$	$\sum(H_i + V_i)$	$\sum D_i$	
2	x	x	x	2	x	x	≥3	8
1	≥1	x	≥1	1	x	≥1	2	7
1	0	≥1	0	1	≥1	0	2	6
1	0	0	0	1	0	≥2	1	5
0	2	x	2	0	x	1	1	4
0	1	x	1	0	x	0	1	3
0	0	≥2	0	0	≥2	≥2	0	2
0	0	1	0	0	1	1	0	1
0	0	0	0	0	0	0	0	0

Table 3.1: Context label formation rules for significance propagation pass and cleanup pass. x indicates any input. [1]

If a coefficient bit is 1, the significance state is set to 1. In addition the sign bit is the immediate next encoded bit (see Section 3.4.9). [1]

The adjacent significance states are weighted differently based on whether they are horizontal, vertical or diagonal neighbours of *X*, as seen in Table 3.1. The weights depend on which sub-band the bitplane is within. [1]

3.4.7 Magnitude Refinement Pass

The magnitude refinement pass encodes the coefficient bits that have significant state 1. This includes all remaining bitplanes after the first 1 bit. Significance states do not change in this coding pass. [1]

The coefficient bit is the decision. The context label is formed based on whether the coefficient has been encoded in this coding pass before or not. Table 3.2 shows the formation rules. If this is the first refinement, the neighbour contribution from the sliding window affects the context label. [1]

3.4.8 Cleanup Pass

The cleanup pass encodes the remaining coefficient bits. That is, the coefficient bits in the bitplane that were encoded during neither the significance propagation pass nor the magnitude refinement pass. The cleanup pass thus ensures all coefficient bits in a bitplane are encoded. [1]

$\sum H_i + \sum V_i + \sum D_i$	First refinement for this coefficient	Context label
x	false	16
≥1	true	15
0	true	14

Table 3.2: Context label formation rules for magnitude refinement pass. *x* indicates any input. [1]

The cleanup pass has two types of context label formation rules. The run-length coding runs on four contiguous coefficient bits in the same column. The normal mode operates identical to the significance propagation pass, but without the prediction. The most current significance states are used, meaning changes made during this cleanup pass are also included. To select run-length mode the four coefficient bits must be in the same column, must not have been encoded in previous coding passes in this bitplane, and must have context label 0. [1]

In normal mode the coefficient bit is the decision. The context label is created from the neighbour contribution, using the same Table 3.1 as in the significance propagation pass. [1]

The run-length mode rules are described in Table 3.3. In run-length mode a decision of 0 indicates all four coefficient bits have significance state 0. A decision of 1 indicates at least 1 coefficient bit of the four have significance state 1, and Uniform will be the next encoding. Uniform indicates which coefficient bit of the four have the first occurrence of significance state 1. Uniform has two decisions in a series, which together create a binary number in range 0 to 3. If the first non-zero coefficient bits is not the last of the four, the remaining after it are encoded in normal mode. [1]

The context labels for run-length mode has two possible values. The first decision has a run-length context label, while the two Uniform decisions have a Uniform context label [1] . The values can be arbitrarily selected, as explained in Section 3.4.1. By continuing the numbering the value 17 can be chosen for the run-length context label and 18 for the Uniform context label.

If a coefficient bit is 1, the significance state is set to 1. In normal mode the sign bit is the immediate next encoded bit, following the scheme in Section 3.4.9. In run-length mode the sign bit encoding is delayed until after the Uniform context label and decisions. [1]

3.4.9 Sign Coding

This coding operation is part of the significance propagation pass and the cleanup pass. Whenever they find a coefficient bit which changes the significance state to 1, the sign is encoded. The context label and decision of the sign coding is inserted in the EBCOT's output immediately after the coefficient bit has been encoded in the caller's coding. This means the coding pass that started the sign encoding continues after the sign bit is encoded. Only one coefficient is encoded when sign encoding is started. Throughout the coding passes, every non-zero pixel of the codeblock is sign encoded once. [1]

The sliding window is used to find the contribution of the adjacent significance states. The neighbour positions are additionally used to find the sign bits of the codeblock. The neighbours contribute as given in Table 3.4, with this table being used identically for both horizontal and vertical neighbours. [1]

Four contiguous coefficients in a column remaining to be encoded and each currently has the 0 context	Symbols with run-length context	Four contiguous bits to be decoded are zero	Symbols decoded with UNIFORM context	Number of coefficients to decode
true	0	true	none	none
true	1	false skip to first coefficient sign skip to second coefficient sign skip to third coefficient sign skip to fourth coefficient sign	MSB LSB 00 01 10 11	 3 2 1 0
false	none	x	none	rest of column

Table 3.3: Decision formation rules for run-length mode in the cleanup pass. The second column shows the decision accompanying the run-length context label. The fourth row shows the two decisions accompanying the Uniform context label. A value of *none* indicates that nothing is produced, not even the context label. [1]

V_0 or H_0	V_1 or H_1	V or H contribution
significant, positive	significant, positive	1
significant, negative	significant, positive	0
insignificant	significant, positive	1
significant, positive	significant, negative	0
significant, negative	significant, negative	-1
insignificant	significant, negative	-1
significant, positive	insignificant	1
significant, negative	insignificant	-1
insignificant	insignificant	0

Table 3.4: The contribution from the significant states and sign bits of the neighbouring coefficients. *Significant* and *insignificant* indicates respectively 1 and 0 significance state. This table is the same for H contribution and V contribution, hence the ”or” in the first row. [1]

Horizontal contribution	Vertical contribution	Context label	XOR-bit
1	1	13	0
1	0	12	0
1	-1	11	0
0	1	10	0
0	0	9	0
0	-1	10	1
-1	1	11	1
-1	0	12	1
-1	-1	13	1

Table 3.5: The contributions are gathered from Table 3.4, which yields the context label and XOR-bit for the sign encoding of one coefficient. [1]

The horizontal and vertical contribution form the context label as detailed in Table 3.5. An *XORbit* is also produced by the same table. The *XORbit* is used in the exclusive or operation in Equation 3.6. The *signbit* is gathered from the sign bit of the codeblock, by using the coefficient's position. The *signbit* is 0 for positive numbers and 1 for negative numbers. This forms the decision output from the sign encoding operation. [1]

$$decision = signbit \oplus XORbit \tag{3.6}$$

3.5 T1 MQ Coder

The Matrix Quantizer coder (MQ) is a binary arithmetic coder. This means it performs arithmetic coding, also called entropy coding [2]. It takes a series of context labels and decisions as input and converts them to a bitstream output. The encoded bytes use less bits than the incoming context labels and decisions, which is how the data is compressed [2]. Figure 3.2 shows the block diagram for the MQ. [1]

The input decision symbol is used to determine the coding method. This is based on its probability, which can be the more probable symbol (MPS) or less probable symbol (LPS). Thus the binary decision has two available coding methods to select. [1]

The input context label is used as a reference to a finite state machine (FSM) entry. For 19 context labels there are 19 FSMs. The FSMs are identical for each context label, but they may have achieved a different progress at any time. There are 47 progress states available, for each of the 19 context labels. The FSM serves as a lookup table that provides a *Qe* value and a next-state, *NMPS* or *NLPS*. The lookup values are based on precomputed probability values. The values are rounded to hexadecimal integer representations of decimal values. The precomputed values are described in more detail in the JPEG2000 standard, but these details are not used in this thesis. [1]

The MPS and LPS coding methods use the *Qe* value to modify the two main values: *A*-reg and *C*-reg. *A*-reg influences the path executed within the MQ; essentially a control signal. *C*-reg contains the intermediate results which occasionally form a byte of the bitstream. The current coding method decide which of the two next states is used for the FSM. The FSM also occasionally provides the new MPS and LPS symbols, through its *Switch* value. [1]

The MQ output bytes should be appended to the packet headers. This means the initial MQ byte comparison to `0xFF` compares with the last packet header byte. Any produced MQ bytes with value `0xFF` must be followed by a byte with a zero MSB. This is achieved by a bit-stuffing procedure. At the end of the list of context labels and decisions, the *flush* procedure is executed. It flushes the remaining bits from *C*-reg to the bitstream. [1]

3.5.1 Value Explanations and Ranges

A-reg and *C*-reg are stored in 32-bits registers, divided into sections as seen in Table 3.6. The 'a' and 'x' bits hold intermediate results from arithmetic operations. The 'b' bits hold the byte that will be output. The 's' bits are spacer bits that ensure there is no overflow from the 'x' bits. The 'c' bit is a carry bit that may be added to the previous output byte. [1]

A-register	0000 0000 0000 0000 1aaa aaaa aaaa aaaa
C-register	0000 cbbb bbbb bsss xxxx xxxx xxxx xxxx

Table 3.6: The 32-bits registers for the *A*-reg and *C*-reg variables in the MQ [1].

