NTNU
Norwegian University of
Science and Technology

A study of hardware compression of images

Per Arne Rønning

Master of Science in Electronics

Submission date:	June 2016
Supervisor:	Bjørn B. Larsen, IET
Co-supervisor:	Roger Birkeland, IET
	Amund Gjersvik, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

ABSTRACT

In this paper, we will take a closer look at the possibility of compressing images to the JPEG 2000 standard using an FPGA's hardware architecture. This FPGA will be put into a satellite, which is going to orbit the earth in a low earth orbit, so considerations must be made regarding this.

The implementation was done in VHDL, and the created modules were simulated using test benches. The simulations were mainly conducted with a generated input from the image sensor. It was run through the module, and the output was analyzed.

These simulations point in the direction that the modules work as intended, but there is still further work that needs to be done, in order for the complete system to be implemented in a physical FPGA and launched into space.

SAMMENDRAG

I denne oppgaven vil vi ta en nærmere kikk på mulighetene for å komprimere bilder til JPEG 2000 standarden ved hjelp av en FPGA sin hardware-arkitektur. Denne FPGAen vil bli plassert inni en satellitt, som vil gå i en low earth orbit, så det er hensyn som må tas på grunn av dette.

Implementasjonen ble gjort i VHDL, og modulene som ble laget ble simulert ved hjelp av testbenker. Simuleringene ble hovedsakelig gjennomført med generert input fra bildesensoren. Disse dataene ble kjørt gjennom modulene, og resultatene ble analysert.

Disse simuleringene peker i den retning at modulene virker som planlagt, men at det er fortsatt mer arbeid som kreves for at systemet skal kunne implementeres i en fysisk FPGA og sendt ut i verdensrommet.

ACKNOWLEDGMENTS

I would like to thank Bjørn B. Larsen for his support and guidance. I would like to thank Anne Elisabeth Vallestad for her technical and professional help. I would also like to thank Ingunn J. Vallestad for her help regarding the report and supporting me throughout the process. I would also give a thank to Kristine Thevik and Robert Balteskard for their help in proof reading the report.

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS AND SYMBOLS

ASIC	Application-specific integrated circuit.
DWT	discrete wavelet transfer.
FPGA	Field Programmable Gate Array.
HDL	Hardware Descriptive Language.
MSB	Most significant bit.
SEB	Single event burn up.
SEFI	Single event funtion interrupt.
SEGR	Single event gate rupture.
SEL	Single event latch up.
SET	Single event transient.
SEU	Single event upset.
VHDL	VHSIC Hardware Description Language.
VHSIC	Very High Speed Integrated Circuit.

0.1 Problem definition

Review of the software and hardware description of the camera module for NTNU Test Satellite (NUTS).

The current focus of the NUTS project is to finish the design and build hardware for an integrated engineering model. The student should focus on the system design of the camera module in general, with particular focus on how to split the functionality between the hardware (FPGA) and the micro controller of the payload module.

The camera module must be designed to be reliable, as maintenance is impossible after launch. Challenges due to the space environment, such as temperature cycles, radiation and vacuum must be identified and discussed. In areas where mitigation of such problems is possible, solutions should be presented. Whether the solutions are to be implemented should be based on a cost/benefit analysis.

Key tasks will be:

- Explore how the Jasper JPEG 2000 algorithm for image compression can be ported to the FPGA, instead of being run on the MCU.
- Compare the different implementations (on FPGA and MCU) by looking at implications for implementation time, run-time, power consumption and hardware cost.
- Participate in the hardware design process of the camera module.

CHAPTER 1
INTRODUCTION

This report will deal with the possibility of compressing images into JPEG 2000 using an FPGA. It will also discuss some of the problems and issues we have to consider when an FPGA is going to space, where it will be operating in a high radiation environment without the possibility of recovery.

Figure 1.1 shows the image compressor, which takes data from the image sensor and processes them into a compressed image.

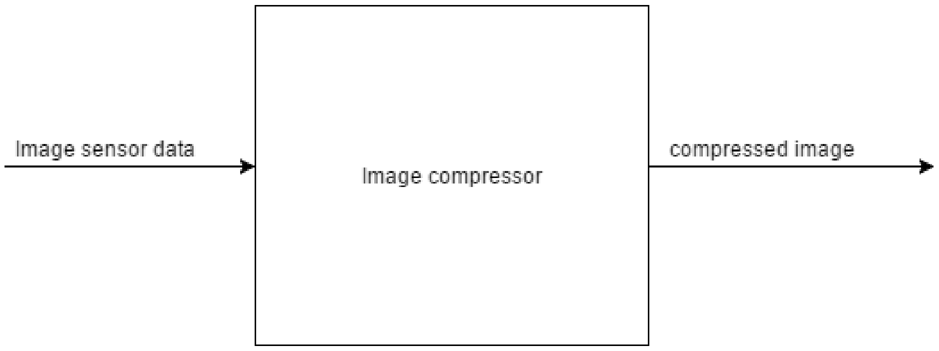


Figure 1.1: Image compressor

Figure 1.2 shows a suggested setup of the completed system, with all the necessary modules. We can see that the micro controller is what communicates with the rest of the system. The micro controller is connected to the FPGA, multiplexer, and the power sequencer. The FPGA will handle the image compression, and will store the compressed image in the flash memory that it shares with the micro controller. It is also connected to and controlling the image sensor and its power sequencer.

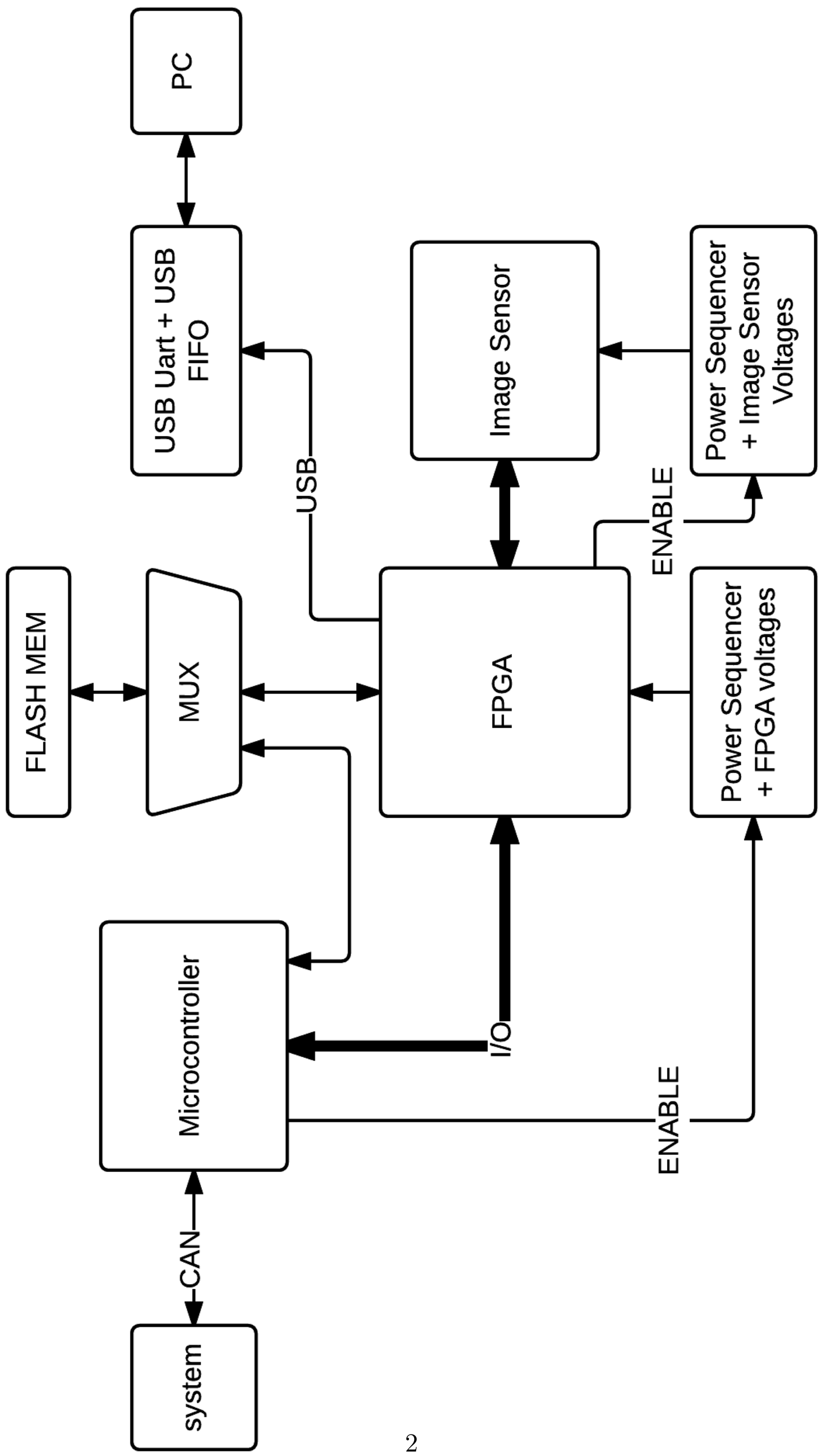


Figure 1.2: Overview of the total payload subsystem of the NUTS satellite

CHAPTER 2
THEORY

2.1 FPGA

FPGA stands for Field Programmable Gate Array, and is an integrated circuit which can be configured after production. They are semiconductor devices that are based around a matrix of configurable logic blocks. These blocks are connected with each other using programmable interconnects. [1]

It is usually configured by using Hardware Descriptive Language (HDL). The FPGA can also be reconfigured several times, and requires a bit file when powering up, to know what it is configured to do.

2.2 ASIC

Application Specific Integrated Circuit (ASIC) is a customized integrated circuit, usually made for a single task, but unlike the FPGA it cannot be configured after production.

2.3 VHDL

VHDL is a Hardware Descriptive Language, used to describe and design hardware.

A VHDL program can be used to program FPGAs or design ASICs. The written code is then converted into hardware which can perform what this code describes.

2.3.1 STD_LOGIC_VECTOR

A STD_LOGIC_VECTOR is a predetermined type, that can be found in the Std_Logic_1164 package where they are described as "a standard one-dimensional array type with each element being of the Std_Logic type." A Std_Logic type is representing a single bit.

2.3.2 Division of STD_LOGIC_VECTOR

Dividing signals and variables of the type STD_LOGIC_VECTOR is usually not allowed by VHDL compilers, because of the varying resolution length of the results, and the fact that division cannot be performed on vectors.

But there are some things that can be done in order to make certain divisions. For example, when dividing by 2, right shifting the vector once gives the result of the division. One will lose the LSB (least significant bit) in this process, as long as the LSB is situated at the right, which is the norm. Dividing by 4 is the same as dividing by 2 twice, so therefore one can right shift the vector by two spaces and get the desired result.

This way of dividing is only viable when the divisor is a power of two, and the division is then a right shift equal number of times, to the exponent of two.

$$\frac{\sum_{n=0}^{2^k-1} n}{2^k}, k = numberofbits \tag{2.1}$$

Using equation 2.1 shows the average value of a 12 bit number is 2047,5. But since we are right shifting the number by two places the potential error will be in comparison to a 10 bit number. Using equation 2.1 again, we get that the average value is 511,5. The error will then on average be off by 0,14%.

2.3.3 Soft Processors

A soft processor is a micro processor that is realized in an FPGA, or as Xilinx puts it, "A soft processor is an Intellectual Property (IP) core that is implemented using the logic primitives of the FPGA. Key benefits of using a soft processor include configurability to trade between price and performance, faster time to market, easy integration with the FPGA fabric, and avoiding obsolescence." [2]

2.4 Colour transform

Colour transform comes in two different forms, irreversible and reversible colour transform. The colour transform requires that the R, G and B are of the same bit-depth and dimensions [3, page 420]. R, G and B are the red, green and blue layer of an image, with each of them representing the amount of said colour in any given pixel. Equation 2.3 shows us that Cr is the difference between the red and green, and Cb is the difference between blue and green. γ is often referred to as luminance, and Cb and Cr combined are referred to as chrominance.

$$\begin{bmatrix} \gamma \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.16875 & -0.33126 & 0.500 \\ 0.500 & -041869 & -0.08131 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.2}$$

$$\gamma = \left\lfloor \frac{R + 2G + B}{4} \right\rfloor, \quad Cb = B - G, \quad Cr = R - G \tag{2.3}$$

Equation 2.2 is the equation for the irreversible colour transform, while 2.3 is for the reversible colour transform. γ can also be referred to as Y, this is seen in figure 2.1.

