NTNU

Norwegian University of
Science and Technology

FPGA Implementation of Hyperspectral Anomaly Detection Algorithm

Martin Haukali

Master of Science in Electronics
Submission date: June 2018
Supervisor: Kjetil Svarstad, IES
Co-supervisor: Milica Orlandic, IES

Norwegian University of Science and Technology
Department of Electronic Systems

Project Assignment

Candidate name: Martin Haukali

Assignment title: FPGA Implementation of Hyperspectral Anomaly Detection Algorithm

Assignment text: This topic is part of the large project Hyperspectral Imaging in Small Satellites. Hyperspectral imaging relies on sophisticated acquisition and on data processing of hundreds or thousands of image bands. Most of the algorithms for hyperspectral imaging perform intensive matrix manipulations, and FPGAs are recommended to be used due to reconfiguration, low consumption, compact size and high computing power.

Anomaly detection is an important task for hyperspectral data exploitation. A standard approach for anomaly detection in the literature is the method called RX algorithm. The computational cost is very high for RX algorithm and current advances in high performance computing can be good solution to reduce the run- time of this algorithm.

Tasks:
- Optimization of RX algorithm for parallel processing
- Covariance computation
- Hardware implementation of inverse matrix problem
- FPGA implementation of RX algorithm

Supervisor: Kjetil Svarstad

Co-supervisor: Milica Orlandic

Abstract

On-board processing of hyperspectral data in satellites is done to perform a wide variety of tasks. Field-Programmable Gate Arrays (FPGAs) are often used for such tasks due to their reconfigurability and efficiency, especially when dealing with applications requiring matrix computation. One of these applications is anomaly detection. Anomaly detection might be used to discover harmful algae blooms, oil spills, micro-plastics and other irregularities in ocean and coastal areas. This might help us understand more about the ocean and to monitor the effects of global warming and human pollution.

The Adaptive Causal anomaly detector (ACAD) is an anomaly detector (AD) developed to solve some of the issues that well-known ADs, such as the Reed-Xiaoli (RX) algorithm, faces. ACAD utilizes inverse matrix computation as a part of the anomaly detection. Computing the inverse matrix is an intensive task. It is therefore important that the algorithm chosen for inverse matrix computation is parallellizable and efficient. The Gauss-Jordan elimination was chosen due to its parallel computation and simplicity.

ACAD is causal, meaning that it relies on previously executed computations. This enables real-time processing and makes it suitable for hardware implementation. ACAD also builds a binary anomaly map, which is beneficial with regards to data transmission, as this will lower transmission time and thereby energy. In this thesis, a proposed implementation of the ACAD algorithm has been made, designed to be scalable for large hyperspectral images. A parallel memory structure consisting of Block RAM (BRAM)-arrays of size P_bands have been made. P_bands is the number of spectral bands of the input pixel data to the ACAD AD. The correlation and inverse modules proposed in this implementation have a large degree of parallelism, computing and updating up to two rows of the correlation and inverse matrix respectively, of size $P\_bands \times P\_bands$, per clock cycle. The design is to be implemented on a Zynq-7000 series System-on-Chip (SoC).

Sammendrag

Prosessering av hyperspektrale bilder på den innebygde datamaskinen i en satelitt blir ofte gjort for å utføre en mengde oppgaver. Field-Programmable Gate Arrays (FPGAs) blir ofte brukt for slike oppgaver på grunn av den høye effektiviteten og muligheten for rekonfigurasjon. Dette gjelder spesielt for applikasjoner som krever matrise-beregning. En slik applikasjon er anomalie-deteksjon. Anomalie-deteksjon kan muligens bli brukt for å oppdage oppblomstring av skadelige alger, oljesøl, mikro-plastikk og andre uregelmessigheter i havet og i kystnære områder. Dette kan hjelpe oss å forstå mer om havet og å observere effektene av global oppvarming og menneskelig forurensing.

Adaptive Causal anomaly detector (ACAD) er en anomalie-detektor (AD) utviklet for å løse noen av utfordringene som velkjente AD-er, slik som Reed-Xiaoli (RX)-algoritmen, har. ACAD bruker invers-matrise-beregning som en del av anomalie-deteksjonen. Å beregne invers av en matrise er en intensiv oppgave. Det er derfor viktig at algoritmen brukt for å beregne invers-matrisen er parallelliserbar og effektiv. I denne hovedoppgaven ble Gauss-Jordan metoden valgt grunnet dens parallelle beregninger og enkelhet.