Table 3.7 shows the value ranges of the essential variables. *Qe* comes from a lookup reference. The value in *A*-reg is forced to always stay within the interval. The operations can decrease its value below the range, which triggers left-shifting until the 16th bit is 1. The value in *C*-reg accumulates in the 16 lower bits (LSBs). The operations can increase its value. *C*-reg is left-shifted every time *A*-reg is left-shifted. When a counter reaches the end of counting, the 'b' bits are extracted to form a byte output. The 'c' bit is prevented from forming a byte, but instead increases the previous byte. *A*-reg is initialized to `0x8000`. *C*-reg is initialized to 0. [1]

Range start		Variable Name		Range end
0x0001	≤	*Qe*	≤	0x5601
0x8000	≤	*A-reg*	≤	0xFFFF
0x0000 0000	≤	*C-reg*	≤	0x0FFF FFFF

Table 3.7: Value range for MQ variables. [1]

3.5.2 MQ Functions In Flowcharts

The JPEG2000 standard shows detailed flowcharts for a recommended MQ implementation [1]. These are too long and too many to list here. They are detailed enough to almost be identical to the implementation; hence see the code in Appendix A. The FSM table is also provided as a function here. The function and variable names are mostly identical to the standard's definitions.

3.6 T2 Data Orderer

The Data Orderer sorts data by precincts, packets and layers. The input bitstreams come from the MQ. The information about tiles, tile components, precincts and similar comes from the Partitioner, and is used to reorder the bitstreams. The reordering allows the bitstreams to be organized in packets, limited by the size of precincts [1]. The packet includes bitstreams from the same precinct locations in other sub-bands of the same resolution level [1]. Packets then form layers [1]. The output is a sorted datastream, sent to the Packetizer.

The bitstreams should not already contain markers from main header, tile headers or *EOC*. However, any packet headers and in-stream markers should be included. [1]

Each layer provides an increase in the reconstructed image's quality [1]. The layers enable the compressed image to be partially reconstructed, i.e. a progressively coded image [1]. The decoder is able to choose how much of the file to reconstruct. The Data Orderer must therefore sort the most important data layers first in the datastream. The progression order determines what is considered as most important.

The progression order is used to determine the order and structure of the file. It is essentially the order of reading the layers, resolution levels, colour components and positions. These can

be arranged in five allowed orders, based on the *COD* marker. The progression order affects how the partially reconstructed image appears. An example is to get a tiny thumbnail, which then grows in size and detail. Another example is to get more and more colours as the image is decoded. [1]

3.7 T2 File Packetizer

The Packetizer receives the ordered datastream from the Data Orderer. Its purpose is to insert markers and headers in between the compressed image data, which describe the characteristics of the image. The output is a compressed image of the jp2 file format. [2]

The markers and headers are well defined in the JPEG2000 standard. They are used to delimit sections of the datastream and make it readable for any JPEG2000 decoder. The header types are one main header and several tile headers. The markers contain metadata that describe properties of the image. Markers bundled together form specific headers. [1]

3.7.1 Headers

Headers consists of markers. The main header begins all JPEG2000 files. The minimum main header consists of *SOC*, *SIZ*, *COD* and *QCD* markers. It can also contain optional markers, some of which can be placed in the tile header instead. [1]

3.7.2 Markers

Markers are sequences of bits that represent a specific meaning. They usually start with a unique label that is reserved for this specific use. If such a unique label is found in the bitstream it is read and decoded as dictated by the appropriate marker. For instance, the 16 bits `0xFF51` indicate the start of the *SIZ* marker. The decoder will then know that the following 328 bits are also part of the *SIZ* marker. These 328 bits then describe image sizes, tile sizes, offset sizes, amount of colour components in image, and more. [1]

The types of markers contain metadata for individual fields. Some of the other metadata that must be described are decomposition levels, amount of empty bitplanes and sizes of different image pieces. [1]

Chapter 4

Design Exploration

This chapter explores solutions to the Encoder. It is based on an analysis of the presented content from Chapter 2 and Chapter 3.

4.1 Encoder

The Partitioner and Data Orderer sub-modules are not defined in the JPEG2000 standard directly. Their tasks are implicitly referenced, for instance by requiring codeblocks for the EBCOT from the sub-bands. Their names are therefore created for this thesis. As there was no Partitioner in the initial stages of this thesis, the first natural step in the implementation was the EBCOT. The EBCOT is defined in the JPEG2000 standard. The approach of implementing the defined sub-module is better than working with many assumptions. The task definitions of the Partitioner and Data Orderer have been uncovered as the work has progressed.

The EBCOT takes codeblocks as input, as Figure 3.1 indicates. The approach to the EBCOT implementation therefore assumes the codeblocks are already formed. The sub-modules are built around this assumption. The relation between the EBCOT and the MQ is based on Figure 3.2. The MQ consumes all outputs from the EBCOT, and does not need any more inputs. The codeblock formation task is given to the Partitioner. The Data Orderer organizes the MQ output, but is affected by the Partitioner's output order. The Data Orderer will compensate by inserting missing items, if the initial codeblock assumption proves wrong. The Packetizer follows the definitions in the JPEG2000 standard, but is not currently implemented.

4.2 Image Parts

Splitting the image into parts reduces the requirements for internal memory when processing. Smaller image pieces demand less internal memory at once. The same actually applies to developers, as the hierarchy structure enables a black-box design thinking. Splitting also allows separately processing some of these pieces, which allows parallel execution. Tile components are treated as individual images, and can be processed separately. The current EBCOT implementation can execute independent of other codeblocks. The same applies for the MQ, but some *if* conditions are not currently used properly.

4.3 EBCOT

The purpose of the EBCOT is to find the order the significant states are updated in. This is the information contained in the context labels and decisions. For instance the significance propagation pass predicts that significance states will update. Both if they do and if they do not it is interesting information, and is useful to encode.

Learning the coding passes in a certain order may make them easier to understand. The magnitude refinement pass is the simpler of the three, while the cleanup pass is more difficult. This is based on the respectively few and many conditions to consider. The magnitude refinement pass also has fewer context labels than the other two.

4.3.1 Coding Pass Roles

The roles of the coding passes are defined by their use of significance states. The magnitude refinement pass use the 1s, while the significance propagation pass and cleanup pass use the 0s. This essentially means that the EBCOT is interested mainly in the 0s, especially when they are about to become 1s. The magnitude refinement pass execution is now separated from the other two. Now the other two coding passes' execution must be separated. The significance propagation pass requires 1s in at least one neighbour. The cleanup pass thus encodes only the coefficient bits that have significance state 0 as well as being surrounded by such neighbours. There exists a coding pass example in the JPEG2000 standard [[1]Table D.6]. It is used as a reference to when the coding passes should execute.

The prediction in the significance propagation pass can compare neighbours to 1 directly, or create and compare the context label against 0. These are equivalent, as seen in Table 3.1. The difference in these approaches are not much. The context label can be reused if this coefficient is to be encoded in this pass, which saves some work. The work is however unnecessary otherwise.

During the EBCOT encoding there is a trend in the roles of the coding passes. The amount of 1s in the significance states increases as the bitplanes are traversed from MSB to LSB. This causes the amount of coefficient bits encoded by the magnitude refinement pass to increase, while those encoded by the cleanup pass decrease. The roles do not at any time go in the opposite direction. The trend does not guarantee the significance propagation pass is affected in one direction only, as the following proves. Consider the case where a rectangle with significance states 1 encircles a 4 times 4 area with 0s. Neighbours of 1s are encoded in the significance propagation pass. If a single significance state 1 is inserted in the center 2 times 2 area, 3 to 5 new bits will be encoded by the significance propagation pass. If the entire area with 0s is now replaced with 1s, there are now fewer bits encoded in the significance propagation pass. The trend requires any non-zero coefficients to exist in the codeblock. If they do not, all the empty bitplanes will be skipped, as already elaborated.

Prior to the initial cleanup pass the neighbour contribution will be 0. The significance propagation pass will therefore not encode any coefficient bits until after the first MSB-plane. The magnitude refinement pass will not encode because none have significance state 1. These are the reasons for skipping the first bitplanes.

4.3.2 Coding Methods

The magnitude refinement coding needs to know if the current coefficient has been encoded in a refinement pass before. If is has, other details are disregarded. If it has not, the magnitude refinement coding regards the neighbours. A context label is created from this knowledge alone. The decision uses the coefficient bit.

The significance propagation coding encodes solely based on the neighbours. Upon finding a coefficient bit with value 1, it also executes sign coding. The sign coding then considers both the neighbours in the significance state array and the sign array. The sign coding does not use the diagonal neighbours of the sliding window.

The normal cleanup coding is identical to the significance propagation coding. This is true because it is already established that the cleanup pass executes this particular coefficient bit.

The run-length cleanup coding looks ahead to determine if the four coefficient bits of one column are encoded in the cleanup pass. If it is, the run-length context label is created. The look ahead should simultaneously detect the first non-zero coefficient bit's position. If there are only 0s, the decision is 0. Otherwise the decision is 1, and the position is used for the next two decisions. These two decisions use the Uniform context label. The position is reduced to a relative value from the top of the column, ranging from 0 to 3. The two bits used to represent the value are sent MSB first as decisions.