Figure 2.1 shows us the result of a colour transform. We observe that the Y channel is the image in black and white.

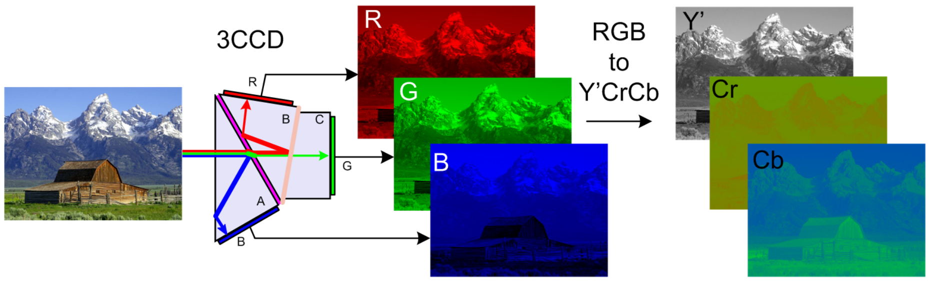


Figure 2.1: Illustration of how an image is deconstructed and the colour transform result. [4]

2.4.1 Luminance

In physics, luminance is the physical measurement of light projected from an area, measured from a specific angle. In digital imagery, it is the brightness of a pixel. This is the only value that is necessary in a black and white picture. The luminance is normally represented by values from 0 to 1, where 1 represents white, and 0 is black. All the values between these represent gray of varying darkness. [5]

2.4.2 Chrominance

Chrominance is the value that explains the colours of the image. The chrominance is usually represented by the difference between the colour and the brightness of the pixel, where the brightness is represented by the luminance. The chrominance consists of the Cb and Cr channels, and can then represent any colour. Figure 2.2 illustrates how Cb and Cr represent the different colours, where each colour is a point in the plane given by Cb and Cr, who can be between -1 and 1. By adjusting the γ we get the different shades of the colours.

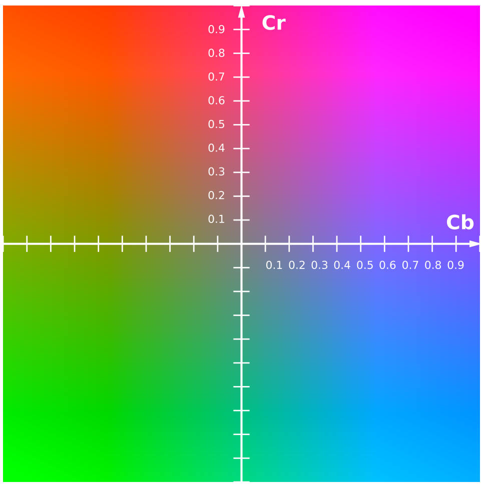


Figure 2.2: The CbCr plane with a constant luminance $\gamma = 0.5$. [6]

2.5 Image compression

Image compression is the process of making an image file smaller. This can be done in several ways, but there are two very important approaches: Lossless and lossy. The difference between these two algorithms is whether or not data is lost in the compression process.

2.5.1 Lossless compression

In lossless compression, no data is lost. This is done by storing the data in a smarter way than the uncompressed data is. The fact that no data is lost, but the file size still decreases, is achievable by making sure the original file can be reconstructed from the compressed file.

2.5.2 Lossy compression

Lossy compression loses data in the compression process, but will get a higher compression rate than lossless compression. In image compression, much of the lost data is irrelevant to the viewer.

2.6 JPEG 2000

JPEG 2000 is an image format that can be both lossless and lossy [7]. JPEG 2000 uses wavelet compression, which makes it possible for it to store data, and extract pictures of different resolutions from it. In [8] we can read about how JPEG 2000 is working compared to other image formats, and why JPEG 2000 is chosen for this project.

2.6.1 Wavelet compression

Wavelet compression is a way of compressing images and video. It compresses images by looking at transients and does a wavelet transfer of this. This results in less information needed to represent this. This process is done several times over for a better and more compressed image, without losing information. This results in an image that has degrees of resolution according to the amount of data you transmit. The first data will be a low resolution image, whereas the later data improves the quality of the image by adding higher frequency information.

Two dimensional discrete wavelet transfer

The two dimensional discrete wavelet transfer (DWT) is made from simple one dimensional building blocks, which convert a finite length input sequence, x[n], into two sub band sequences y0[n] and y1[n]. These two sub bands can be seen as a low pass and high pass sub band, respectively, who have then been sub-sampled by disregarding every second sample [3, page 423]. The high pass and low pass filters must be related to each other and are quadrature mirror filters. This causes half of the frequencies to go to each of the outputs. Since each of the outputs now have half of their frequencies removed, half of the samples can be removed according to Nyquist's rule, and therefore every second sample is removed. The high frequency and low frequency outputs are then processed again through another set of quadrature mirror filters and down-sampled. The result of this is four outputs called LL, HL, LH, HH, where L stands for low passed filtered, and H stands for high passed filtered. HL is then the output from the signal that went first through the high pass filter, and thereafter through the low pass filter. Figure 2.3 show how the input is going through two filters for each of the outputs.

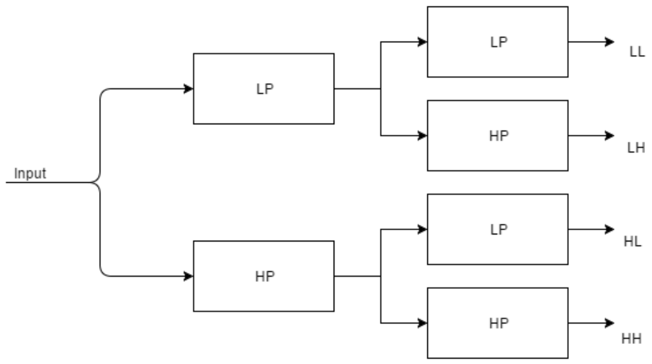


Figure 2.3: 2 dimensional DWT

Multilevel two dimensional discrete wavelet transfer

To do a multilevel two dimensional discrete wavelet transfer, do a two dimensional DWT of the LL output of an already transformed signal. Repeat this over and over again until you either run out of samples or wish to stop for some other reason [3, page 428-430]. Figure 2.4 illustrates the process, and it shows how the image is compressed over and over again.

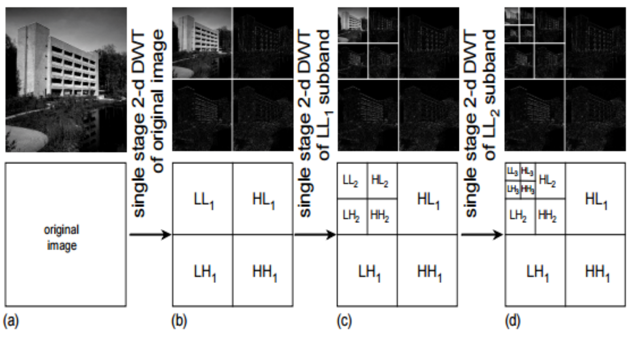

original image
single stage 2-d DWT of original image
LL1
HL1
LH1
HH1
single stage 2-d DWT of LL1 subband
LL2
HL2
LH2
HH2
single stage 2-d DWT of LL2 subband
LL3
HL3
LH3
HH3
(a)
(b)
(c)
(d)

Figure 2.4: Illustration of how DWT works with upper left image being compressed over and over again [9]

2.7 Gamma correction

The human eye, under normal lighting conditions, is better able to differentiate between the darker lighting levels than the lighter ones. This ability follows a non-linear curve, and this is what the gamma correction tries to simulate. Without the gamma correction, humans will find the images looking artificial and erroneous.

Gamma adjustment is then the process of taking the linear colour range observed by the camera and adjusting it into nonlinear values, more similar to the way a human observes the world.

Gamma correction value is given by:

$$Z = X^Y \tag{2.4}$$

Where X is the original value, Y is the gamma adjustment and Z is the scaled result.

2.8 Histogram

The computation of a histogram is to put individual pixel values into different predetermined bins. This can be done either before or after the gamma correction. The histogram will then have information of the colour composition of the image, which can be used for further assessment of how good the image quality is.

The quality of the image can be seen by analyzing the distribution of the colour in the image. In a good image, there will be a good distribution of the colour levels, and there should not be indication of colours only being in the darker or brighter area.

2.9 Min-Max computation

Min-Max computation is, as the name suggests, finding the lowest and highest values of the pixels from the captured image. The information gathered from the min-max computation can be used to see if the image is over- or underexposed. In a good picture, neither the minimum nor the maximum pixel values should be at the lowest or highest possible levels, respectively. A minimum or maximum value at an extreme level means that the image is under- or overexposed, this in turn means that the shutter time of the camera is not optimal, and should be changed.

2.10 Unsigned extension

Unsigned extension is to extend the size of an unsigned bit vector. To do this is to pad one of the edges with zeros. One puts them in the most significant bit (MSB) position, in order to retain the original value, as long as it is not a signed variable or floating point. For example, a 12 bit variable with the value 111100001111, with the MSB on the left side, will become 0000111100001111 if it gets extended to 16 bits.

2.11 Demosaicing

When taking a picture, the pixels of a camera are covered with individual filters, to filter out certain colours. This means that every pixel only sees one of the primary colours (red, blue or green). Hence, for each pixel we have to estimate the colour values of the two colours not present. For example we must find the blue and red value of a pixel that only sees green. The process of doing this is called demosaicing.

Because of the way the human eye works, green will be the most important colour, as this is the colour we differentiate the easiest. Therefore, there are twice as many green pixels as there are red or blue. This also makes it much easier to calculate the green values of red or blue pixels, as one only needs to calculate the average of the neighbouring green pixels to get a good estimate.

For the estimation of the red and blue pixels, we must do it in two stages. First we estimate the red values of the blue pixels, and the blue value of the red pixels. If we look at figure 2.5, we can see that the blue pixels are surrounded by red pixels diagonally, and that the same goes for the red pixel in correlation to blue. To estimate the red and blue values we will then take the average of these diagonally surrounding pixels. After this is done, we will have a similar pattern for red and blue pixels, as we have for green. To find the remaining unknown blue and red values, who are situated over the original green pixels, we can estimate these values in the same manner as we did for estimating the green value [10]. We can see an illustration of the demosaicing process in figure 2.5.

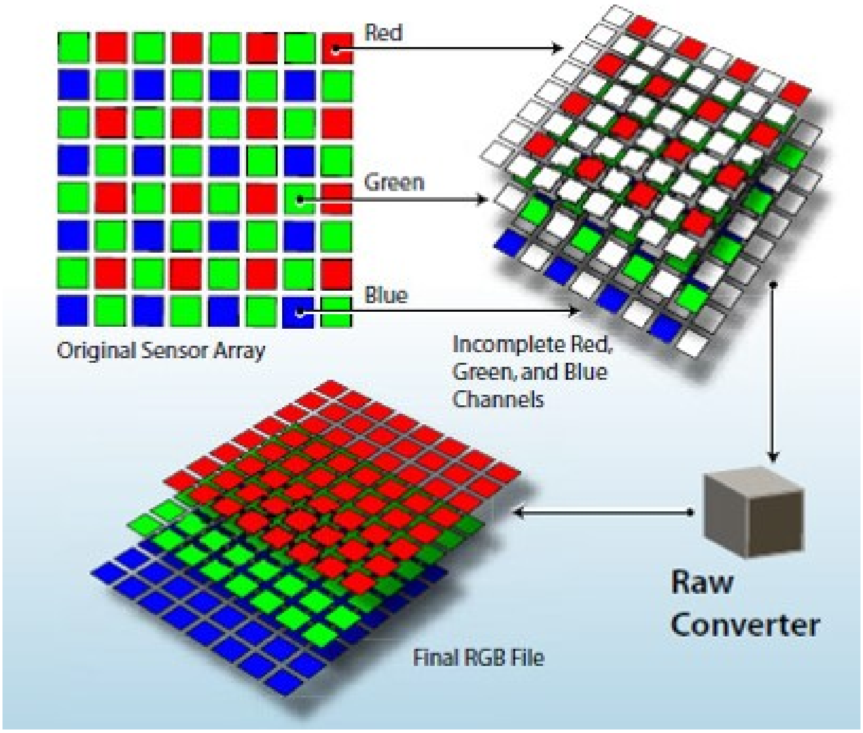


Figure 2.5: Going from a three channel image, to three one channel images. [11]

2.12 Ring buffer

A ring buffer is a way of temporarily storing data. The ring buffer starts storing data in an arbitrary place, and stores the consecutive data in the next spaces in rising order. When the memory reaches the highest address it is allowed to access, it starts again at the lowest address and continues storing data. Then when it has stored data at all the available addresses it will overwrite the oldest data.