ACAD er kausal, som betyr at den avhenger av tidligere utførte beregninger. Dette muliggjør sanntidsprosessering og gjør algoritmen egnet for implementasjon i maskinvare. ACAD lager også et binært anomalie-kart, som er fordellaktig med tanke på data-overføring, da det vil senke transmisjonstiden og dermed transmisjonsenergien. I denne hovedoppgaven har en foreslått implementasjon av ACAD blitt laget, designet for å være skalerbar for store hyperspektrale bilder. En parallell minne-struktur bestående av Block RAM (BRAM)-tabeller av størrelse P_bands har blitt designet. P_bands er antallet spektrable bånd i inngangs-piksel-dataen til ACAD AD-en. Korrelasjons- og invers-modulene foreslått i denne implementasjonen har en stor grad av parallellisme. Modulene beregner og oppdaterer opp til to rader av henholdsvis korrelasjons-og invers-matrisen, som er av størrelse $P\_bands \times P\_bands$, per klokke-sykel. Designet skal bli implementert på en Zynq-7000 serie System-on-Chip (SoC).

Preface

This thesis is submitted to the department of Electronics Systems at NTNU as part of the Master of Science degree in Electronics, with specialization in digital circuit design. The thesis is part of the SmallSat project at NTNU, a research project that is focused on the design and creation of small satellites. The mission objectives of the SmallSat project is to "provide and support ocean color mapping through a Hyperspectral Imager payload, autonomously processed data, and on-demand autonomous communications in a concert of robotic agents at the Norwegian coast". The thesis was started on in the mid of January. It was not a continuation of the project thesis written in the past fall.

The implementation of an anomaly detector in hardware proved to be a challenging task. Choosing the best anomaly detector for this project in an as objective manner as possible proved to be difficult, as this meant creating synthetic images to provide objective metrics for performance measurement and testing the anomaly detectors on real hyperspectral images. In the end, the Adaptive Causal anomaly detection algorithm was chosen. The main challenge faced with the implementation in hardware was how to use the given resources of the FPGA to compute the anomaly detection as efficient as possible. Anomaly detection is a computationally intensive process, especially when doing inverse computation. As such, the trade-off between resource utilization and throughput was a difficult one.

Thanks to my co-supervisor Milica Orlandic for great guidance and help during the semester. I would also like to thank my supervisor Kjetil Svarstad for a read-through of my thesis in the latter stages.

The code, both VHDL and MATLAB, developed and used in this thesis is made available public on github. The most important code can also be found in the appendices to this thesis.

The VHDL source code is located on the following website, on the "invert_matrix_computation"- branch: `https://github.com/marthauk/Anomaly-detection/tree/invert_matrix_computation/FPGA_implementation/Anomaly_detection/Anomaly_detection.srcs/sources_1/new`.
The VHDL-testbenches used are avaialable on the following website, on the "invert_matrix_computation"- branch:
`https://github.com/marthauk/Anomaly-detection/tree/invert_matrix_computation/FPGA_implementation/Anomaly_detection/Anomaly_detection.srcs/sim_1/new`.

The MATLAB-code used and developed for hyperspectral processing can be found on the following website, on the "dev" branch:

https://github.com/marthauk/HyperSpectralToolbox/tree/dev/functions

Contents

List of Figures

List of Tables

List of Abbreviations

- ACAD - Adaptive Causal anomaly detection
- AD - Anomaly detector
- ALRX - Adaptive Local RX
- AVIRIS - Airborne Visible Infrared Imaging Spectrometer
- BRAM - Block RAM
- CLB - Configurable Logic Block
- CORDIC - COrdinate Rotation DIgital Computer
- DMA - Direct Memory Access
- DSP - Digital Signal Processor
- FPGA - Field-Programmable Gate Array
- FSM - Finite State Machine
- GRX - Global RX
- HAB - Harmful algae bloom
- HSI - Hyperspectral imaging
- IP - Intellectual Property
- LRX - Local RX
- LUT - Look-up-table
- MSB - Most Significant Bit
- PCA - Principal Component Analysis
- RAM - Random Access Memory
- RX - Reed-Xiaoli
- SIPO - Serial-in Parallel-Out
- SNR - Signal-to-noise ratio
- SoC - System-on-Chip
- TDP - True dual port
- WNS - Worst Negative Slack
- WSN - Wireless sensor node

Also, it is worth mentioning that Zedboard Zynq Evaluation and Development kit will be referred to as Zedboard.