The run-length coding has the ability to reduce the amount of context labels created, compared to only normal cleanup coding. This occurs in the circumstance where there are four continuous coefficient bits. In normal coding these produce 4 context labels, plus 4 from sign coding if the bits are 1s. The run-length coding produces 1 context label for the same first situation. The second situation with 1s produces minimum 4 context labels for a bit sequence of `0001` and maximum 10 context labels for a bit sequence of `1111`. All sign codings are kept, but the context labels up to the first non-zero bit are replaced by the 3 context labels for run-length coding. Experimenting with sequences of bits with one value 1 gives a constant amount of 5 context labels for normal coding. Run-length coding produces 4 context labels for the sequence `0001`, increasing by one up to a total 7 context labels for `1000`. This proves that it can actually increase the total results produced in the EBCOT. They are, however, compressed in the MQ, which should still decrease the total image size. The run-length coding is not repeated on a column after finding a bit value of 1.

The run-length coding seem to try to simplify the encoding results, by possibly reducing the amount of context labels. It is most efficient in the empty or almost empty bitplanes, but the first empty bitplanes are removed before the first cleanup pass. The run-length also gives a very precise description to the reconstructing decoder. Thus the initial conditions of the codeblock is easily determined.

For the coding passes, only the sum of the three directional categories are important, namely H, V and D. For the sign bit encoding, however, the individual horizontal and vertical neighbours are of importance.

4.3.3 Trivial Parts of Implementation

The scan pattern implementation is trivial. It is easily verified by filling an array with numbers, similar to Figure 3.6. Producing the sign bits and magnitude bits are also trivial by using built-in functions for sign and absolute values.

Significance states can be considered as another, but temporary, bitplane. The states are represented by a single bit each. As the coding passes conclude, the significance states have fulfilled their use. They can now either be reinitialized to 0 and reused for new codeblocks, or thrown away. This depends on whether they are implemented as something that stays in the EBCOT or something that follows the codeblocks. If reused, they must tolerate different sizes of codeblocks.

Similar to the image pieces, the use of the sliding window is only a means of acquiring positions. The sliding window use the same index as the current values $\pm$ 1 to get all eight directional positions. At the edges of the codeblock the sliding window will attempt to access neighbours that are outside the codeblock. This access would require a memory fetch from another codeblock. Instead of allowing this access, they are treated as 0. This can be achieved by simply padding the significance state array with 0s around the edges. The padding negates the need for border conditions, which makes the scan pattern easier to implement.

4.3.4 Read and Write Observations

Presented here are the coefficient bits, significance states and sign bits, and their type of usage. Their sliding window position is included. The only coefficient bit ever used is at the center of the sliding window. The significance states use all positions. The sign bits use all but the diagonal positions. The coefficient bit is never written to; only read. The central significance state is written to if the coefficient bit is 1, but never read. The neighbouring significance states are only ever read. All sign bits are only ever read.

4.4 MQ

The MQ is described in high detail in the JPEG2000 standard. It shows detailed flowcharts for each of the functions implemented in this thesis. Thus there is not much to analyze and discuss here.

The MQ is initialized to certain state values, for all of the 19 context states. For the most part the states increase if the decision equals the MPS, and decreases if it equals the LPS. The context labels and decisions are read one pair at a time. The states are updated after each. The states provide a *Qe* which is added or subtracted to/from *A*-reg or *C*-reg. *C*-reg periodically returns an output byte, which is here collected in a list. At the end of the MQ encoding, the remaining contents of *C*-reg is collected in the list in a special *flush* procedure. The complete bitstream list is sent as an output from the MQ, to be sorted in the Data Orderer.

The three procedures/functions *Encode*, *Code1* and *Code0* from the flowcharts are merged into one in the implementation. They determine whether the current decision symbol is the MPS or

the LPS, which triggers execution of the appropriate coding method. This is a minor modification.

The bitstream produced in the current implementation does not yet contain packet headers and in-stream markers. The lack is a result of not yet having investigated how to treat the MQ's output. The packet headers are intended to be inserted by the Data Orderer. The MQ output bytes require a bit-stuffing procedure in relation to the byte at the end of packet header. The Data Orderer must therefore perform this bit-stuffing upon inserting the packet headers. Inserting a 0 after each sequence of `0xFF` should hopefully be trivial to implement.

4.5 Investigating the Partitioner and Data Orderer

The work of this thesis has so far been concentrated on the EBCOT and the MQ. Continuing to the Partitioner and Data Orderer, the Encoder module as a whole must be considered. The considerations regard the memory management and the sorting in the Data Orderer.

4.5.1 Data Orderer Considerations

The sorting in the Data Orderer will be affected by the output order from the Partitioner. The sorting is based on the packet formation. Therefore, the packets must be considered. The packets consists of certain bitplanes of several codeblocks. This is in contrast to encoding whole codeblocks, as initially assumed. The system must be adapted to the now altered conditions.

Initially it appears that the EBCOT has to be modified to pause after certain bitplanes. It will continue after encoding the other codeblock parts of the packet. This method produces one finished packet before the next. However, a more thorough examination points to the only issue being the MQ output. To adjust the current implementation to the updated setting, the MQ's bitstream can be separated into individual streams for each bitplane. The packets are thus produced codeblock by codeblock. Each packet, in one precinct location, is finished simultaneously. The precinct location is used in all sub-bands of the resolution level.

It is difficult to split the finished bitstream of a codeblock without providing lots of dictations to the MQ during the coding passes. If also including the creation of the packet headers, the Partitioner must transfer the information about the image sizes as well. This will essentially merge the MQ with the Data Orderer. Merging the sub-modules will create a larger system block, which is less manageable. When the data is appended to a packet, the bit-stuffing procedure must be repeated.

Instead of this complicated modification, the bitstreams can be split during their generation. This corresponds more with the current MQ implementation, because only the output address must be sent as additional input to the MQ. The address takes the place of the byte stream start pointer *BPST*. The input address will change throughout the codeblock execution in the EBCOT, when the encoded content belongs in another packet. The MQ must finish and reset before the change in *BPST*, as the *BPST* is set only in the initialization of the MQ. The output bytes are appended to the packet headers.

The bitstreams should be processed by the Data Orderer. This lets it sort them into packets. The sorting is performed by providing the address *BPST* to the MQ sub-module. To know the

appropriate addresses, the Data Orderer should create the packet headers. The actual sorting is determined by the progression order and compression rate. These must be investigated in a future work, along with the content of the packet headers.

4.5.2 Memory Considerations

The EBCOT and MQ implementations currently produce results in new memory locations; in temporary lists. The incoming quantized image of sub-bands is already stored in a memory location. The same location should eventually be reused, when the data there is no longer needed. A temporary location requires doubly the size the current image, minus the reduction to the size, which should be avoided.

The output of the Partitioner does not need storage, because the incoming data is not modified; it is merely split into codeblocks. The output of the EBCOT does not need storage, because the MQ consumes it immediately. The bitstream output must be stored in different packets, by either the MQ or the Data Orderer. Providing the address input to the MQ from the Data Orderer serves to keep the sub-modules' tasks in smaller and more manageable blocks. The Data Orderer then also naturally takes the task of storing the bitstreams, in the packets. The bitstream storage requires room for all the encoded codeblocks of one precinct position.

With codeblocks encoded one full codeblock at a time, the codeblock's memory location can be overwritten. The purpose of the EBCOT and MQ is to reduce the size of the data, meaning the encoded data will not exceed the codeblock data. The encoded data overwriting will therefore not overflow into the other, un-encoded codeblocks. The resulting packet datastreams may however get a complex address look-up. Their coordinates will follow raster order within a codeblock. This is simple enough for finished streams, but these streams will be appended to and grow over time. It is therefore preferred to store packets temporarily in a new memory location, until the packet is complete or the outdated data is of sufficient size for the packet.

The Data Orderer can sort the packet datastreams into layers after the whole image is encoded, or during the progress. Regardless of this, the result must be an encoded datastream in sorted order. As they are sorted the overwriting operation ignores the previous boundaries and data content. The completed Data Orderer operation has thus produced a single stream of packets or layers. The Packetizer should not need to read the data from arbitrary codeblock locations. The datastream must still be sectioned so that the Packetizer can find the starts and ends of where it should place its markers and headers, but this is what the packet headers are for.

4.6 Partitioner

The task of dividing the sub-bands into precincts and codeblocks is performed in the Partitioner. It creates a stream of codeblocks, from the incoming quantized sub-band image. These are then sent for encoding in the EBCOT. The metadata information must be sent to the Data Orderer when the bitstream arrive from the MQ.