2.13 Space Radiation

NASA says “Radiation may be defined as energy in transit in the form of high-speed particles and electromagnetic waves.” They can have different forms and can be divided into two main groups: ionizing and non-ionizing radiation. Space radiation consists mainly of the ionizing kind. This is the radiation with the highest energy, and therefore the highest damage potential when interacting with objects, such as an FPGA or other circuitry. [12]

2.13.1 Radiation Effects On FPGA

The ionizing radiation can have unwanted effects on transistors, and may therefore prevent the FPGA from doing its task properly and as expected. These effects are caused by the radiation hitting critical regions of the transistors, and may cause various failures or single event effects. The sensitivity of a component to the space radiation and single event effects differs. SRAM is very susceptible to it, while if one has radiation hardened hardware it may be almost immune to it. The single event effects are divided into to main categories; hard and soft errors. [13].

Soft errors

The soft errors are self correcting over time or by rewriting a memory element. There are three subclasses of soft errors: single event transient (SET), single event upsets (SEU) and single event function interrupts (SEFI).

SET A high energy particle impacts a combinatorial path and induces a voltage. If this impact is of great enough magnitude and length, it may cause an error that propagates through the combinatorial path, and ends with an erroneous result.

SEU The high energy particle changes the state of a memory element and can be single bit or multibit upsets.

SEFI This is when the device functionality is no longer the same as intended, and a power cycle may be the only way of getting back to the correct way of operations.

Hard errors

These are errors with lasting effects on the device. There are three subclasses of hard errors: single event latch up (SEL), single event burn up (SEB), and single event gate rupture (SEGR).

SEL A circuit gets a latch up because of induced radiation, and this latch up may or may not be cleared by a power cycle. A latch up is a short circuit in an integrated circuit where a low-impedance path is generated between the power and ground in a MOSFET. This causes the MOSFET to function incorrectly, and can cause over-currents, leading to the destruction of the circuit or component.

SEB A high energy ion impacts a transistor source, causing it to become forward biased.

SEGR A rupture in the gate ion oxide caused by an impact from a high energy ion.

CHAPTER 3
REQUIREMENTS

The NUTS project has several requirements for its modules and submodules. For the camera module, the requirements are listed in table 3.1.

ID	Specification
R05-CAM-COM-001	**COM = Internal Communication Bus** Must be able to communicate with the other sub systems using the back plane
R05-CAM-COM-002	Must be able to capture image on request
R05-CAM-COM-003	Must be able to send images to the OBC on request
R05-CAM-COM-004	Must be able to change image sensor parameters on request
R05-CAM-CPR-001	**CPR = Compression of images** Must to be able to read images from the image sensor and compress them to reduce file size
R05-CAM-CPR-002	Must be able to produce thumbnails
R05-CAM-CPR-003	Must be able to produce histograms of pixel values
R05-CAM-CPR-004	Must be able to detect and not process unwanted images (Pictures of space or the sun)
R05-CAM-CPR-005	Must be able to make gamma corrections on captured images
R05-CAM-IMG-001	IMG = Storing of images Must be able to store compressed images to local memory
R05-CAM-IMG-002	Must be able to retrieve images from local memory
R05-CAM-REP-001	**REP = Reprogramming** The compression logic should be able to be reprogrammed in flight

Table 3.1: Camera module requirements

Some of these requirements are not applicable to my work and will be handled by other people working on this module now and in the future. My focus will be towards the compression of images requirements (R04-CAM-CPR-001 to R04-CAM-CPR-005).

CHAPTER 4
APPROACH

The earlier work on the camera module for the satellite, done by Andreas Bertheussen and Thomas Nornes [14,15], had a working prototype for image capturing and compression. This prototype had a soft-processor, which had a Linux operating system, and a software implementation of the image compression written in C.

This way of doing image compression is not the most ideal, and is not ideal for a space mission either. Because of the difficulty of problem fixing after launch, the module should have as few possible states as possible. This is so one can have a better overview of the system, and therefore know what happens in all situations.

There are 4 main ways to improve on the existing code

1. To change all the existing C-code into VHDL code
2. To change parts of the C-code into VHDL code
3. Review the C-code and remove superfluous parts
4. Make a new VHDL code, by converting existing jasper-codec files, and not based on the earlier work of Andreas and Thomas.

First I tried to convert the existing C-code into VHDL, but several of the functions used in the existing code were not supported by the converting software, Vivado HLS [16]. This resulted in me trying to convert an existing jasper-codec code [17], but this led to the same problem. The final solution was to write the VHDL for the needed functions by hand, and not relying on conversion software.

The plan was to develop the codes with the existing structure still in place, so that when a picture was taken, the original system would run in parallel with the new modules and output data would be compared on the fly. This became problematic during testing, and I ended up testing each module individually.

Figure 4.1 shows how the compression pipeline is supposed to look [14][page 9]. In this pipeline we can see all the modules we need for a complete compression of a captured image into JPEG 2000 format. In the sensor capture block, there needs to be a demosaicing process to be able to do the rest of the processes correctly.

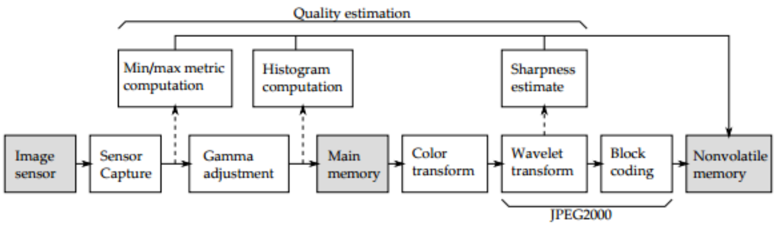


Figure 4.1: Suggested pipeline [14]

4.1 Unsigned extender

The task of this module is to take the incoming bit stream from the camera, and make it into a bit vector. The incoming data is 12 bits long, and after recommendation from [14], the vector is extended to be 16 bits long.

The module also has an output ready signal which will be used for the module that receives its output, so that it only receives bit vectors that are correctly put together.

The module uses a counter, which counts each incoming bit. This counter is used to place the incoming bits in the correct position in the vector. It is also used to detect when the last bit for the pixel has been received, and therefore knowing when it is possible to push out a completed vector.

4.2 Demosaicing version 1

At the start of this module is the interface for gathering pixel values from the camera. Detection of when to start capturing is relatively easy. One only needs to look for the values of FV and LV to be high as shown in figure 4.2, taken from [18, page 13].

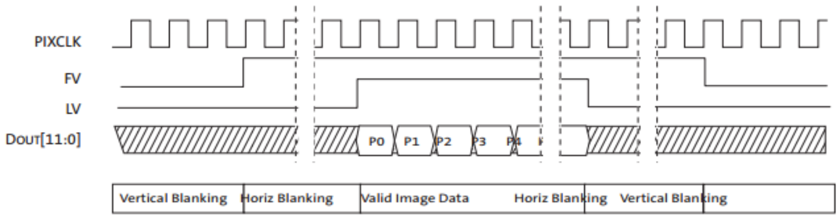


Figure 4.2: Sending of valid frame

I started making the demosaicing module by finding the green values. Figure 4.4 shows the algorithm for estimating the missing green values. To find if a pixel is empty or not is just a matter of looking at the pixel position. From the datasheet we can find which pixel positions correspond to which colour pixels. In figure 4.3, the position of the pixels in the image sensor is shown. The pixel position is indicated by i and j, where i represents the horizontal position, and j represents the vertical position.

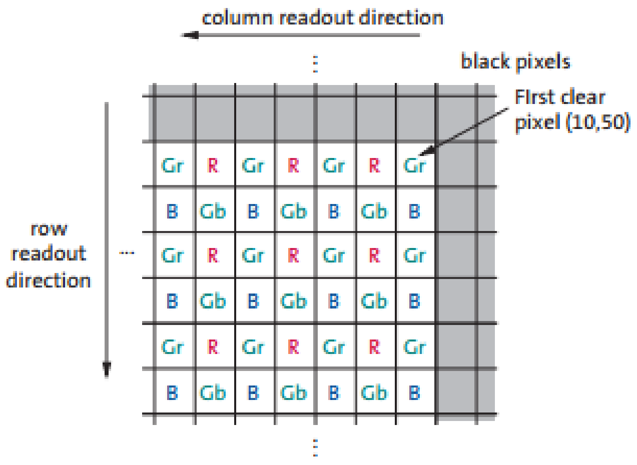


Figure 4.3: Overview from [18] of pixel positions

The special cases refer to the edges of the image. Since they are not completely surrounded by neighbouring pixels, we have to estimate their values in a different manner than the non-edge pixels.

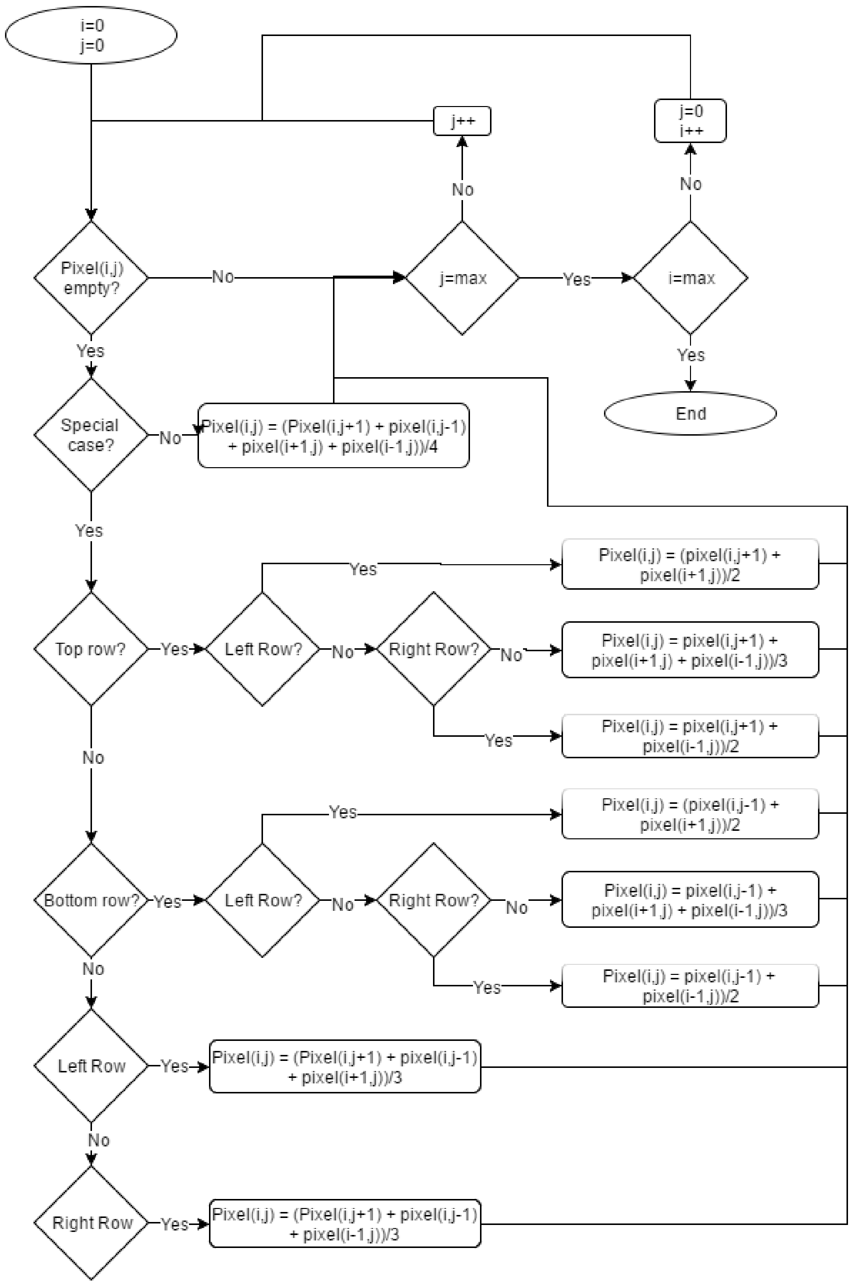

i=0
j=0
j++
j=0
i++
No
No
Pixel(i,j) empty?
No
j=max
Yes
i=max
Yes
End
Yes
Special case?
No
Pixel(i,j) = (Pixel(i,j+1) + pixel(i,j-1) + pixel(i+1,j) + pixel(i-1,j))/4
Yes
Yes
Pixel(i,j) = (pixel(i,j+1) + pixel(i+1,j))/2
Top row?
Yes
Left Row?
No
Right Row?
No
Pixel(i,j) = pixel(i,j+1) + pixel(i+1,j) + pixel(i-1,j))/3
Yes
Pixel(i,j) = pixel(i,j+1) + pixel(i-1,j))/2
No
Yes
Pixel(i,j) = (pixel(i,j-1) + pixel(i+1,j))/2
Bottom row?
Yes
Left Row?
No
Right Row?
No
Pixel(i,j) = pixel(i,j-1) + pixel(i+1,j) + pixel(i-1,j))/3
Yes
Pixel(i,j) = pixel(i,j-1) + pixel(i-1,j))/2
No
Left Row
Yes
Pixel(i,j) = (Pixel(i,j+1) + pixel(i,j-1) + pixel(i+1,j))/3
No
Right Row
Yes
Pixel(i,j) = (Pixel(i,j+1) + pixel(i,j-1) + pixel(i-1,j))/3

Figure 4.4: Flowchart of the process of finding the missing values of the green pixels.