Chapter 1

Introduction

1.1 Motivation

This master thesis is part of the NTNU SmallSat [5] project. One of the projects mission objectives is to use hyperspectral imaging to observe and collect ocean color data, and to detect and characterize spatial extent of algal blooms. A small satellite will be launched in 2020 to be able to meet these objectives. The payload of the satellite will be a 1/3 U push-broom type hyperspectral imager, dedicated to take images of a $30 \times 50km^2$ area. In regular Red Green Blue (RGB) imaging each of the image pixels is made up of three frequency components that represent the intensities in red, green and blue frequencies respectively. Such a component is referred to as a band. In hyperspectral imaging (HSI), a pixel will typically consist of hundreds to thousands of bands, providing more information than regular images. This information can be used for a lot of different purposes. It can for example be used to detect different materials in an area, by using spectral signatures of materials as identifiers. Figure 1.1 shows the functional concept of HSI.

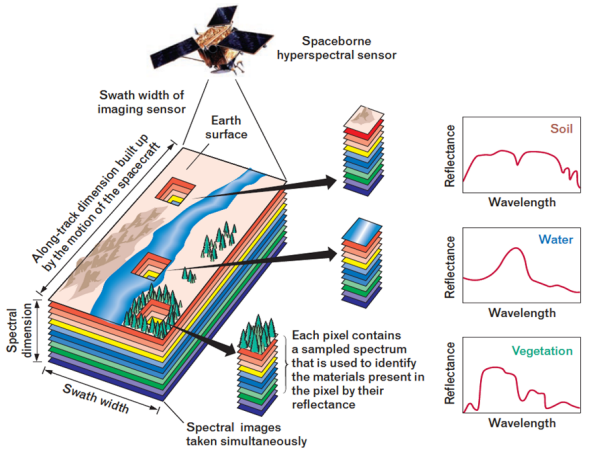

Figure 1.1: Functional concept of HSI [1].

NTNU SmallSat aims to use this information to detect algae blooms, phytoplankton, oil spills, microplastic and possibly other irregularities or *anomalies* in the ocean. An anomaly in the context of HSI is a spectral pixel vector that has significant spectral differences from its surrounding background pixels [13]. Detection of harmful algae blooms (HABs) is particularly interesting for the salmon farms located along the coast of Norway, as such blooms can be toxic, even deadly, for the salmon. Algae were most likely the cause of death for 38 000 salmons in southern Troms in September of 2017 [2]. An image of such a bloom can be seen in Figure 1.2. Increasing ocean temperatures as a consequence of global warming may lead to more frequent and intense HABs [14].

Figure 1.2: Image of an algae bloom along the coast of Troms in Norway [2].

Algae will have a spectral signature that are different to the background, which will be ocean water or land. Algae may therefore be considered anomalies.

Anomaly detection may help combat and monitor the challenges faced globally as a consequence of global warming and human pollution. One of these challenges is the vast amount of micro-plastic (plastic particles smaller than $5mm$) in the world's oceans. Anomaly detection may be used to detect spots of ocean water having higher density of micro-plastic than the surrounding ocean water, if such spots exist.

1.1.1 Main contributions

The main contributions in this thesis are:

- Making a fork of the MATLAB Hyperspectral toolbox. This is used to test high-level models of the considered anomaly detectors (ADs).
- Making models in MATLAB of the Local Reed-Xiaoli (LRX), the Adaptive Causal anomaly detector (ACAD) and the Adaptive Local Reed-Xiaoli (ALRX).
- Developing synthetic images for testing of ADs in MATLAB.
- Testing of ADs on real hyperspectral image data from the Cuprite mining scene.
- Doing an objective review of the considered ADs.
- Proposing hardware implementation of the chosen anomaly detection algorithm, the ACAD algorithm.
- Synthesis and simulation of the ACAD hardware implementation.