The VHDL implementation [2] provides a quantized image of sub-bands with tile size 128×128 and 5 decomposition levels. Descriptions from sections 3.2.2, 3.2.3 and 3.2.4 lead to the following computations and results: From 5 decomposition levels the $N_L LL$ sub-band has a

size of 4×4. As precincts are parts of a sub-band, the maximum precinct size of the N_LLL band is also 4×4. The N_LLL band is at resolution level 0, which gives codeblocks of maximum size 4×4. This codeblock value is also the minimum allowed, from Equation 3.3. The CB' value may, however, be higher for other resolution levels. The values for maximum sizes of the other resolution levels are computed in the same fashion. The values used remain to be selected, but these maximum values are used to implement a test Partitioner, provided in Appendix A.

The order of sending codeblocks to the EBCOT must be determined. The simplest solution for the Partitioner is to send in raster order. Both raster order and other orders stay true to the initial intentions for the Partitioner and EBCOT relation, by using entire codeblocks. The partially fulfilled Partitioner implementation currently finds codeblocks by precinct positions in decomposition levels, as in the packet formation. It attempts to produce the codeblocks in packet order for the Data Ordererto handle them easier. The Partitioner does not currently send the found codeblocks to the EBCOT.

4.7 Data Orderer

The Data Orderer sorts the bitstreams from the MQ and metadata from the Partitioner. The resulting datastream is sent to the Packetizer. The Data Orderer must organize bitstreams into packets and layers. The bitstreams do not overwrite any data in the current implementation, so the Data Orderer could actually acquire the metadata by itself.

The packet headers contain information about which codeblocks and coding passes are included. This information must be used to place the MQ bytes at addresses of different packets. Packet headers should exist upon initiating the MQ encoding to achieve proper results. The Data Orderer will control the byte's output location and sort as items are encoded.

4.8 File Packetizer

The Packetizer finalizes the output of the JPEG2000 compression system. The resulting jp2 file is readable by any decoder. The file has some required and some optional file markers and headers. The main header identifies the file syntax. A tile header precedes each tile component. At the end of the file the *EOC* marker is placed, indicating End of Codestream.

As a specialized encoder it can use hardcoded data for constants. This simplifies some operations by reducing the amount of conditions. If for example the camera sensor does not have adjustable image resolution and the same tile size is used, a pregenerated *SIZ* marker can be used.

The main remaining decision is the behavior of the output. This involves where and how it will be placed. Currently affected sub-modules of the Encoder are the MQ, Partitioner and Data Orderer. Due to not fully performing these tasks, the Packetizer implementation is not yet begun. This is also the reason why the theory and description for the Packetizer lack vital information. Consult the documentation for details [1] [2].

Chapter 5

MATLAB Implementation

This chapter highlights parts from the MATLAB implementation. Extracts from the code is presented, accompanied by explanations for the choices. For the full code, see Appendix A. The implementation was made with MATLAB version R2017a [3].

The implementation is made with emphasis on being understandable. Many variable names are long, as they have deliberately not been abbreviated. This hopefully helps the reader. Extra steps have been made for the same reason, such as using the variable *coefficientBit* instead of directly accessing *bitplane(row,col)*.

Please note that MATLAB automatically converts numbers to *double* format inside every function call. An effect of this was an unintended higher-than-byte value in the MQ's bitstream output. This particular bug was fixed, but similar bugs could still exist.

This work aims for a hardware implementation of the JPEG2000 compression algorithm. The MATLAB implementation is the first step towards that goal.

5.1 EBCOT

The EBCOT encodes codeblocks. This produces context labels and decisions at various intervals. The output is collected in a list *EBCOToutput*. Results are appended to the end of the list as they are created, by a function call to *appendData(..)*.

The function declarations are listed in Listing 5.1. These are intended for reference throughout this chapter. The functions are actually placed in separate files in the implementation, as is the tradition in MATLAB. Throughout the listings in this section an extract of their code will be presented.

Listing 5.1: The declarations for functions in T1 EBCOT.

```
1 % Executes the EBCOT coder on one codeblock.
2 function [EBCOToutput, emptyBitplanes] = T1_EBCOT(codeblock,
     subbandType, bitsPerPixel)
3
4 % Appends second data input to the first list input.
5 function EBCOToutput = appendData(EBCOToutput, newData)
```

```
6
7  % Performs normal cleanup coding on one bit.
8  function [EBCOToutput, significantStatesPadded] = cleanupCodingNormal
     (EBCOToutput, bitplane, significantStatesPadded, subband,
     codeblockSignsPadded, row, col)
9
10 % Performs run-length coding on four bits.
11 function [EBCOToutput, significantStatesPadded] =
     cleanupCodingRunLength(EBCOToutput, bitplane,
     significantStatesPadded, subband, codeblockSignsPadded, baserow,
     col)
12
13 % Executes cleanup pass on one bitplane.
14 function [EBCOToutput, significantStatesPadded] = cleanupPass(
     EBCOToutput, bitplane, significantStatesPadded, subband,
     codeblockSignsPadded, needCleanup)
15
16 % Performs magnitude refinement coding on one bit.
17 function [EBCOToutput, needFirstMR] = magnitudeCoding(EBCOToutput,
     bitplane, significantStatesPadded, needFirstMR, row, col)
18
19 % Provides lookup value for context label.
20 function contextLabel = magnitudeContext(firstMR, slidingWindow)
21
22 % Executes magnitude refinement pass on one bitplane.
23 function [EBCOToutput, needFirstMR] = magnitudePass(EBCOToutput,
     bitplane, significantStatesPadded, needMR, needFirstMR)
24
25 % Provides directional sums from sliding window.
26 function [sumH, sumV, sumD] = neighbourContribution(slidingWindow)
27
28 % Performs sign coding on one bit.
29 function EBCOToutput = signCoding(EBCOToutput, slidingWindow,
     codeblockSignsPadded, row, col)
30
31 % Performs significance propagation coding on one bit.
32 function [EBCOToutput, significantStatesPadded, remainingCoefficients
     ] = significanceCoding(EBCOToutput, bitplane,
     significantStatesPadded, subband, codeblockSignsPadded,
     remainingCoefficients, row, col)
33
34 % Provides lookup value for context label.
35 function contextLabel = significanceContext(subband, slidingWindow)
36
37 % Executes significance propagation pass on one bitplane.
38 function [EBCOToutput, significantStatesPadded, remainingCoefficients
     ] = significancePass(EBCOToutput, bitplane,
     significantStatesPadded, subband, codeblockSignsPadded)
```

The input in the FPGA will be of the signed magnitude representation. This means the first

bit is a sign bit, with 0 for positive and 1 for negative. The remaining bits hold the absolute value, i.e. greater than or equal to 0. To enforce conforming with the representation, the built in functions for sign value and absolute value is executed. The sign function however returns +1 or -1, instead of 0 and 1. It is currently interpreted correctly within the sign coding function, but should also be fixed in the *codeblockSigns* variable. *codeblockAbs* is used to fill bits of the bitplanes.

The sliding window should read 0 when attempting to access significance states outside the codeblock. This is achieved by adding a border with 0s around the edges of the array. The resulting *significantStatesPadded* is higher and wider by two coefficients. To reach the current scan coordinate at (*row*, *col*), the *significanceStatePadded*(*row*+1, *col*+1) must be accessed. The padded array is simpler than using conditional checks at the edges. It does come at the cost of a larger array in memory. The trade-off should be evaluated in the VHDL code.

Deciding which coding pass should encode a particular coefficient bit is done by looking at the significance states. If the current coefficient's significance stateis 1, it is encoded in the magnitude refinement pass. If it is 0, the significance propagation pass or the cleanup pass encodes it. The significance states used to determine this are the significance states from before the current bitplane begins its encoding. For instance, if the significance states are updated in the significance propagation pass, the roles of the passes should not be affected. An updated value would make the magnitude refinement pass try to encode the coefficient bit. The new-found significance states must be excluded, as the coefficient bit must be encoded just once in a bitplane. The solution is to store the significance states array prior to the first coding pass. This guarantees that changes does not affect the magnitude refinement pass. Additionally, the first time a coefficient is encoded by the magnitude refinement pass must be detected. Another array, *needFirstMR*, is used for this.

The significance states are initialized to 0. An array *needCleanup* determines when the cleanup pass encodes a particular coefficient bit. It is initialized to 1, because the cleanup pass encodes the first bitplane. An array *needMR* determines when the magnitude refinement pass encodes a particular coefficient bit. It is a copy of the significance states array. It does not use the same array, because that array is modified thoughout the bitplane coding. An array *needFirstMR* tracks the first time the magnitude refinement pass encodes a particular coefficient (not the bit). It is initialized to 1, and updates to 0 when encoded.

The bitplanes are collected in a variable *codeblockBitplanes* through the *bitget(..)* function. The implementation simply uses two for-loops to iterate over each coefficient position, and one for-loop to repeat this over each bit. *bitget(..)* returns bit at the input *n*-th position of the input number.