To estimate the red and blue values, we must take the average of the diagonals, and then find the rest of the values, using the same method as for the green values.

In figure 4.5 one can see how the missing blue values are estimated. First the blue value of the red pixels are estimated, because they are surrounded by blue pixels on their diagonals. After this is done, the blue value of the green pixel are estimated by taking the average of the horizontal and vertical neighbours. The red pixels have changed colour to purple, to represent that they at that moment have both a known red value, and an estimated blue value.

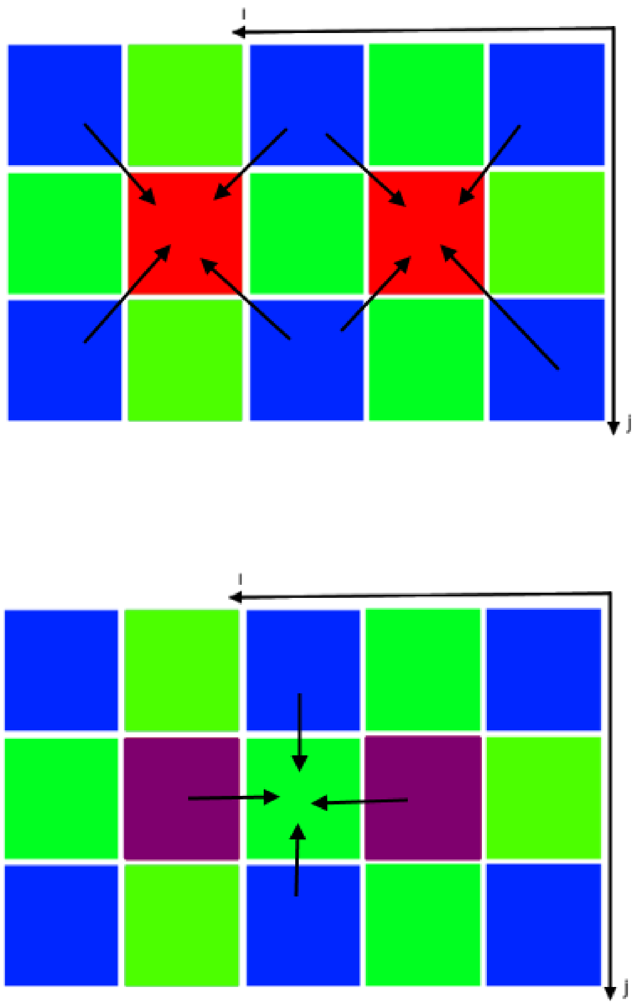
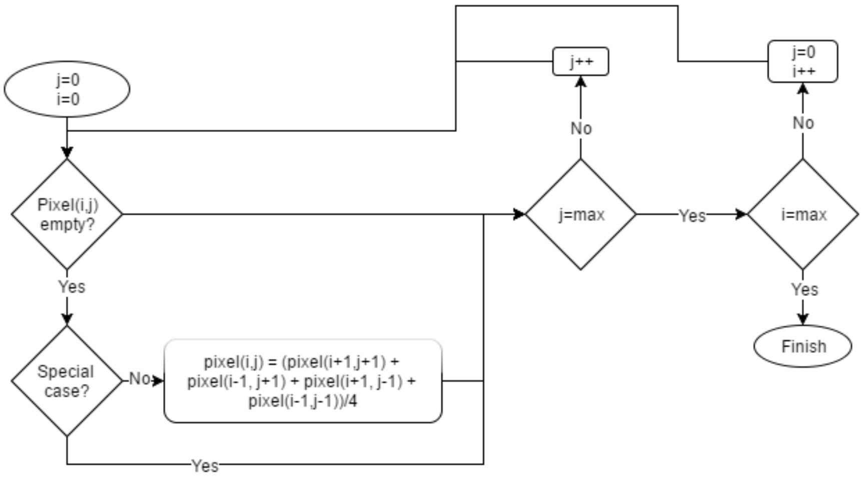


Figure 4.5: Estimating the blue values

This means that we can do the first estimations for the red and blue layer at the same time as the green layer is being estimated. See figure 4.6



j=0
i=0
Pixel(i,j) empty?
Yes
Special case?
No
pixel(i,j) = (pixel(i+1,j+1) + pixel(i-1, j+1) + pixel(i+1, j-1) + pixel(i-1,j-1))/4
Yes
j++
No
j=max
Yes
i=max
No
j=0
i++
Yes
Finish

Figure 4.6: Flowchart for finding the first values for red or blue.

As we can see in the datasheet for the camera chip [18], we have the format of 2592 x 1944 pixels that are active, but the active pixels do not start in the point (0,0). The first readable pixel exists in the (10,50), but this is still not one of the active pixels. They start at (16,54). This was taken care of by the camera chip itself, as it explains [18, page: 13], readout of the pixel values can wait until the two correct signals are high, and therefore only exports active pixels. The pixel readout position (0,0), the first pixel to be read, will be in the top right corner. This is important to know in order to correctly store the image and handle the special cases during the demosaicing.

The algorithm for estimating the green values, shown in figure 4.4, is well suited for a VHDL design in space. As stated earlier, the less complex the better. Therefore the module was simplified, by postponing the special cases (edge pixels) until after the non-edge pixels had been processed in full.

Figure 4.4 and 4.6 show that except for the special cases we need division by 4. We choose to do this by a right-shift by two to get the correct values. The values that are handled are 12 bits long and unsigned. The maximum error of a right shift of two bits is 0.75.

Chapter 2.3.2 shows that this is a feasible way of doing the necessary division of the numbers, as the potential errors are small enough to not matter.

4.2.1 Processes

The algorithm is realized in VHDL and this consists of several processes. The code can be found at `https://peraro90@bitbucket.org/peraro90/satellite.git`

Figure 4.7 shows the state machine that is being realized in the demosaicing module.

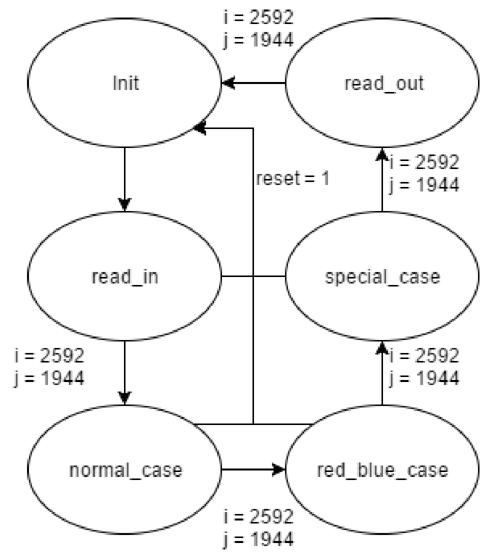


Figure 4.7: Demosaicing state machine

state_changer

This process handles the changing of states, and in addition it contains the counters for the signals i and j.

The signals i and j are used in the layering process to determine the position in the array to store, estimate and read values from. The i variable is indicating the column position while j is indicating the row.

nstate

This process is a companion of the state_changer and is responsible for deciding what the next state will be, so that state_changer later can change the state into the correct one.

layering

In this process the incoming data is captured. Furthermore, it estimates the values for the three layers of the output data. It also generates the output.

It also interprets the state machine that nstate and state_changer drives. This is what happens during the six states:

init This is the first state, and the default state after a reset. Its main task is to set initial vaules, and to be a known state to start in.

read_in The read_in is where the incoming data from the camera will be sent to memory before further operations is done on that data.

read_out Usually the final state of a demosaicing process, unless a reset is applied during the run. Here the estimated data is sent out from the module so it can be further processed.

normal_case This is where most of the estimation happens. The majority of the green layer is estimated here, with the exception of the edges. Half of the estimation of the red and blue layer also happens here, with the exception of the edges and the red and blue values in the green pixels.

red_blue_case This case is where the remaining red and blue values are estimated on the basis of the known values and the previously estimated ones.

special_case This is where the edges are taken care of. The edges can not be estimated in the same manner as the other pixels, and are therefore only a copy of a neighbouring pixel instead of an average of the four neighbouring pixels of the same colour. This deviates from the flowchart in figure 4.4, as this required to find the average of the two or three neighbouring pixels depending on the position of the pixel.

4.3 Demosaicing version 2

This version uses many of the same algorithms as version 1, but it uses ring buffers instead of storing all the incoming data.

By doing this, we get parallelism, which decreases the time from completed reading of the input to the generated output. The estimation of the unknown pixels can in principle be started when we have read three rows of the image. For some extra margin, I let the ring buffer contain six rows.

The output of a given pixel is generated as soon as the unknown values of that pixel are estimated.

The version 2 design is a less intuitive, something that can be seen in figure 4.8. This flowchart is not complete, as it is missing the description of how to find all the red and blue pixel values.

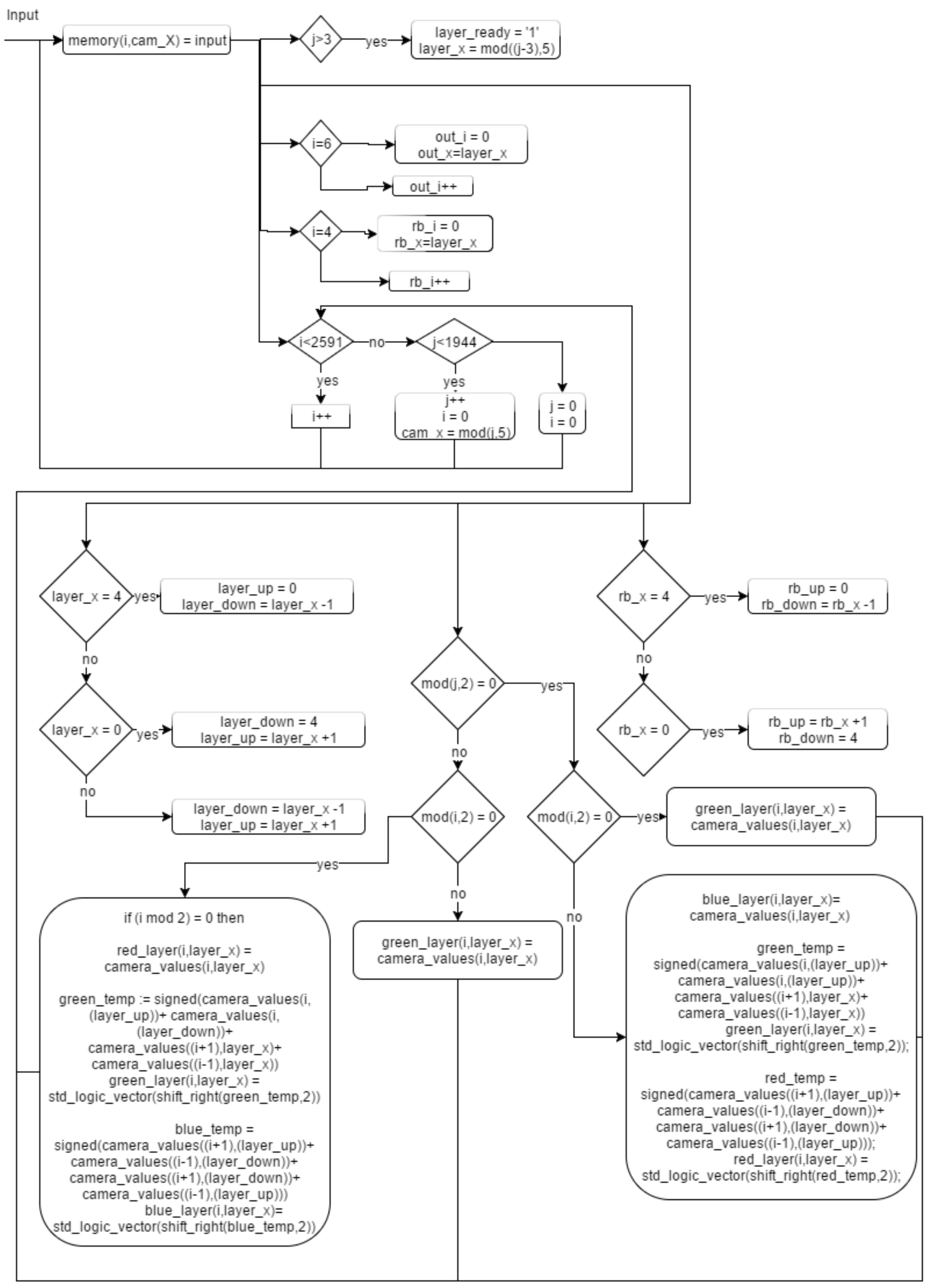


Figure 4.8: Flowchart of the demosaicing module, version 2

4.3.1 Processes

This version of the demosaicing module does not use a state machine. Instead it uses several different processes operating in parallel. These processes have different tasks.