1.1.2 Problem statement

The assignment text states that optimization of the Reed-Xiaoli (RX) algorithm should be done. It also states that covariance computation is one of the tasks. As is described in Section 3, the RX algorithm is less suited for implementation in hardware than the ACAD algorithm. As such, this thesis describes an implementation of ACAD instead of RX. ACAD utilizes correlation computing instead of covariance computation. Therefore, this thesis describes implementation of correlation computation.

1.1.3 Master thesis overview

The following chapters in this report describe the implementation of an AD on a Xilinx Kintex-7 FPGA. The AD is made for the SmallSat project [5].

Background theory is presented in Chapter 2. Algorithms considered used for anomaly detection are described in this chapter. This include the RX algorithm, the LRX algorithm, the ACAD algorithm and a proposed AD algorithm by the author, called the ALRX algorithm. Inverse matrix computation is a part of all the considered ADs. Therefore, Chapter 2 also contains theory about inverse computation.

Chapter 3 contains a review of the considered ADs presented in chapter 2. The main object of this review is to provide means to decide which of the considered ADs is most suited for hardware implementation. In this section, tests of the different ADs have been done in MATLAB on synthetic images and on real image data from the Cuprite scene.

In Chapter 4, the proposed hardware implementation of the ACAD algorithm is described. The architecture is presented, along with different design considerations. The architecture is divided into five parts; **FSM ACAD**, **Shiftregister**, **ACAD inverse**, **ACAD correlation** and **dACAD**. The **FSM ACAD** acts as the control logic of the anomaly detector. **Shiftregister** handles input data from the Cube DMA used by the SmallSat project. Inverse computation is done in **ACAD inverse**. In **ACAD correlation**, the correlation matrix computation is done. The **dACAD** computes the final result of the anomaly detection, and decides if a pixel is anomalous. Implementation of

these blocks are described in this chapter.

Results are presented in Chapter 5. This section contain synthesis results and simulation results.

Chapter 6 is the Discussion section. The results presented in Chapter 5 are discussed, including timing and synthesis results.

Chapter 7 is the Conclusion section. The most important results are presented here. At last, the concluding remarks and recommendations for future work and optimization are given.

In appendix A, the most important MATLAB code used in this thesis is found. Appendix B contain a short description of the VHDL source code, which can be found in C.

Chapter 2

Background theory

2.1 Hyperspectral imaging

Hyperspectral imaging collects information from the electromagnetic spectrum. This information can be utilized for a wide range of application, including anomaly detection.

2.1.1 AVIRIS

The Airborne Visible Infrared Imaging Spectrometer (AVIRIS) is a hyperspectral imager launched by NASA. Its main objective is to "identify, measure and monitor constituents of the Earth's surface and atmosphere" [15]. AVIRIS has a spatial resolution of 224 spectral bands. In MATLAB-processing 163 of these bands are used.

2.1.1.1 Cuprite scene

Data from the Cuprite mining district [3] captured by the AVIRIS imager is often used as a benchmark scene for different image processing algorithms, including anomaly detection algorithms. Scene 02 from the Cuprite mining can be seen in Figure 2.1. Twelve different minerals with their respective spectral signature are extracted from the scene [4]. The spectral signatures of the different minerals can be seen in Figure 2.2.

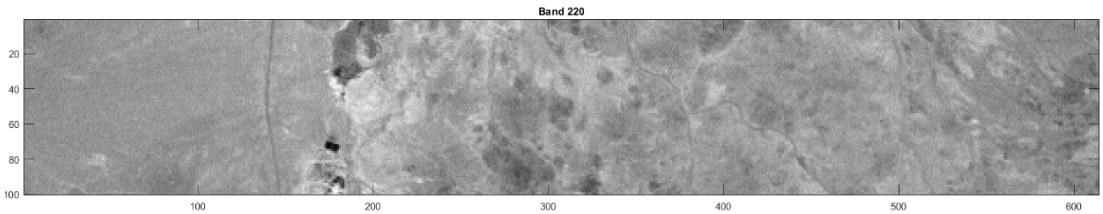

Figure 2.1: Band 220 from the Cuprite scene 02 [3].