5.1.1 Coding Passes

Listing 5.2 shows how the empty bitplanes are skipped. The four *any* functions guarantee the result is one boolean value, rather than a vector or array. The current and subsequent bitplanes are not skipped. The amount of bits here begins with *bitsPerPixel*, which is the number of bits used to store the values. This may have to be reduced by one to be correct, based on signed magnitude representation. The bitplanes should not include the sign plane.

Listing 5.2: The empty bitplanes are skipped.
T1_EBCOT.m:

```
1 for bitNumber = bitsPerPixel:-1:1         % MSB to LSB.
2     bitplane = codeblockBitplanes(:,:,bitNumber);
3     if any(any(any(any(bitplane))))
4         break;
5     end
6 end
7 emptyBitplanes = bitsPerPixel - bitNumber;
```

An initial cleanup pass is executed before entering the main loop. This is a function call, identical to the cleanup pass in the main loop. Hence it will not be listed explicitly here.

The main loop is seen in Listing 5.3. It begins with fetching the correct bitplane. *codeblockBitplanes* is a collection of bitplanes that is created by using MATLAB's built-in *bitget* function. The *needMR* array is updated before the significance states may be modified. The coding passes then execute in the normal order, as presented earlier.

Listing 5.3: The main loop for the coding passes.
T1_EBCOT.m:

```
1  for i = (bitNumber-1):-1:1       % MSB to LSB.
2      bitplane = codeblockBitplanes(:,:,i);
3
4      needRefinement(1:codeblockHeight,1:codeblockWidth) ...
5          = significantStatesPadded(2:(codeblockHeight+1), 2:(
              codeblockWidth+1));
6
7      [EBCOToutput, significantStatesPadded, needCleanup] ...
8          = significancePass(EBCOToutput, bitplane,
              significantStatesPadded, subbandType, codeblockSignsPadded
              );
9      [EBCOToutput, needFirstRefinement] ...
10         = magnitudePass(EBCOToutput, bitplane,
              significantStatesPadded, needRefinement,
              needFirstRefinement);
11     [EBCOToutput, significantStatesPadded] ...
12         = cleanupPass(EBCOToutput, bitplane, significantStatesPadded,
               subbandType, codeblockSignsPadded, needCleanup);
13 end
```

5.1.2 Scan Pattern

All three coding passes use the scan pattern shown in Listing 5.4. The specific code for each coding pass replaces "<Insert code>". The implementation has two sections. The first is for regular four-column-coefficients scan. The second is for the last rows containing less than four coefficients. This method is used instead of an if-condition on every scan coefficient, checking if the height is less than an integer multiple of four.

Listing 5.4: The scan pattern traversal. ”<Insert code>” is replaced by pass specific code.
significancePass.m,
magnitudePass.m and
cleanupPass.m:

```
1  for baserow = 1:4:(4*floor(bitplaneHeight/4))
2      for col = 1:bitplaneWidth
3          for row = (baserow+0):(baserow+3)
4              % <Insert code>.
5          end
6      end
7  end
8  if (row < bitplaneHeight)
9      finalRow = row + 1;
10     for col = 1:bitplaneWidth
11         for row = finalRow:bitplaneHeight
12             % <Insert code>.
13         end
14     end
15 end
```

5.1.3 Significance Propagation Pass

The significance propagation pass executes the code in Listing 5.5. This is the code that replaces ”<Insert code>” within the scan pattern. If the current coefficient’s significance state is 0, the significance propagation pass or the cleanup pass encodes it. The array *remainingCoefficients* collects the coefficient bits not encoded in the significance propagation pass, so the cleanup pass will encode them.

Listing 5.5: The <insert code> content of the significance propagation pass.
significancePass.m:

```
1  if significantStatesPadded(row+1, col+1) == 0
2      [EBCOToutput, significantStatesPadded, remainingCoefficients] =
          significanceCoding(EBCOToutput, bitplane,
          significantStatesPadded, subband, codeblockSignsPadded,
          remainingCoefficients, row, col);
3  end
```

5.1.4 Significance Coding

The significance coding implementation is shown in Listing 5.6. The prediction is considered before the significance coding is executed. If the context label is 0, no neighbours have significance state 1. *contextLabelForSignificance* is used to find the context label, equivalenting to Table 3.1. If the context label is 0 the coefficient bit is not encoded in this pass, but is sent to the cleanup pass instead. Otherwise the context label and decision is added to the end of the *EBCOToutput* list. The significance states are updated if necessary, which triggers the sign coding as the next step.

Listing 5.6: The significance coding function.
significanceCoding.m:

```
slidingWindow = significantStatesPadded((row):(row+2),(col):(col+2));
contextLabel = contextLabelForSignificance(subband, slidingWindow);

if contextLabel ~= 0
    coefficient = bitplane(row,col);
    decision = coefficient;
    EBCOToutput = appendData(EBCOToutput, [contextLabel, decision]);

    if coefficient == 1
        significantStatesPadded(row+1,col+1) = 1;

        EBCOToutput = signCoding(EBCOToutput, slidingWindow,
           codeblockSignsPadded, row, col);
    end
else
    remainingCoefficients(row,col) = 1;
end
```

5.1.5 Sign Coding

Listing 5.7 shows the sign coding function. The sliding window is used to determine the contribution of H and V, as per Table 3.4. The contributions are then used in the implementation of Table 3.5 to provide a context label and an *xorBit*. The decision is then determined from the *xorBit* and the sign bit. The context label and decision is appended to *EBCOToutput*.

Sign coding encodes one coefficient bit before returning to the coding pass that triggered it. It is triggered by either the significance propagation pass or the cleanup pass.

Listing 5.7: The sign coding is called in the significance propagation pass and the cleanup pass.
signCoding.m:

```
contributionH = signContribution('H', slidingWindow,
   codeblockSignsPadded);
contributionV = signContribution('V', slidingWindow,
   codeblockSignsPadded);
[contextLabel, xorBit] = signContext(contributionH, contributionV);

coefficientSign = codeblockSignsPadded(row+1, col+1); % +1 or -1.
signbit = (coefficientSign < 0) + 0;
decision = bitxor(signbit, xorBit);

EBCOToutput = appendContextData(EBCOToutput, [contextLabel, decision
   ]);
```

5.1.6 Magnitude Refinement Pass

The magnitude refinement pass executes the code in Listing 5.8. This replaces ”<Insert code>” within the scan pattern. If the current coefficient’s significance state is 1, it is encoded in the magnitude refinement pass.

Listing 5.8: The <insert code> content of the magnitude refinement pass.
magnitudePass.m:

```
1 if needMR(row,col) == 1
2     [EBCOToutput, needFirstMR] = magnitudeCoding(EBCOToutput,
        bitplane, significantStatesPadded, needFirstMR, row, col);
3 end
```

5.1.7 Magnitude Refinement Coding

Listing 5.9 shows the coding for magnitude refinement. The implementation is straightforward. The context label is generated from an implementation of Table 3.2, and the decision is the coefficient bit. The logical value indicating if this is the first time this coefficient bit is magnitude refinement coded, *needFirstMR*, is updated to 0.

Listing 5.9: The magnitude refinement coding function.
magnitudeCoding.m:

```
1 coefficient = bitplane(row,col);
2 slidingWindow = significantStatesPadded((row):(row+2),(col):(col+2));
3 firstMR = needFirstMR(row,col);
4
5 contextLabel = contextLabelForMagnitude(firstMR, slidingWindow);
6 decision = coefficient;
7 EBCOToutput = appendData(EBCOToutput, [contextLabel, decision]);
8
9 needFirstMR(row,col) = 0;
```

5.1.8 Cleanup Pass

The cleanup pass executes the code in Listing 5.10 within the <insert code> in the scan pattern. Note that the innermost for-loop in Listing 5.4) is replaced by the for-loop in Listing 5.10, so as not to repeat the same.

The cleanup pass encodes remaining coefficient bits. This is achieved by marking coefficient bits that have been encoded. A shortcut past checking remaining after the magnitude refinement pass is done by checking the significance states. Only significance states that are 0 are encoded by the cleanup pass. This reduces the task to finding which coefficient bits were encoded in the significance propagation pass. An array for marking the coefficient bits is initialized to 0. A position is updated to 1 if the significance propagation pass does not encode it.

The cleanup pass first determines whether to use *cleanupCodingNormal* or *cleanupCodingRun-Length*. This requires verifying all four coefficient bits in a column, which means the innermost

for-loop must complete its iteration. All four coefficient bits must remain to be encoded by the cleanup pass, i.e. encoded in neither the significance propagation pass nor the magnitude refinement pass. All four coefficient bits must currently have a context label that is 0, i.e. all neighbours' significance states are 0.

If the conditions are met, the run-length cleanup coding is executed. Otherwise, the normal cleanup coding is executed, where the four coefficient bits are again iterated through the four coefficient bits.