counter

This process contains the different counters that are needed to operate the other processes. All of these counters are clocked, and are synchronous with the positive edge of the system clock. All of the counters are also reset to the default value, 0, when the system is reset.

The counter that determines what row in the ring buffer to be written to or read from, uses a modulo calculation of the j variable, which indicates what row we are currently reading from the image sensor. In this case, it is a modulo of six, since we have six rows in the ring buffer.

read_in

This process, as the name implies, reads in the data coming from the camera. The data is then stored in the ringbuffer, with two counters determining the position for the storage.

read_out

Writes the red, green and blue pixel values out so other modules may use them further down the compression chain. As with the read_in process it uses counters to determine position, but instead of determining where to write, the counters determine what position to read from.

layering

This is the process that does the estimating of the unknown pixel values based on the neighbouring known values. It uses several of the counters that are driven in the “counter” process. These counters ensure that estimations depending on the results of other estimations are delayed enough, so that it may estimate on the correct basis.

4.4 Finding minimum and maximum values

I created a module whose task was to extract the minimum and maximum value of each colour in the image.

This module will be situated right after the the demosaicing module, and read the three output channels and from them determine what the maximum and minimum value of each colour is.

The values are found by first setting the variables for the maximum values to all zeroes, and the variables for the minimum to all ones. Then we compare the read in values to the current variable for the respective colour. If the colour value from the demosaicing module is larger than the variable for the maximum value, it becomes the new maximum value. Ditto for the minimum value, except the other way round.

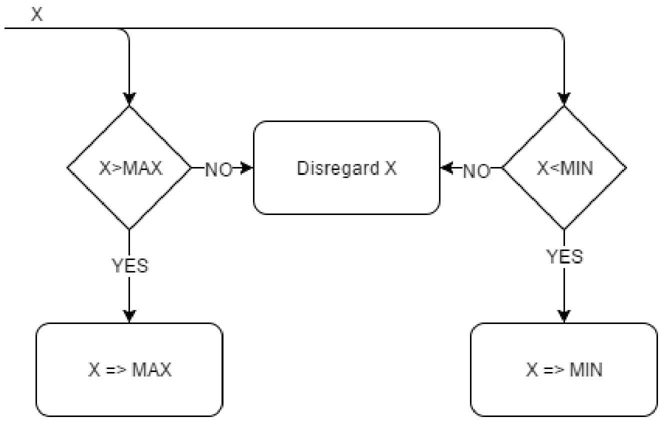


Figure 4.9: Flowchart for the min-max module

4.5 Histogram

The histogram module was made to produce histograms of the image data. The values of the pixels are evaluated, and the correct part of the histogram is increased by 1.

The incoming data from the image sensor is 12 bits before it gets extended in the “Unsigned extender”. This means that the MSBs of the vectors that represent the pixel value will always be zero. The easiest way to divide up the histogram is to divide it into a number of bins that is a power of two. This is because, as described in 2.3.2, it is very easy to divide by these numbers.

I made the histogram module such that it produces histograms with 8 bins. And since the MSBs are always zero because of the unsigned extension, I read bits 11 to 9. These three bits will represent a number between 0 and 7, which will decide what bin is increased.

4.6 Colour transform

After demosaicing, the next step towards a fully compressed image is a colour transform, and now with all three colours represented in each pixel, we can just do the needed arithmetic and create the transformed format.

By taking the generated three layers from the demosaicing process, the only thing that was needed was to use the correct equations to get the correct transforms, ready for further compression work.

4.7 Gamma correction

The normal gamma correction coefficient is 2.2 [19], but calculating the gamma value with Y = 1/2.2 is not easy in hardware. I used Y = 1/2 instead, that is, taking the square root of the input. Hardware solutions for square root of integers were available. I chose one that requires only one clock cycle, and produces an integer result.

CHAPTER 5
TESTING

5.1 Unsigned extender

This module was tested by making a test bench and analyzing the resulting waveform. In figure 5.1 we can see the waveform generated by the test bench for the unsigned extender. If we analyze this waveform we can see incoming bits are being grouped together, and they form a bit vector that is outputted.

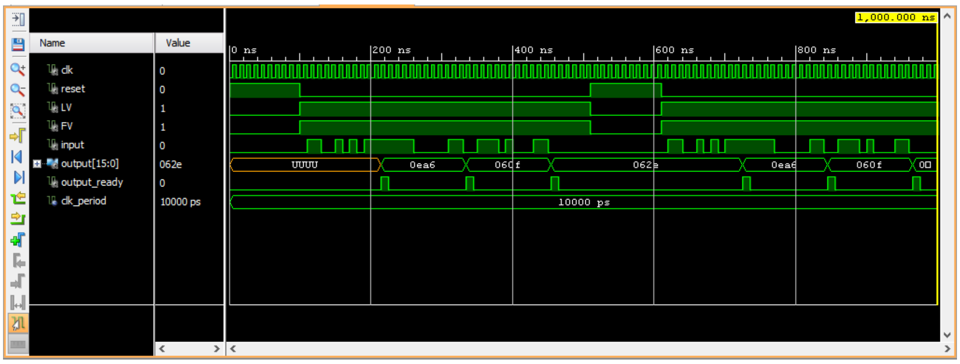


Figure 5.1: Waveform for the unsigned extender

5.2 Test image

To test the demosaicing module, I generated several test images using MATLAB. These images were made by first making them into a three layer image, where each layer represented red, green or blue. Then the image was made into an array of numbers, in the same style and format as the values from the camera.

The first test image to be made, was a total black image, this was made to verify that it would give the correct number of pixels back, and it is easy to check if all the pixel values are zero, as they should be for a black picture.

The next image was a completely white image, this was done to see that all the colours were calculated at the same time, and all the pixel values should be at the maximum value.

Then single-colour images were made, this was to see if the output image would be the same colour, and therefore no colour estimations were made on the wrong premises.

Finally, a more complex image with shapes and bordering colours of different hues were made. It also has colours with smooth gradients to see how these are preserved at the output. The image has colours with hard borders between them to observe how the transition looks at the output. This image can be seen in figure 5.2

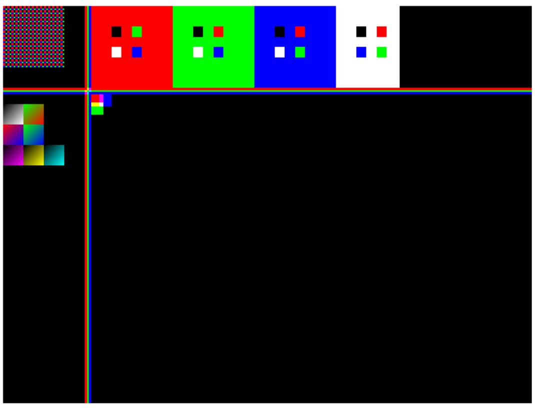


Figure 5.2: The matlab generated input image used to simulate demosaicing

5.2.1 Test image results

To the human eye, the reassembled output images are clearly the same as the input images. In the case of the single coloured images, we got images of only the same colour back, as expected.

The more complex test image returns what at first glance looks identical to the input, but when zooming in at the transition areas between two colours, one can see that the transition is not perfect. This is to be expected, however, because of the way the bi-linear demosaic process works.

As we can see from figure 5.3, the demosaic process works, and the image reassembled from the three output layers is clearly the same as the input image of figure 5.2. In figure 5.4, 5.5 and 5.6 we can see monochrome representations of the red, green and blue layers respectively. The degree of white is how close it is to its maximum level. These images are as expected, and shows that the output correlates to the input, and the outputted image is visually the same as the input image.

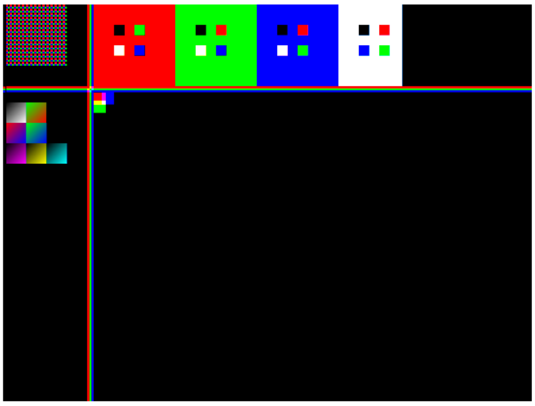


Figure 5.3: The image reassembled from three output layers from the demosaicing simulation

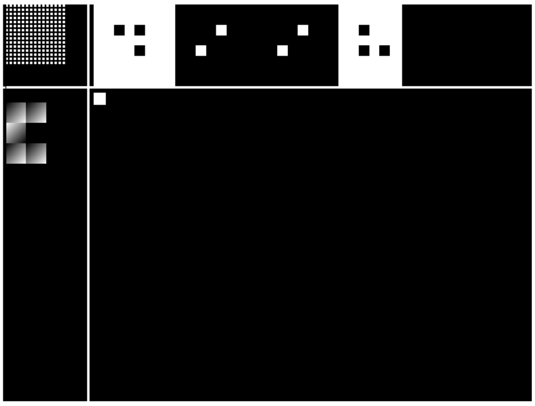


Figure 5.4: The red output layer from the demosaicing simulation

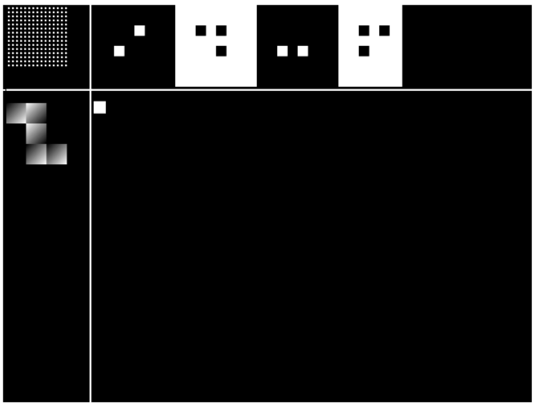


Figure 5.5: The green output layer from the demosaicing simulation

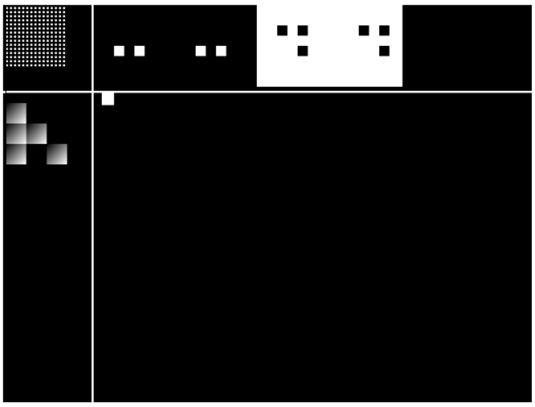


Figure 5.6: The blue output layer from the demosaicing simulation

Figure 5.7 shows the transitional properties between the blue and black areas. We can see that the intersection is not a sharp line. This is expected after the demosaic process because of the way it estimates the values.