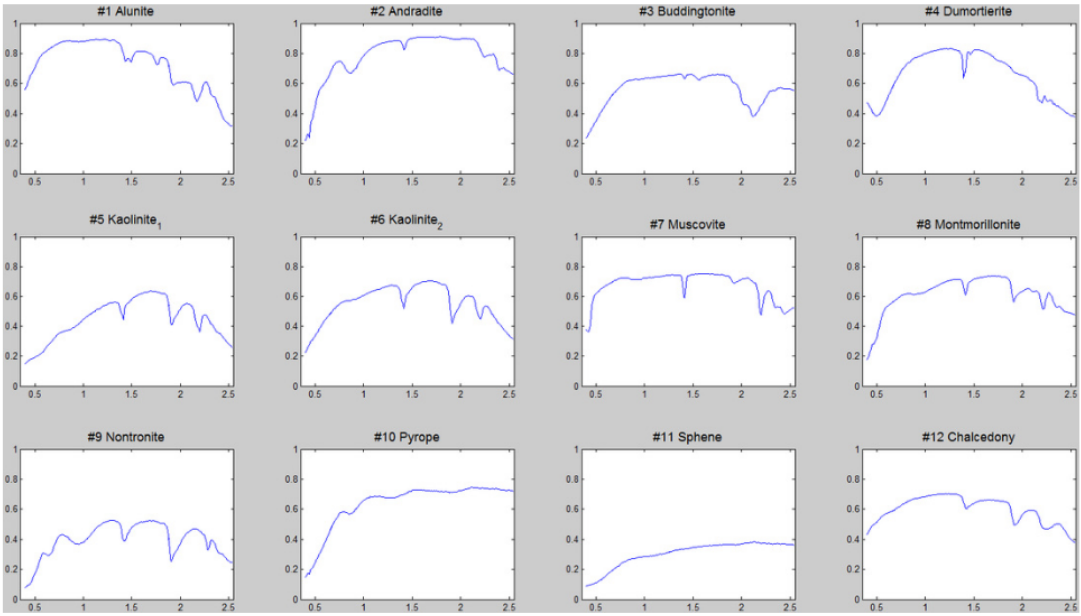

Figure 2.2: Spectral signatures of minerals from the Cuprite mining district [4].

2.1.2 NTNU SmallSat project

The hyperspectral imager used in the NTNU SmallSat project has $N_{BANDS} = 100$ usable bands. For a generic image cube the number of spectral bands is referred to as P_bands. NTNU SmallSat's imager has a sensor resolution of 2048 x 1088 pixels. The number of effective pixels per row of the image, N_{pixels}, is 578, and the number of pixel rows, N_{rows}, is 1088. A hyperspectral image cube of size $N_{pixels} \times N_{rows} \times P\_bands$ can be seen in Figure 2.4. Each of the elements in the cube has a width of $Pixel\_data\_width$. A generic spectral pixel vector can be seen in Figure 2.3. Figure 2.5 displays the functionality of the hyperspectral imager. The imager captures data one pixel at the time, in a row-wise fashion [5].

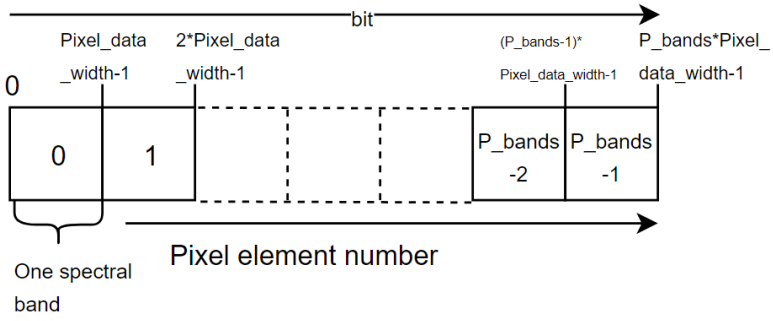

Figure 2.3: A spectral pixel vector.