Listing 5.10: The first <insert code> content of the cleanup pass. The inner for-loop of Listing 5.4 is not used.
cleanupPass.m:

```
1  normalMode = false;
2
3  %%% Determine run-length coding or normal coding for this column.
4  for row = baserow:(baserow+3)
5      if (needCleanup(row,col) ~= 1)
6          normalMode = true;
7          break;
8      end
9
10     slidingWindow = significantStatesPadded((row):(row+2), (col):(col
          +2));
11     contextLabel = contextLabelForSignificance(subband, slidingWindow
          );
12
13     if (contextLabel ~= 0)
14         normalMode = true;
15         break
16     end
17 end
18
19 if (normalMode == false)    %%% Use Run-length Coding.
20     [EBCOToutput, significantStatesPadded] = cleanupCodingRunLength(
          EBCOToutput, bitplane, significantStatesPadded, subband,
          codeblockSignsPadded, baserow, col);
21 else       %%% Use Normal Cleanup Coding.
22     for row = baserow:(baserow+3)
23         if (needCleanup(row,col) == 1)
24             [EBCOToutput, significantStatesPadded] =
                  cleanupCodingNormal(EBCOToutput, bitplane,
                  significantStatesPadded, subband, codeblockSignsPadded
                  , row, col);
25         end
26     end
27 end
```

The second part of the scan pattern only executes normal cleanup coding. This is the part from line 8 to 15 in Listing 5.4. The reason is because there are less than four rows remaining, so

there is no run-length. The cleanup pass is therefore different in the two <insert code> parts. The second part is seen in Listing 5.11.

Listing 5.11: The second <insert code> content of the cleanup pass. (The for-loop is the same as Listing 5.4.)
cleanupPass.m:

```
1  %%% Normal Cleanup Coding.
2  [EBCOToutput, significantStatesPadded] = cleanupCodingNormal(
      EBCOToutput, bitplane, significantStatesPadded, subband,
      codeblockSignsPadded, row, col);
3  end
```

5.1.9 Normal Cleanup Coding

Listing 5.12 shows the normal cleanup coding. This is rather straightforward. The context label is generated as per Table 3.1 and the decision is gathered from the current coefficient bit. This is appended to the *EBCOToutput*. The significance state is updated if the coefficient bit is 1, and sign coding is triggered.

Listing 5.12: The normal cleanup coding function.
cleanupCodingNormal.m:

```
1  coefficient = bitplane(row,col);
2  slidingWindow = significantStatesPadded((row):(row+2),(col):(col+2));
3
4  contextLabel = contextLabelForSignificance(subband, slidingWindow);
5  decision = coefficient;
6  EBCOToutput = appendData(EBCOToutput, [contextLabel, decision]);
7
8  if coefficient == 1
9      significantStatesPadded(row+1,col+1) = 1;
10
11     % Immediate next encoding is sign bit.
12     EBCOToutput = signCoding(EBCOToutput, slidingWindow,
          codeblockSignsPadded, row, col);
13 end
```

5.1.10 Run-length Cleanup Coding

The run-length coding is shown in Listing 5.13. At first the for-loop checks if the four continuous coefficient bits are all 0, or if there is a 1 present.

With all-zeroes, the short run-length solution is produced. The short solution is where the context label holds the run-length value, here chosen to 17, and the decision is 0. This is appended to *EBCOToutput*.

With any coefficient bit 1, the long run-length solution is produced. The context label is 17 and the decision is 1. Th uniform context label then follows for two entries. It is here chosen to be

18. The two entries have context label 18, while the decision is two bits representing the row where the coefficient bit equal 1 was found. These are gathered from the variable *decisionBase*.

Still considering the coefficient bit 1 case, the sign coding follows, as usual for finding a coefficient bit 1. Finally, the normal cleanup coding encodes any coefficient bits remaining, of the four continuous in a column.

Listing 5.13: The run-length coding function.
cleanupCodingRunLength.m:

```
for columnIndex = 0:3
    row = baseRow + columnIndex;
    coefficientBit = bitplane(row, col);
    if (coefficientBit == 1)
        break;
    end
end

if (coefficientBit == 0)
    contextLabel = 17;
    decision = 0;
    EBCOToutput = appendData(EBCOToutput, [contextLabel, decision]);
else
    contextLabel = 17;
    decision = 1;
    EBCOToutput = appendData(EBCOToutput, [contextLabel, decision]);

    contextLabel = 18;
    decisionBase = columnIndex - 1; % Result is one of {0, 1, 2, 3}.

    decision = bitget(decisionBase, 2);
    EBCOToutput = appendData(EBCOToutput, [contextLabel, decision]);

    decision = bitget(decisionBase, 1);
    EBCOToutput = appendData(EBCOToutput, [contextLabel, decision]);


    slidingWindow = significantStatesPadded((row):(row+2), (col):(col
        +2));
    EBCOToutput = signCoding(EBCOToutput, slidingWindow,
        codeblockSignsPadded, row, col);


    significantStatesPadded(row+1, col+1) = 1;

    for i = (row+1):(baserow+3)  % Normal Mode if any remain.
        [EBCOToutput, significantStatesPadded] = cleanupCodingNormal(
            EBCOToutput, bitplane, significantStatesPadded, subband,
            codeblockSignsPadded, i, col);
    end
end
```

5.2 MQ Coder

The implementation follows the detailed flowcharts in the JPEG2000 standard [1], and is therefore not interesting to comment in details here. Only a brief selection will be displayed. See Appendix A for the full code. The function declarations are listed in Listing 5.14. In the implementation they actually have their own files.

Listing 5.14: The declarations for functions in T1 MQ.

```
1  % Executes the MQ coder on one list of EBCOT output.
2  function MQoutput = T1_MQ(EBCOToutput)
3
4  % Appends byte to MQ output.
5  function [MQoutput, BP, C, CT] = byteOut(MQoutput, BP, C)
6
7  % Executes LPS branch.
8  function [MQoutput, indexCX, A, BP, C, CT, mpsCX] = codeLPS(MQoutput,
        indexCX, A, BP, C, CT, mpsCX)
9
10 % Executes MPS branch.
11 function [MQoutput, indexCX, A, BP, C, CT] = codeMPS(MQoutput,
       indexCX, A, BP, C, CT)
12
13 % Executes MPS or LPS branch based on decision.
14 function [MQoutput, indexCX, mpsCX, A, BP, C, CT] = decideEncode(
       MQoutput, indexCX, mpsCX, descision, A, BP, C, CT)
15
16 % Appends final remaining bytes to MQ output.
17 function [MQoutput, BP, C, CT] = flush(MQoutput, A, BP, C, CT)
18
19 % Provides lookup table for Qe and next-state values.
20 function [Qe, NMPS, NLPS, switchValue] = probabilityEstimation(
       indexCX )
21
22 % Shifts the A and C registers after each decision branch.
23 function [MQoutput, A, BP, C, CT] = renorme(MQoutput, A, BP, C, CT)
24
25 % Sets 'x' bits to 1 before flushing.
26 function C = setBits(A, C)
```

Similar to the EBCOT implementation, the MQ stores its output in a *MQoutput* variable. As was mentioned earlier, this is most likely not ideal for the MQ. The result should rather be stored in a common memory location.

The main loop of the MQ executes as shown in Listing 5.15. Two lists, *indexLookup* and *mpsLookup*, represent the context states for index and symbol. The current context label from the EBCOT decides which state is being modified. The modification involves the next state, *NMPS* or *NLPS*, and the encoding variable *Qe*. Due to MATLAB's 1-indexed lists, an offset of 1 is added to each context label before being used to access the lists. The *decideEncode(..)*

function is the merged *Encode*, *Code0* and *Code1* functions defined in the JPEG2000 standard. It is shown in Listing 5.16.

Listing 5.15: The main loop of the MQ function.
T1_MQ.m:

```
1  for i = 1:size(EBCOToutput,1)
2      contextLabel = EBCOToutput(i,1);
3      decision = EBCOToutput(i,2);
4
5      indexCX = indexLookup(contextLabel + offset);
6      mpsCX = mpsLookup(contextLabel + offset);
7
8      [MQoutput, indexCX, mpsCX, A, BP, C, CT] = decideEncode(MQoutput,
           indexCX, mpsCX, decision, A, BP, C, CT);
9
10     indexLookup(contextLabel + offset) = indexCX;
11     mpsLookup(contextLabel + offset) = mpsCX;
12 end
```

Listing 5.16: The three merged functions for reading decision.
decideEncode.m:

```
1  if descision == mpsCX
2      [MQoutput, indexCX, A, BP, C, CT] = codeMPS(MQoutput, indexCX, A,
           BP, C, CT);
3  else
4      [MQoutput, indexCX, A, BP, C, CT, mpsCX] = codeLPS(MQoutput,
         indexCX, A, BP, C, CT, mpsCX);
5  end
```

The function *byteOut* for acquiring the 'b' bits of *C*-reg has a conditional increment. This occurs when the carry bit 'c' must be added to the previous byte. MATLAB treats the value in an undesired way for this implementation. Instead of ignoring the higher-than-byte bits, any such causes the value to be rounded up to max, i.e. `0xFF`. This was fixed by first removing the higher bits, then copying the 'b' value. It is, however, worth noting for future work.