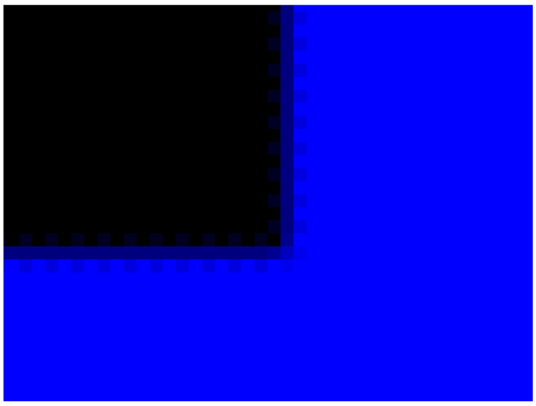


Figure 5.7: Zoomed in image of blue and black transitional area

Figure 5.8 shows red and blue, and we can see the transitional area has the same pattern as in figure 5.7. This can also be seen in figures 5.9, 5.10 and 5.11. We can see how the transitional area mixes the colours from the two areas, this is to be expected because of the averaging process in the demosaicing process.

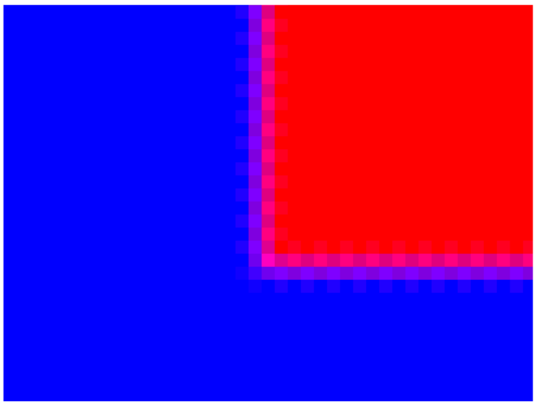


Figure 5.8: Transition from blue to red

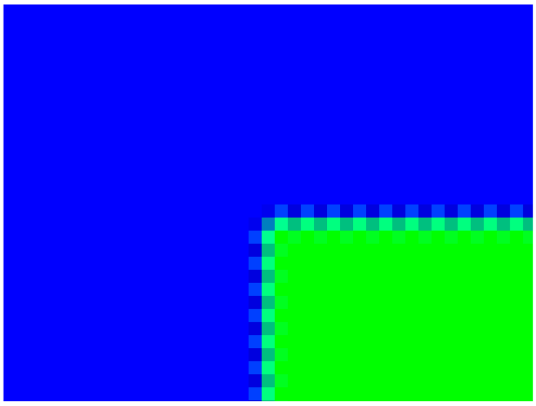


Figure 5.9: Transition from blue to green

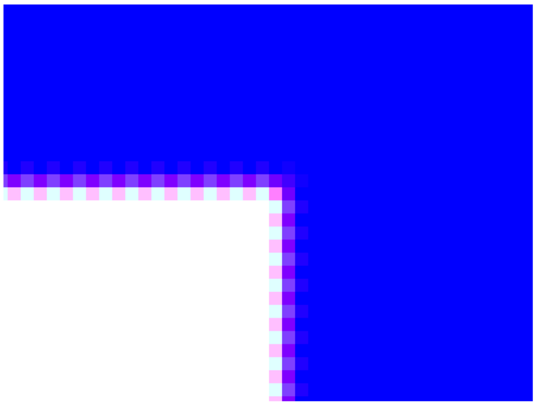


Figure 5.10: Transition from blue to white

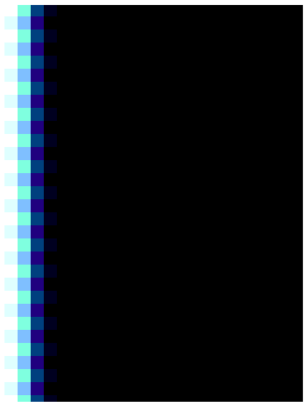


Figure 5.11: Transition from white to black

Figure 5.12 shows how a zoomed in portion where the colour change from green to blue is very gradual. We can see that this change is nice and smooth, indicating that the demosaicing process has worked as intended. Since the estimated pixel values are estimated from the average of the neighbouring pixels, it handles the gradually changing colours very well.

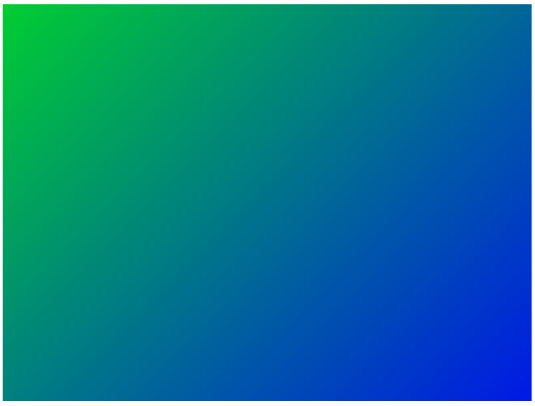


Figure 5.12: Gradual change from green to blue

Anomalies

When I was running the tests, the outputted text file from the simulation would always print out 14 very large negative numbers at the beginning. I was not able to trace the source of this anomaly, and the images generated from these textfiles would then be skewed by 14 bits. This is not a very significant fault if it is only an error in the test bench because the output generation is wrong, but is more serious if this correctly reflects the output.

When one looks at the waveform from the simulation, one cannot see the source of the error. This indicates that this is only an error in the writing process in the test bench, and not in the actual output from the module.

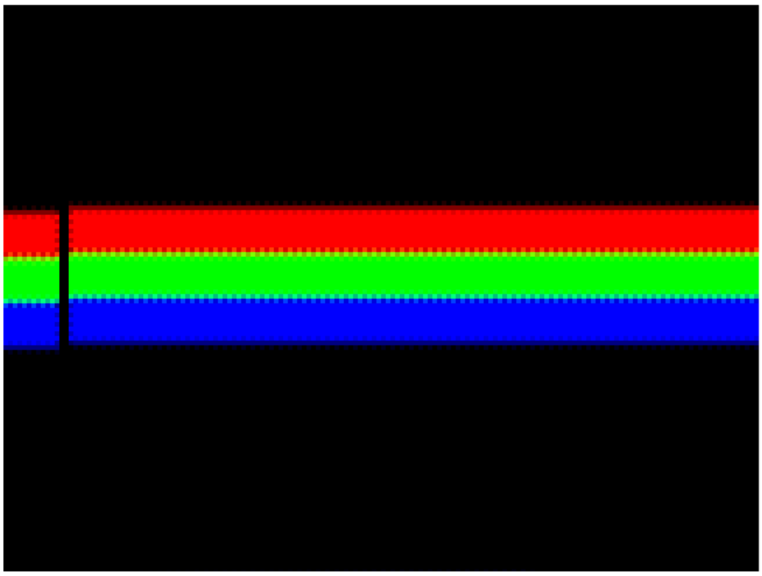


Figure 5.13: The anomaly creates a break in the line from starting the image 14 pixels later than it should.

5.2.2 Test image on demosaicing version 2

The same test images were run through the second version of the demosaicing module in the same manner, with first a black image, then a white, followed by images in only red, green or blue. These images gave the expected results.

Then, as was done for testing version 1, I sent the more complex test image through. At first glance it looked good, but on closer inspection there were faults in the red and blue layer. The green layer, however, looked correct and to expectations. In the red and blue layer, there were faint lines across the image that were not expected, and they should not be there. The problem of the 14 large negative numbers persisted in this version, but this can be explained by the fact that they are using the same test bench.

Figure 5.14 shows the output of the simulation from version 2, and we can see the way strange lines have emerged across the image which are clearly not meant to be there. If we look at the single colour output (figures 5.15, 5.16 and 5.17) we can see that the green layer is correct, but the red and blue layer also have the mysterious streaks. This points to the fact that the error has to be in the estimation of the blue and red layers.

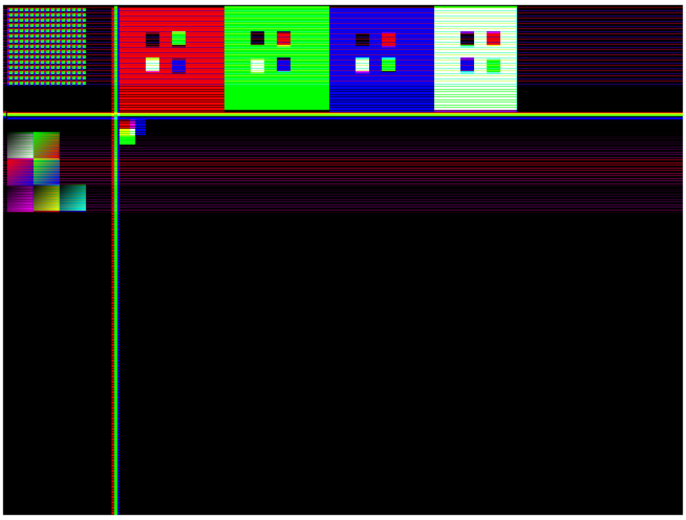


Figure 5.14: Output from demosaicing version 2

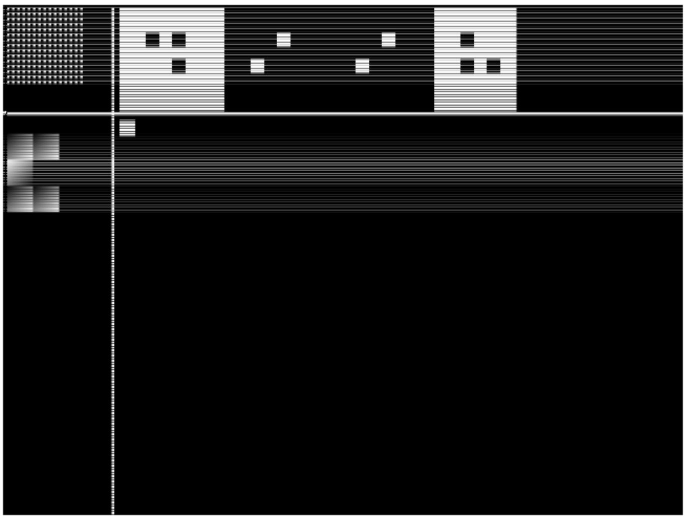


Figure 5.15: Red layer from demosaicing version 2

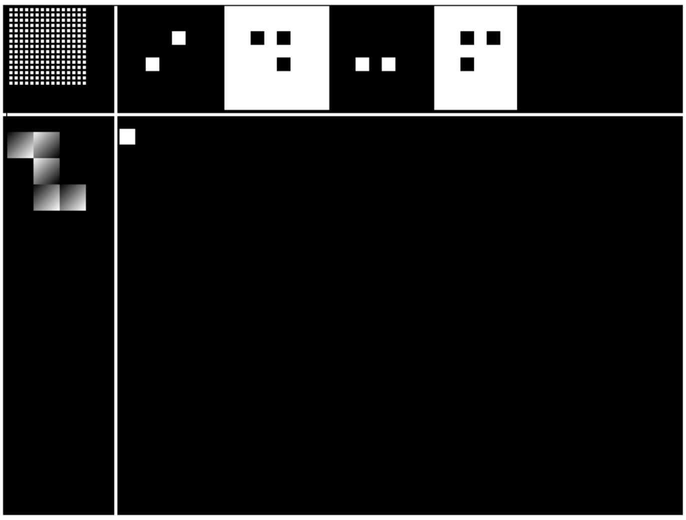


Figure 5.16: Green layer from demosaicing version 2

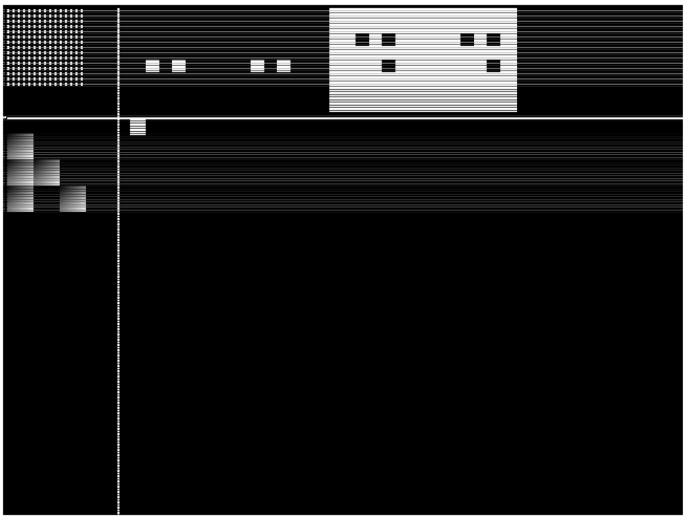


Figure 5.17: Blue layer from demosaicing version 2

If we look more closely at the outputted image, we can see from figure 5.18 that the skew also exists here, and as explained earlier, the same unexplained numbers appear in the output as they appeared in version 1.