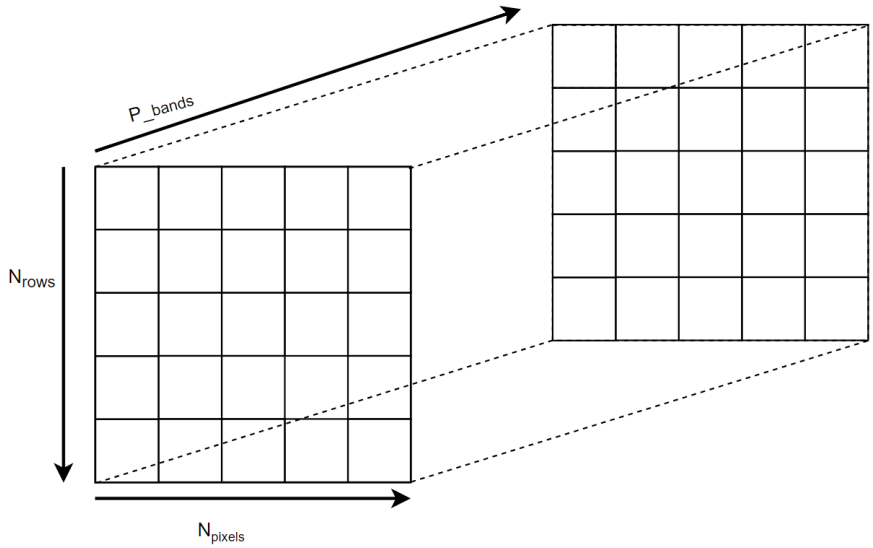

Figure 2.4: Hyperspectral image cube.

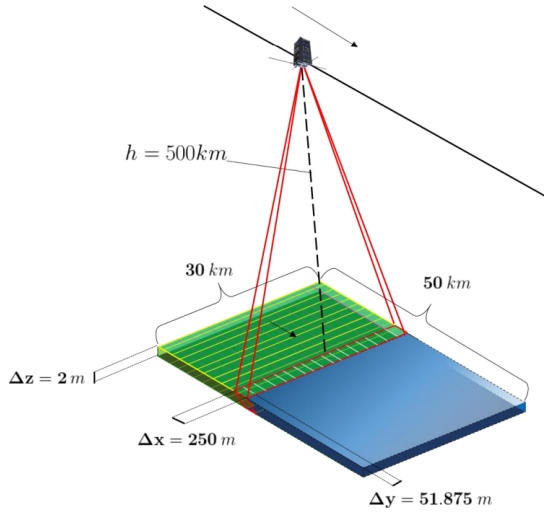

Figure 2.5: Push-broom hyperspectral imager mode of operation [5].

2.2 NTNU SmallSat's hardware platform

The NTNU SmallSat's on-board processing system is a Zynq-7000 series System-on-Chip (SoC). The SoC can be divided into two parts: the processing system and the programmable logic. This is illustrated in Figure 2.6. The processing system consists of a dual core ARM Cortex A9, while the programmable logic main processing unit is a Artix-7 or Kintex-7 Series FPGA, depending upon the version of the Zynq-7000 series.

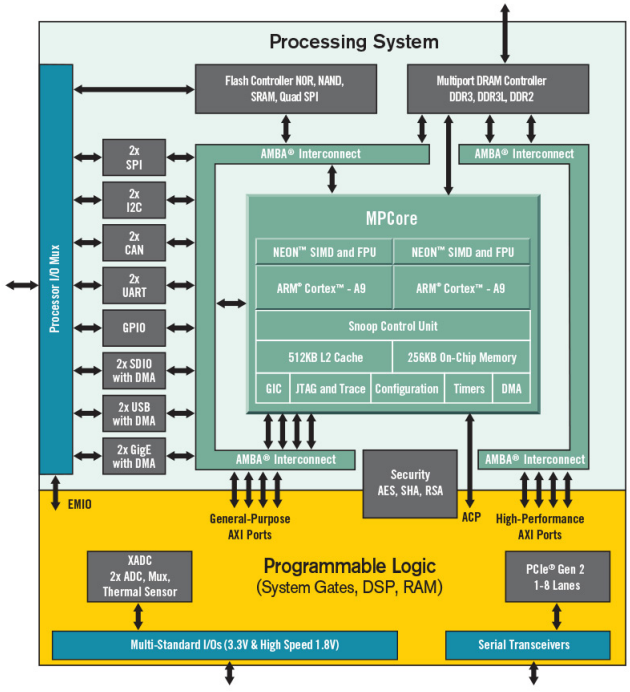

Figure 2.6: Zynq-7000 architecture [6].

In NTNU SmallSat Project, an initial prototype will be developed on a Zynq Zedboard Evaluation and Development kit (from now on referred to as Zedboard), featuring a Zynq-Z7020, which contains an Artix-7 device. Later stage prototypes will feature Zynq-Z7030 or Z-7035. These contain Kintex-7 devices.

The anomaly detection results will be transmitted from the satellite to a ground base station. The data budget for packet transmissions is as given on page 23 in [5].