5.3 Verification

Test files exists for the EBCOT, MQ and Partitioner implementations, included in Appendix A. They basically execute the functions with appropriate input parameters. The Partitioner finds the values for sizes for image, tile, sub-bands, precincts and codeblocks. The codeblocks are accessed in the proper order for packet formation. The EBCOT provides the inputs for sub-band type, codeblock and bits per pixel. These should be provided by the Partitioner at a later stage. The MQ test file only provides the EBCOT output as information. The two latter test files were mostly used to build the parts of the sub-modules. The functions were extracted into separate files throughout implementation.

Chapter 6

Discussion

6.1 Verification

The work presented in this thesis mainly consists of translating and explaining parts from the JPEG2000 standard. The goal is to achieve progress and to guide future works. The intention is to implement the Encoder in hardware, but the progress has not yet surpassed a reference software model implementation. The model must still be verified along every step of the way.

Throughout the implementation of the JPEG2000 standard's Encoder module the design has been verified. The test files are included in Appendix A. In every stage of the implementation, all parts are of course cross-checked with the analysis of which they are based upon. They are also compared to what the other parts demand from them. The concept could however have been misinterpreted, which then affects everything presented in this thesis.

The intended verification method was to compare with the existing openJPEG implementation [4]. It proved non-straightforward to install and understand, thus it was not prioritized. The main obstacle was the code readability. Their implementation uses structs within structs for convenience, and is built for speed. Many abbreviations exist without clear instructions for what they abbreviate. Thus, understanding it was too time consuming. The MATLAB implementation has also not reached very far, as the focus has been on research. However, it should definitely be consulted for verification both in this and future work. A step by step debug tool is very valuable. Once the MATLAB models for all parts are created, the complete compression chain can be tested. Tests include using any jp2 image viewing applications and converting with openJPEG. Regardless, the current tests are used without openJPEG and cover several topics.

The current implementation has a shorter progress than desired. The goal of this thesis was to produce at least one finished VHDL sub-module. This would have reduced the complexity for the future works. As it is not finished, the future works still has to delve into all the sub-modules herein presented. However, this thesis should be a helpful map for the continued effort. The content should aid both in the continued MATLAB implementation and the entire VHDL implementation of the Encoder module. The presented material would have been of good value at the beginning of this thesis. As part of a standard, the information is less likely to become outdated.

The Encoder module needs more thorough validation of correct behaviour, especially for corner cases. As this is an early development stage, the focus has been more on demonstrating the concept.

6.1.1 Partitioner

The Partitioner has been tested by reversing the read operation. Writing 1s and 0s to the coordinates proves they are accessed in proper orders for the packet formation. There is not much more to do for this currently small sub-module.

6.1.2 EBCOT

The EBCOT is verified by pen and paper walkthrough of every stage. It is then compared with a step by step traversal in debugging mode. The most difficult to understand, and thus verify, was the run-length cleanup coding. The section is explained with long sentences with possible misunderstandings.

The coding passes' order of appearance, or role, is compared to results from an example codeblock in the JPEG2000 standard. It demonstrates when and which of the different coding passes execute on specific bits of the codeblock. It does not show the correct context label sequence. The example has been elaborated by manually generating the output sequence, based on the appropriate context label tables. Because the pass roles are demonstrated and the tables well defined, the output should be fairly accurate. Human error may be a factor, and not all cases are proven in the short example codeblock. Segments of other codeblocks have also been cross-checked manually.

6.1.3 MQ

The MQ has been executed in debug mode and step by step traversed to see that it executes properly. It appears to do so and produces some results, but is by no means verified. The procedures/functions of the MQ are however well described. The merging of three functions into one *decideEncode* is verified by comparing results before and after.

6.2 Unfinished Items

Parts of the Encoder module remain to be implemented as MATLAB reference model. The entire VHDL implementation of the Encoder module also remain. This thesis should serve as guidance in both implementations, and thus save time for future works. The Packetizer is however not thoroughly explained here.

The desire of Orbit to build from scratch is part of the reason the openJPEG [4] implementation is not used instead. The previous work also use a MATLAB implementation, which becomes a full model when finished.

6.2.1 Unimplemented Items

At the current implementation stage, certain items are not yet included. Layers are not investigated. The progression order is yet to be investigated and determined. The Data Orderer is not implemented, but contain useful explanations and analysis. The Packetizer and its headers are not implemented nor investigated.

Sizes for various parts are undecided, e.g. *CB* and *PP*. These should be considered alongside camera sensor, desired image parameters and other items discovered at a future stage.

6.2.2 Partially Implemented Items

Some items are partially included at the current implementation stage. The order to encode codeblocks is built in decreasing resolution order, while using raster order within packets. The codeblocks are not sent to the EBCOT, because the output location was not yet determined. They can be easily sent by adding a function call at the indicated lines in the test file.

The current implementation does not use the proper output location in the MQ. The majority of the MQ is however implemented, although not verified due to the output. A solution is however proposed for it, thus the approach is clear. It should be prioritized, as it finishes one more sub-module. The output first demands the creation of packet headers. The MQ must be adapted minorly for the *BPST* input. The preceding byte, at *BPST*'s position, should also be included for convenient comparison, instead of forcing MATLAB to read at the address location. The output will then be properly appended by providing the packet header end address as input *BPST*.

The finish and reset signal from the EBCOT to the MQ are currently automatically determined by the list lengths of context label and decision pairs. The complete list from one codeblock can easily be split in sections after a complete EBCOT execution. Each sectioned context label and decision list can then be sent to the MQ to append to a packet, with the MQ automatically *flush*ing and resetting between each list. To perform the sectioning procedure the context label values can simply be read to determine which pass they originate from. The values are unique for the pass types, and can therefore be processed without knowledge of anything else. The significance propagation coding and normal cleanup coding are easier identified if knowing the codeblock size, however, as counting the amount of context labels allows noticing the boundary between each bitplane.

The packet boundaries must be decided. The Data Orderer should switch which packet is being appended to by MQ encoding of the current bitplane. Information about the end of each coding pass is necessary, for instance signalled from the EBCOT.

6.3 Thoughts On JPEG2000 Standard

The JPEG2000 standard contains details regarding many aspects. The explanations are not easily interpreted for inexperienced participants. This is due to being a complex topic, but also because of the many sidetracks to explain extensions and special cases. It is a heavy read that can be misinterpreted. It appears to mainly be intended for decoder implementations.

A considerable amount of time has been spent to understand the content. The explanations presented in this thesis should assist future works.

6.4 Improvements

The following subsections discuss alternatives and improvements to the encoding operations. This included both changes to the implementation and general considerations. As the goal is to create a hardware solution, relevant optimizations should be considered. System parts and sub-modules should be placed in parallel if possible, to speed up execution time at the cost of area. A more streamlined pipeline will also contribute to this. It is however still early in the implementation development. A working solution should be constructed before initiating drastic performance increasing solutions. Too many things going on at a time will make progress slower, due to higher complexity.

The implementation is now intended for an FPGA. This raises the question of whether a hybrid software/hardware solution has been investigated. It may be more optimal to perform certain things in one type environment.

6.4.1 EBCOT

To better present the content of the individual coding passes for packet generation, the EBCOT could send an output signal after completing each coding pass. The Data Orderer should then count the desired amount for a packet, then switch to the next packet. In this case the Data Orderer must also trigger the MQ to perform a byte *flush* and reset its progress. In one circumstance the MQ can be modified to receive such reset signals. This will however be achievable by using the list lengths of the context label and decision pairs. The input list to the MQ should therefore be sectioned into packet lengths, instead of entire codeblock lengths, of encoded codeblock data. The pairs should therefore not be sent immediately upon discovery, but be temporarily stored in lists of various lengths.

The method for context label and decision pairs described above is presumed preferred, but this is for a software implementation. The end goal is a hardware implementation. The described method is a push method, because results are pushed to the next stage. With a pull method, only the requested material would be produced. Thus, the Encoder module would not spend time producing unused results, for instance if not all bitplanes are to be encoded. Implementing a pull method could therefore be beneficial for when using different compression rates. Additionally, the intermediate results would presumably be smaller, meaning less temporary internal memory. It may also aid to pipeline the system.

6.4.2 EBCOT In Parallel

Instances of the EBCOT sub-module could potentially execute in parallel in a hardware implementation. The codeblocks are read, then independently encoded. New data is thus generated without affecting the old data. Although the data write is not fully determined, it cannot be allowed to overwrite codeblocks before they are encoded. Thus, several EBCOT sub-modules can execute independently in parallel. The system must however be able to handle the data

write from several of these. The MQ may be a bottleneck, limiting the write for the parallel EBCOTs. The bitstreams must be appended to packets in appropriate orders, which could also be a limiting factor for the EBCOTs.