In figure 5.19 we can see that the transition between the colours is not as smooth as in version 1, and that we have the streaking going across the image. In figure 5.20 it might seem like the streaking originates in the the colour blocks. We can also see, in both these images, that the red appears to be striped in stronger and weaker stripes of red. This is the blue streaks coming over the red and obscuring the image.

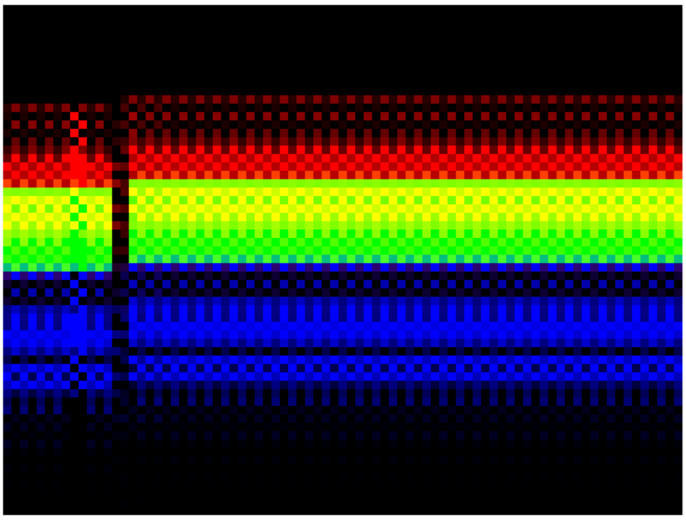


Figure 5.18: Anomaly from version 2

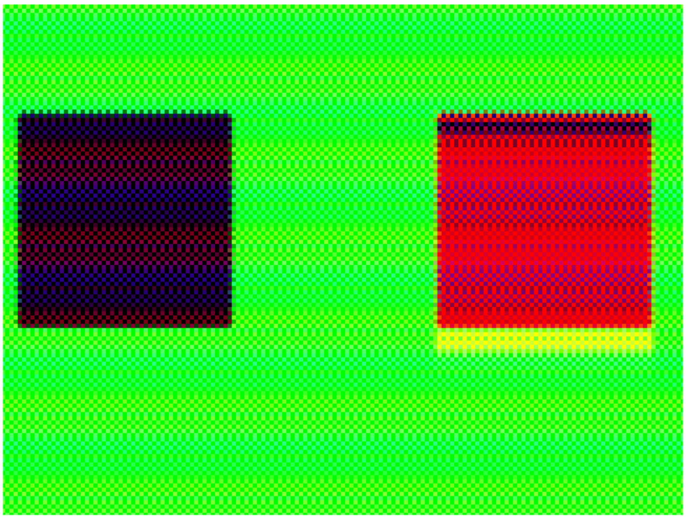


Figure 5.19: Anomaly from version 2

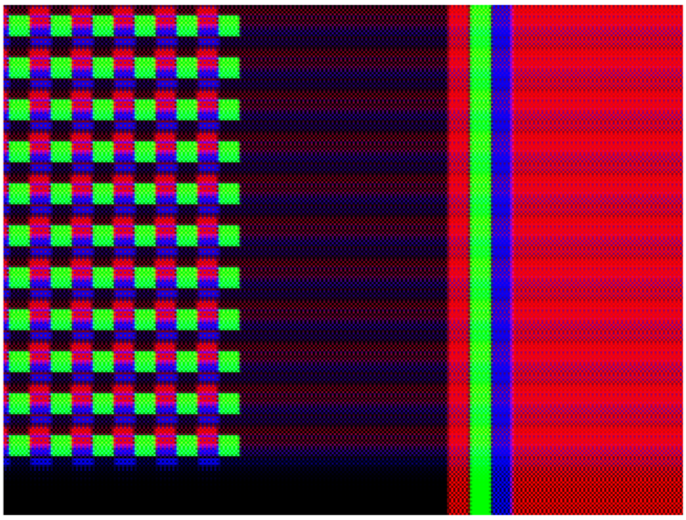


Figure 5.20: Anomaly from version 2

5.3 minmax module

The module that finds the largest and smallest value for the three output colours from the demosaicing module was tested by using a test bench where only a few input samples were simulated. This produced a waveform which was analyzed. Figure 5.21 shows that the module correctly changes its maximum and minimum values for red, green and blue, when new values of larger/smaller size are detected.

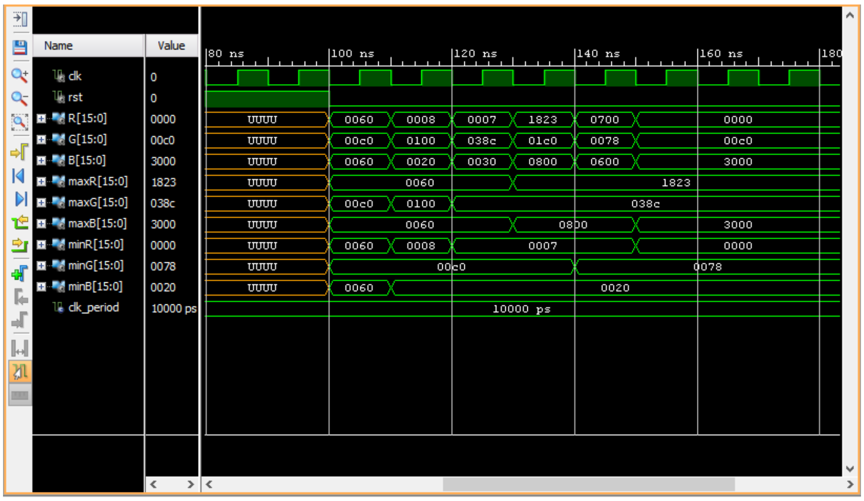


Figure 5.21: Waveform generated by the test bench belonging to the minmax module

5.4 Histogram

The histogram module was tested by looking at the waveform generated by the test bench to the module. Figure 5.22 shows part of the waveform that was generated in the test bench. It shows that the counters representing the different bins in the histogram are increased, and they are increased at the correct times.

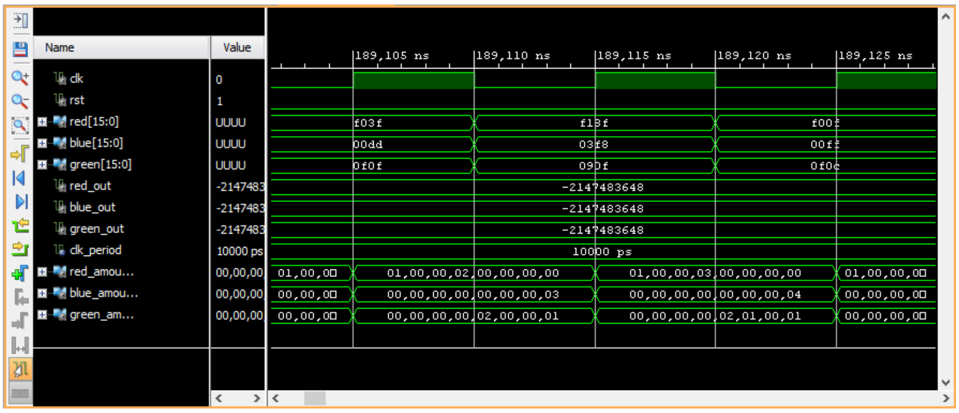


Figure 5.22: Waveform for the histogram module

5.5 Gamma correction

The testing of the gamma correction module was done in a very similar way as the testing of the minmax module. I made a test bench and ran several different values as input, and studied the waveform to see how the module behaved, and if that behaviour was correct.

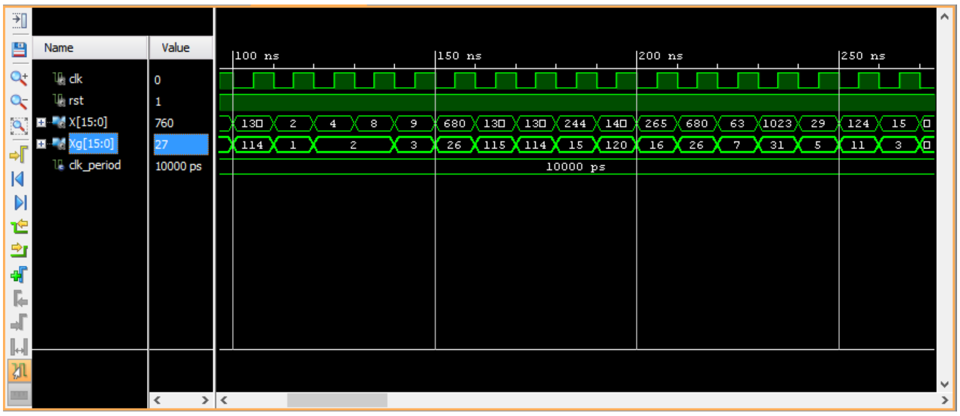


Figure 5.23: Waveform generated by the test bench for the the gamma correction module

We can see from figure 5.23 that the output is the square root of the input, but is rounded down to the closest integer. This is what was expected, and are good results.

CHAPTER 6
DISCUSSION

6.1 Star tracking

With the camera pointed towards space instead of earth, it may be possible to perform what is called star tracking. This is to determine one's position by using the known position of stars in relation to earth. This is a possible extra use of the camera, and can help the telemetry system of the satellite. The problem is that we do not know how well the camera manages to see faint light sources in an dark environment (stars in space).

finn
kilde

6.2 Scalability

The current system is made to work with the camera at full resolution. The image sensor has the ability to take pictures in different resolutions [18, page 16]. If there should be a wish to change the resolution, this system would not allow for that. To make a more scalable system would mean to make it more complex, and as explained earlier in this thesis, that is not good.

On the other hand, a scalable system would have the opportunity to take smaller pictures, which could be transmitted to earth faster. They could even be transmitted in a raw format over a short period of time.

6.3 Bi-linear demosaicing

There are many solutions to how to perform the demosaicing process. All of them have their strengths and weaknesses. The more complex the method, the less errors in the result, but there is no 100 % error free way of doing it. This is because we are only estimating the unknown values of the different colours, and we have no way knowing what the correct values really are.

The simplest way of doing it would be to just copy the value of a neighboring pixel of the same colour to the current pixel being estimated. This is also the method with the biggest errors [10].

The more accurate the estimations become, the more complex the structure needed to do the estimation becomes. As stated earlier, high complexity is not good, because of the difficulty of knowing how it will react in all situations. Because of this, one needs to find a compromise between complexity and accuracy. I believe bi-linear demosaicing is the best solution for our purpose, as it provides us with good estimations of the unknown colour values, and at the same time is relatively easy to implement in VHDL.

6.4 Demosaicing version 1 versus version 2

The two versions of the demosaicing module solve the same task, and are both using the same basic algorithm to estimate the missing values. The difference lies in the parallelism of the two systems, where version 2 manages to do the different tasks in parallel, while version 1 performs them consecutively.

The other big difference is in the use of memory. Version 1 uses a lot of memory, as it stores all the data that comes from the camera before it continues its processing, and it also stores all the data from the three colours. Version 2 uses much less memory because of its use of a ring buffer, and because it neither stores all the data from the camera nor the data for the three colours that are outputted.

The problem with version 2, though, is that simulations have produced some anomalies that we do not see in version 1, and hence will require further work before it can be seen as a successful demosaicing module. So my recommendation for the current versions is to use version 1 until version 2 has been fixed and tested properly.

6.5 Storing of raw data

By using ring buffers we do not need to store much of the raw data from the camera. This decreases the need for storage space and RAM drastically, but at the same time, we lose the raw data during the process.

The loss of raw data means that there is no way of downloading this data to earth. However, this is an unlikely scenario as the download would take a long time due to limited bandwidth and the large size of the raw data.