2.2.1 AXI-Stream

AXI-Stream is a slimmed-down protocol for transfers, without any concept of addresses, where data is moved from one point to another. It is based on the read and write channels in the AXI protocol. As for AXI buses, handshaking signals (**TREADY** and **TVALID**) are used when transferring data.

The Cube Direct Memory Access (DMA) [16] used by NTNU SmallSat utilizes AXI-Stream as the communication protocol between Intellectual Properties (IP). The operating frequency of the AXI-Stream protocol in the NTNU SmallSat project is 100 MHz. The Cube DMA will be interfaced by the anomaly detector.

2.3 Anomaly detection

The process of detecting anomalies in a hyperspectral image is called anomaly detection. For a spectral vector to be considered as an anomaly, it has to be significantly different to its neighboring background. Four issues arising in anomaly detection are [7]:

- Q1: How large should a target be to be considered as an anomaly?
- Q2: How does an anomaly respond to its neighbouring pixels?
- Q3: How sensitive is anomaly detection to noise?
- Q4: How are different anomalies to be detected and classified?

The above issues are important for the choice of anomaly detection algorithm, and will be further discussed in this chapter.
Algorithms used for anomaly detection output a scalar for each pixel in an image indicating the relative probability that the spectral pixel vector is an anomaly. A higher output indicates a higher probability that the pixel vector is an anomaly.

2.3.1 Reed-Xiaoli algorithm

The Reed-Xiaoli (RX) algorithm [17] is one of the most widely used algorithms for anomaly detection in HSI, and it is considered as the benchmark anomaly detection algorithm for hyperspectral data [13].
The RX algorithm was developed to address the scenario where no prior knowledge about the target signatures is available. Assuming that a single pixel target, $\mathbf{x}$, is the observation test vector, the result of the RX algorithm is given by the filter in equation 2.1;

$$RX(\mathbf{x}) = (\mathbf{x} - \mathbf{u}_b)^T \Sigma^{-1} (\mathbf{x} - \mathbf{u}_b), \tag{2.1}$$

where $\mathbf{u}_b$ is the estimated background clutter sample mean, computed from the set of all pixel vectors in the image (referred to as the *global set*). Σ is the estimated background clutter covariance, estimated on the global set. Since the covariance is computed on the global set of pixels, the HSI needs to collect all data contained in the entire image before the RX AD can start executing. This means that the RX-algorithm does not have the possibility to operate in real-time.

2.3.2 Local RX algorithm

An often used and important variant of the RX algorithm is the local RX (LRX) [9] algorithm. By substituting the sample covariance matrix computed on the global set with the correlation matrix computed on a kernel of size $K \times K$ pixel vectors, it is possible to increase the parallelism of the AD and get near real-time performance. The LRX can be considered as a local AD because each pixel of the image has its own correlation matrix.

Each correlation matrix is computed on a square kernel of size $K \times K$. The result of the LRX AD can be expressed as follows:

$$\delta_K^{LRX}(\mathbf{x}) = \mathbf{x}^T \mathbf{R}_{K\times K}^{-1}(\mathbf{x})\mathbf{x}, \tag{2.2}$$

where $\mathbf{x}$ is the observation test pixel vector, $\mathbf{R}_{K\times K}(\mathbf{x})$ is the correlation matrix of pixel vector $\mathbf{x}$ computed on a square kernel of size $K \times K$ containing local neighbouring pixels. See Figure 2.7.

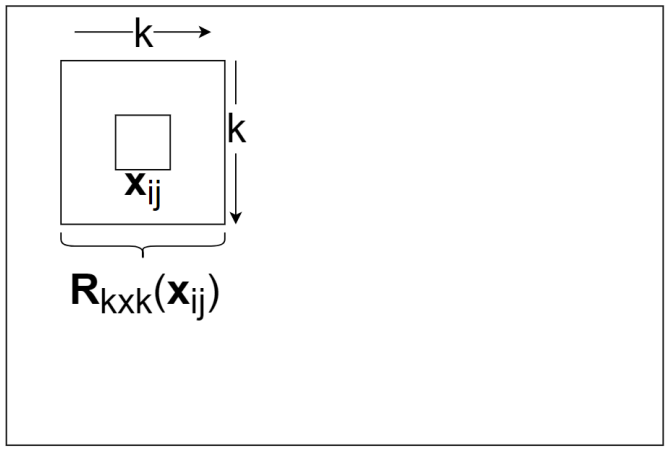

Figure 2.7: Visualization of a kernel of size $K \times K$ used in LRX.