6.4.3 Coding Pass Optimizations

Internally, the EBCOT runs sequentially. The significance states of which the coding passes depend on are updated in a specific order. They order it is updated in is the main purpose of the EBCOT. The straightforward realization of the material presented in Section 3 uses three scan traversals for one bitplane. This is used in the implementation in Chapter 5. The three scans occur in a deep level of for-loops, iterating through tiles, decomposition levels and codeblocks, among others. Time can therefore be saved by merging the three scans into one. If merging is possible, that is. The results must of course still be in the same order. The magnitude refinement pass and the cleanup pass outputs must be temporarily stored in buffers until the significance propagation pass finishes. The significance propagation pass should be mostly unmodified, as it already executes first and is independent of the other passes.

There must be a one-way dependency for the updated significance states. The initial states and the updates from the significance propagation pass are used by all coding passes. The cleanup pass's updates are only used by the cleanup pass. The main issue with merging the three scans is that significance states no longer update at correct points in time. The 5 positions in the lower and right parts of the sliding window do not have an updated value from the significance propagation pass to be used in the other two passes. They would therefore have to either refresh their results, or delay their context label creation.

Refreshing means adapting the previous context label result to updates in its sliding window. Updates in the sliding window will affect sumH, sumV or sumD, but must not necessarily give a new context label. The value is however guaranteed to not decrease, as seen in Table 3.2 and Table 3.1. The value can therefore be incremented by an appropriate amount, e.g. any update near a context label of 14 gives +1. Alternatively the appropriate table is reverse looked up from the context label value, the updated neighbour is added to the directional sum, and the new context label is produced from the table.

The delay of the context label creation would last until the final neighbour is scanned and potentially updated. This is effectively a pipelined solution of the three scans, and not a merged solution. They still execute faster than the three scan solution. The pipeline solution will produce the context label as if the three scans ran separately, but the next pass runs as soon as the neighbours are stable. A result can be categorized as stable when it no longer changes upon any updates in its sliding window. Before this it is unstable.

Results can become stable before all neighbours are updated. It is determined by the value of the context label. For instance, with context label of 3 and only D_3 of the sliding window is unscanned, the context label is not modified for any update to D_3. Neither will the result update if the context labelis at the maximum values of 8, 15 or 16, where further updates do nothing. The results therefore become stable either when the last neighbour is updated, or when the result is at maximum value.

The pipeline solution will only be computed once. The refresh solution may be computed up to six times, if all neighbours below and to the right are updated. The pipeline must have separate

traversals, while the refresh solution must have conditions to refresh its results. Only D_0, H_0 and V_0 of the sliding window are guaranteed to not update. The refresh solution may actually just be a disguised pipeline solution. In effect, the refresh solution will always have to update the result at the current D_0 position. To refresh, the sliding window for the D_0 position is used. This is the pipeline with a slight delay.

The magnitude refinement pass is stable in two of three cases, as seen in Table 3.2. It is only in the case where both it is the first refinement and the sum of neighbours is 0, that it is unstable. If any neighbours then updates, it is incremented to 15 and immediately becomes stable. The magnitude refinement pass can therefore be determined simultaneously to the significance propagation pass, and refreshed if any neighbour updates.

Results become stable once the scan pattern has progressed sufficiently far away from them. The cost considerations for the merged solution requires the approximate maximum of the codeblock's width plus the column's height unstable context label results. The column height is always 4, and the codeblock width is variable. This amount of unstable results do not require too much temporary storage. The merged solution has potential to be three times as fast, but overhead control signals will most likely not give the ideal speedup.

Perhaps the merged solution makes it easier to spot four in a row for the run-lengths, as opposed to performing the double column scan currently implemented in matlab. The context label list is looked at in hindsight, and four cleanup pass results are replaced by the run-length context label. Hindsight would make the solution partially not merged, but the solution is not a goal in itself. Only by improving the overall execution should it be considered.

6.4.4 MQ

The pointer start variable *BPST* is not properly used in this implementation. It should point to the base address in memory of a byte preceding the current bitstream. This is assumed to contain the final byte result from the packet header or the previous bitstream encoding. This implementation has only been tested with a single codeblock encoding at a time. Thus, there has only ever been one bitstream result, and no packets are created yet. The *BPST* value is therefore not correctly implemented. This in turn affects the *BP* pointer and the *B* byte result. The *if* condition in the initialization is also affected, as it now cannot detect if the previous byte was `0xFF`. The behaviour and result for the output must therefore be produced before using the MQ implementation. The bitstream is now sent as a function output, instead of being written to a memory location.

A-reg and *C*-reg can be simplified in the future FPGA implementation. They are currently assigned 32 bits registers. In the hardware implementation they can be assigned exactly the bits they use: respectively 16 bits and 28 bits. The reason is because the 16 MSBs of *A*-reg are always 0, and likewise for the 4 MSBs of *C*-reg. Addition operations can thus also be simplified, as the reduced bits only need to propagate a carry bit with half-adders.

6.4.5 Stream Sorting

Old data may be overwritten in a practical implementation to utilize memory. The packet generation decides the data write location. If packets overwrite old data on a codeblock basis, the

resulting packets will be segmented across different codeblock locations due to being appended piece by piece from different codeblocks. The codeblocks only become old data after being encoded. Accessing the packets afterwards may give complex address patterns. With enough resources, this would not be a problem. In an ideal system with infinite memory the packets are built as separate streams. These are collected in a sorted, continuous datastream only after the packets are complete.

The sorting is based on packets. The packet headers requires knowledge of sizes of the image pieces. These are already known to the Partitioner. The packet headers can therefore be constructed by the Partitioner, and forwarded to the Data Orderer only when they are complete packets. The packets are then only sorted in the Data Orderer. This allows the *BPST* and packets to always be sent forward in the compression chain, as opposed to the Data Orderer sending backwards. Sending forward should make future pipelining optimizations easier to implement.

Chapter 7

Conclusion

The EBCOT sub-module is implemented in MATLAB and tested thoroughly. It is however not compared towards already existing solutions, thus it is not completely verified. The context labels and decisions produced are currently generated in a continuous list for one entire code-block. Because of how the packets are created, the continuous list should be split into sections for each packet. The splitting should be sufficiently easy to do after the EBCOT, meaning the sub-module does not need modifications, due to the unique values for context labels. Alternatively, the EBCOT can be modified to produce tokens at each potential section.

The MATLAB implementation of the MQ sub-module is almost complete. The packet headers are missing, which is where the output will be appended to. The MQ bitstream output is affected by the final byte of the packet header. The append operation can potentially be delayed until the Data Orderer sub-module, which must then perform the bit-stuffing operation now performed in the MQ. It will however be difficult to split the bitstream into different packets, thus the Data Orderer should not perform this. Instead, the MQ implementation should be modified to take the output address *BPST* of the packet header as input.

A Partitioner sub-module is implemented in MATLAB as a simple way of accessing each code-block, with appropriate sizes for each sub-band and precinct.

The Encoder module is not fully implemented in MATLAB. The currently connected sub-modules are the EBCOT and MQ. The Partitioner could be modified slightly to be included in the pipeline. The output behaviour is not yet determined, which is part of the unimplemented Data Orderer. After this stage the produced result will be a sorted datastream, intended for finalizing in the unimplemented Packetizer. The implemented sub-modules also have accompanying test files.

Due to the current state of the MATLAB Encoder module, the VHDL implementation for the FPGA is not started.

A considerable amount of time has been spent to understand the content. The explanations presented in this thesis should assist future works.

Chapter 8

Future Work

The next item to be considered is the packet formation. Producing packets will enable fulfilling the MQ and the Partitioner. The packets order to be sorted must be investigated, by looking at layers and progression orders. After this there should be sufficient knowledge to implement the Data Orderer. Headers and markers must be investigated next for a Packetizer implementation. Several values must be decided, such as codeblock sizes.

The openJPEG [4] implementation is recommended for acquiring debugging information and design inspiration. It can be used to verify the parts, the complete Encoder and the entire JPEG2000 implementation.

With the model in place, the VHDL implementation should be built. The JPEG2000 system will then be one long chain, which must surely need validation of proper behaviour. The compression system will still require the Rate Control module and an interface.

Bibliography

[1] ISO/IEC JTC 1/SC 29. *NEK ISO/IEC 15444-1:2016*, 2016. Information technology. JPEG 2000 image coding system: Core coding system. https://www.iso.org/standard/70018.html. Retrieved 8th March, 2018.

[2] Ole Kristian Hamre Sørlie. *JPEG2000 Image Compression in Hardware*, 2017. Master's thesis, NTNU, Norway. https://brage.bibsys.no/xmlui/handle/11250/2467026. Retrieved 19th December, 2017.

[3] The MathWorks, Inc. *MATLAB R2017a*, 2017. Computer application for developing with the MATLAB programming language. https://www.mathworks.com/. Retrieved 10th June, 2018.

[4] uclouvain et al. *OpenJPEG*, 2015. Source code repository. http://www.openjpeg.org/. Retrieved 25th May, 2018.

Appendix A

Source Files

The files for the MATLAB implementation can be found at <https://github.com/torevb/JPEG2000-Encoder>.