6.6 Reprogrammability

The FPGA will be powered down when it is not needed, and each time it is turned on it will read the bit file that holds the information of its functionality. This means that if other parts of the satellite have access to the memory where it reads the bit file from, it will be possible to reprogram the FPGA in flight.

This introduces both advantages and problems. The advantage is the possibility to reprogram the FPGA, and that we can make sure the bit file is correct, and change it, if a bit flip were to happen to it. The problem is the increased complexity of the system, and the possibility of the bit file to be overwritten by a fault, and thereby be erroneous.

6.7 Black and white images

After the colour transform, we have three new layers: Y, Cb and Cr. The Y layer represents the luminescence, and when looking at this layer it is evident that it will be an approximation of a black and white version of the image. This may be used to generate black and white images, which can be compressed to generate even smaller files than a colour image would.

6.8 Choice of FPGA

The current chosen FPGA is not rad-hardened. This can create problems, as space is a high radiation environment. There exist FPGAs that are made specifically for space operations. We have for example virtex 4 and virtex 5 from Xilinx.

6.9 Future work

The tested modules are not tested in cohesion with each other and this needs to be done before the final implementation. However, the most important task is to make a module for the compression of the image, using the pre-processed data from the modules created.

The modules are also not implemented on a physical FPGA, they are only simulated using test benches. So there needs to be done synthesis and "place and route", and this may uncover errors that were not discovered by simulation.

Further, there should be taken a closer look at whether the current FPGA is good enough, and if one should change it to a rad-hardened device.

The code is not synthesized or placed and routed. This needs to be done. This may uncover unknown faults in the design.

CHAPTER 7
CONCLUSION

The simulations show that the modules are working individually, but they are not tested with each other. The module for compressing the captured images is also missing, but [9] and [20] claim that doing DWT in an FPGA is no problem.

The code is not synthesized or placed and routed, and this may reveal problems that simulations have not discovered. The lack of synthesis also means we do not know if these modules can handle the desired clock frequency. Also, without the synthesis, one does not know how large the physical circuit will be, and if the currently chosen FPGA will be large enough.

Because of problems with the prototype, there has not been done any testing on a physical circuit, and therefore we do not know if this new solution is better regarding speed and power consumption.

In the future there should be made an effort to tie the current modules together and synthesize them. Further, they should be tested on the prototype and see how well this performs compared to the already existing solution made by Andreas Bertheussen and Thomas Nornes.

REFERENCES

[1] Xilinx, "Field programmable gate array (fpga)." http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm, 2016.

[2] Xilinx, *MBv7 FAQ*. Xilinx, http://www.xilinx.com/ipcenter/processor_central/microblaze/doc/mb_faq.pdf, jul 2008. Datasheet.

[3] D. S. Taubman and M. W. Marcellin, *JPEG2000: Image Compression Fundamentals, Standards and Practice (The International Series in Engineering and Computer Science)*. ISBN 0-7923-7519-X, 2002.

[4] LionDoc, "Own work, public domain." https://commons.wikimedia.org/w/index.php?curid=19224869. figure.

[5] schorsch, "Luminance." http://www.schorsch.com/en/kbase/glossary/luminance.html, 2013.

[6] S. A. Eugster, "Own work, public domain." https://commons.wikimedia.org/w/index.php?curid=10972475. figure.

[7] "Overview of jpeg 2000." https://jpeg.org/jpeg2000/. online.

[8] N. O. Bakkeb, E. L. Flogard, M. Gammelster, H. S. Mork, A. F. Pignde, and S. Solberg, "Bildekomprimering," project report for eit, NTNU, https://www.ntnu.no/wiki/download/attachments/63574301/Bildekomprimering.pdf?version=1&modificationDate=1423675763000&api=v2, may 2014.

[9] M.Puttaraju and A.R.Aswatha, "Fpga implementation of 5/3 integer dwt for image compression," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 10, pp. 187 – 191, 2012.

[10] H. S. Malvar, L. wei He, and R. Cutler, "High-quality linear interpolation for demosaicing of bayer-patterned color images." http://research.microsoft.com/pubs/102068/Demosaicing_ICASSP04.pdf, 2016.

[11] D. Khashabi, S. Nowozin, J. Jancsary, A. W. Fitzgibbon, and B. Lindbloom, "Pattern-independent demosaicing." http://i.i.cbsi.com/cnwk.1d/i/tim/2012/02/06/Adobe-raw-demosaic-diagram.jpg, 2016.

[12] J. S. C. Space Radiation Analysis Group, "What is space radiation?." http://srag-nt.jsc.nasa.gov/spaceradiation/what/what.cfm, may 2014.

[13] D. White, "Considerations surrounding single event effects in fpgas, asics, and processors." http://www.xilinx.com/support/documentation/white_papers/wp402_SEE_Considerations.pdf, may 2012.

[14] A. Bertheussen, "Digital processing system for a cubesat camera," project report, NTNU, https://www.ntnu.no/wiki/download/attachments/63574301/digital_processing_for_cubesat_camera_andreas_bertheussen.pdf?api=v2, 2014.

[15] T. H. Nornes, “Prototype design for cubesat camera,” project report, NTNU, https://www.ntnu.no/wiki/download/attachments/63574301/prototype_design_for_cubesat_by_thomas_hanssen_nornes.pdf?version=1\&modificationDate=1420909519000\\&api=v2, 2014.

[16] Xilinx, “Vivado high-level synthesis.” http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. online.

[17] M. Adams, “The jasper project home page.” https://www.ece.uvic.ca/~frodo/jasper/. Location of the jasper codec.

[18] O. semiconductor, “1/2.5-inch 5 mp cmos digital image sensor.” http://www.onsemi.com/pub_link/Collateral/MT9P031-D.PDF, 2015. Datasheet.

[19] C. in Colour, “Understanding gamma correction.” http://www.cambridgeincolour.com/tutorials/gamma-correction.htm. Tutorial.

[20] T.Vijayakumar and S.Ramachandran, “Fpga implementation of 2d-dwt and spiht architecture for lossless medical image compression,” *International Journal of Scientific and Engineering Research*, vol. 4, August 2013.

APPENDIX A
CODE REPOSITORY

All relevant code mentioned in this thesis can be found in `https://peraro90@bitbucket.org/peraro90/satellite.git`. In this repository one will find several files and folders. There are .m files, which are MATLAB scripts, and the folders are projects done in Vivado.

The folders demosaic, demosaic_2, demosaic_3 contain different versions of the demosaicing version 1, where demosaic_3 is the completed one. Demosaic_4 contains the demosaicing version 2 module.

The gamma folder contains the gamma adjustment module. Histogram contains the histogram module, minmax_2 contains the minmax module, and unsigned_extender contains the unsigned extender module.

APPENDIX B
IMAGE GENERATION AND REGENERATION

The images were generated in MATLAB using MATLAB scripts. Here are the scripts for making the test image, and how to read the outputted data from the simulation and make an image from that data.

B.1 Producing original image

```
i=1;
j=1;
image = zeros(1944,2592,3);
%red squares
for i = 1:2:30
    for j = 1:2:30
        image((10*(i-1)+1):((10*(i-1))+10),(10*(j-1)+1:((10*(j-1))+10)
            ),1)=1;
    end
end
%green squares
for i = 2:2:30
    for j = 2:2:30
        image(((10*(i-1))+1):((10*(i-1))+10),((10*(j-1))+1:((10*(j-1))
            +10)),2)=1;
    end
end
%blue squares
for i = 2:2:30
    for j = 1:2:30
        image((10*(i-1)+1):((10*(i-1))+10),(10*(j-1)+1:((10*(j-1))+10)
            ),3)=1;
    end
end

%coloured ribbons that cross
image(401:410,:,1)=1;
image(411:420,:,2)=1;
image(421:430,:,3)=1;

image(:,401:410,1)=1;
image(:,411:420,2)=1;
image(:,421:430,3)=1;

%intersecting squares
image(431:490,431:490,1)=1;
image(471:530,431:490,2)=1;
image(431:490,471:530,3)=1;
```

```

%graded square black to white
for i = 1:100
    for j = 1:100
    image(i+480,j,1)=((i+j)/2)/100;
    image(i+480,j,2)=((i+j)/2)/100;
    image(i+480,j,3)=((i+j)/2)/100;
    end
end

%graded squares of different colours
for i = 1:100
    for j = 1:100
    image(i+480,j+100,1)=((i+j)/2)/100;
    image(i+480,j+100,2)=(100-((i+j)/2))/100;
    end
end

for i = 1:100
    for j = 1:100
    image(i+580,j+100,3)=((i+j)/2)/100;
    image(i+580,j+100,2)=(100-((i+j)/2))/100;
    end
end

for i = 1:100
    for j = 1:100
    image(i+580,j,3)=((i+j)/2)/100;
    image(i+580,j,1)=(100-((i+j)/2))/100;
    end
end

for i = 1:100
    for j = 1:100
    image(i+680,j,3)=((i+j)/2)/100;
    image(i+680,j,1)=(((i+j)/2))/100;
    end
end

for i = 1:100
    for j = 1:100
    image(i+680,j+100,2)=((i+j)/2)/100;
    image(i+680,j+100,1)=(((i+j)/2))/100;
    end
end

for i = 1:100
    for j = 1:100
    image(i+680,j+200,2)=((i+j)/2)/100;
    image(i+680,j+200,3)=(((i+j)/2))/100;
    end
end

%red block with contrasts
image(1:399,431:830,1)=1;
image(100:150,531:581,1)=0;
image(200:250,531:581,1:3)=1;
image(100:150,631:681,2)=1;
image(100:150,631:681,1)=0;
image(200:250,631:681,3)=1;
image(200:250,631:681,1)=0;

%green block with contrasts
image(1:399,831:1230,2)=1;
image(100:150,931:981,2)=0;
image(200:250,931:981,1:3)=1;
image(100:150,1031:1081,1)=1;
image(100:150,1031:1081,2)=0;
image(200:250,1031:1081,3)=1;
image(200:250,1031:1081,2)=0;

%blue block with contrasts
image(1:399,1231:1630,3)=1;
image(100:150,1331:1381,3)=0;
image(200:250,1331:1381,1:3)=1;
image(100:150,1431:1481,1)=1;
image(100:150,1431:1481,3)=0;
image(200:250,1431:1481,2)=1;
image(200:250,1431:1481,3)=0;


%white block with contrasts
image(1:399,1631:1944,:)=1;
image(100:150,1731:1781,:)=0;
image(200:250,1731:1781,1:2)=0;
image(100:150,1831:1881,2:3)=0;
image(100:150,1831:1881,3)=0;
image(100:150,1831:1881,2)=0;
image(200:250,1831:1881,1:2:3)=0;

%show image
figure, imshow(image)

for i = 1:1944
    for j= 1:2592
        if mod(j,2) == 0
            if mod(i,2) == 0
                image_out(i,j) = image(i,j,2)*4095;
            else
                image_out(i,j) = image(i,j,3)*4095;
            end
        else
            if mod(i,2) == 0
                image_out(i,j) = image(i,j,1)*4095;
            else
                image_out(i,j) = image(i,j,2)*4095;
            end
        end
    end
end

fid=fopen('bilde.txt', 'wt');
for i = 1:1944
    for j= 1:2592
```

```
        fprintf(fid, '%d\n', image_out(i,j));
    end
end
fclose(fid);
```

Listing B.1: Producing original image

B.2 Reproduce after simulation

```
fileID1 = fopen('bilde_out_red.txt','r');
fileID2 = fopen('bilde_out_blue.txt','r');
fileID3 = fopen('bilde_out_green.txt','r');
formatSpecs = '%f';
A = fscanf(fileID1,formatSpecs);
B = fscanf(fileID2,formatSpecs);
C = fscanf(fileID3,formatSpecs);
n = 1;
picture_red(1:1944,1:2592) = 0;
picture_blue(1:1944,1:2592) = 0;
picture_green(1:1944,1:2592) = 0;
image = zeros(1944,2592,3);

for i = 1:1943
    for j= 1:2592
        picture_red(i,j) = A(n)/4095;
        image(i,j,1) = A(n)/4095;
        picture_blue(i,j) = B(n)/4095;
        image(i,j,3) = B(n)/4095;
        picture_green(i,j) = C(n)/4095;
        image(i,j,2) = C(n)/4095;
        n = n+1;
    end
end

figure, imshow(image)
figure, imshow(picture_red)
figure, imshow(picture_green)
figure, imshow(picture_blue)
```

Listing B.2: Reproduce after simulation