2.3.3 Adaptive Causal anomaly detection

An AD developed to solve the issues of the RX AD is the Adaptive Causal anomaly detection (ACAD) [7]. One issue of the RX algorithm is that previously detected anomalies with strong spectral signatures may have an impact upon the detection of later anomalies, as they might influence what is considered the background, which is shown in [7]. ACAD is adaptive in the way that it builds a map of detected anomalies and removes the previously detected anomaly pixel vectors from the causal sample correlation set.
Another benefit of ACAD relative to RX and LRX is that it might be computed in real-time. This is achieved by using the causal correlation matrix $\mathbf{R}(\mathbf{x}_k)$, presented in equation 2.3:

$$\mathbf{R}(\mathbf{x}_k) = \frac{1}{k}\sum_{i=1}^{k} \mathbf{x}_i \mathbf{x}_i^T, \tag{2.3}$$

instead of the covariance or the correlation matrix computed on the global or a local set of pixel vectors, as in RX and LRX, respectively. In equation 2.3, $\mathbf{x}_k$ is the observation test pixel vector, and k is the index of the pixel vector currently being processed. The summation in equation 2.3 sums the correlation matrix for the pixel sample vectors $\mathbf{x}_1$, ...$\mathbf{x}_k$.

To remove the previously detected anomalous pixel vectors from the correlation set, the sample spectral correlation matrix, referred to as the causal anomaly-removed sample

spectral correlation matrix, is presented in equation 2.4 [7]:

$$\tilde{\mathbf{R}}(\mathbf{x}_k) = \mathbf{R}(\mathbf{x}_k) - \sum_{t_j \in \Delta(k)} \mathbf{t}_j \mathbf{t}_j^T, \tag{2.4}$$

where $\Delta(k)$ is the set of all earlier detected anomalous pixel vectors $\mathbf{t}_j$ prior to the image pixel currently being processed, $\mathbf{x}_k$.
ACAD can then be defined as follows:

$$\delta^{ACAD}(\mathbf{x}_k) = \mathbf{x}_k^T \tilde{\mathbf{R}}^{-1}(\mathbf{x}_k)\mathbf{x}_k. \tag{2.5}$$

ACAD is a causal filter, meaning that only the pixels previously processed and the current pixel are used for anomaly detection. ACAD computes the causal correlation matrix for the previously captured pixel sample vectors $\mathbf{x}_1$, ...$\mathbf{x}_{k-1}$ up to the pixel currently being processed, $\mathbf{x}_k$, as shown in equation 2.3. This means that ACAD might be implemented in real-time or near real-time, as pixels can be processed as soon as they are captured by the push-broom HSI. ACAD does not need to wait for the entire image to be loaded into memory.

Processing of pixels in ACAD can be visualized in Figure 2.8:

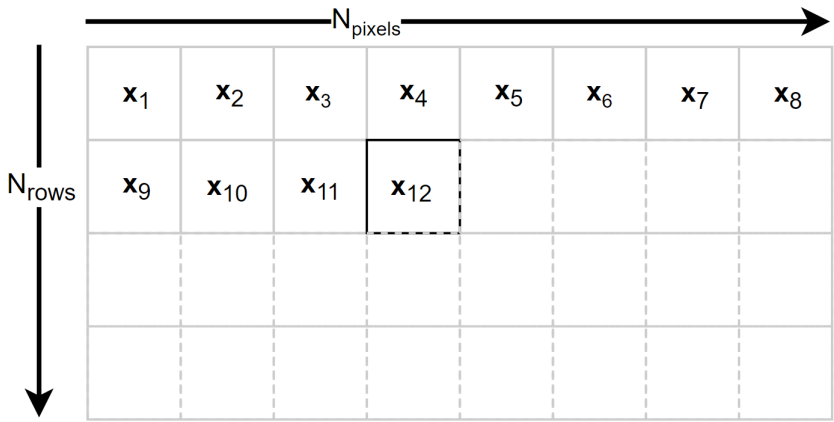

Figure 2.8: Visualizing processing of pixels in ACAD.

$\mathbf{x}_{12}$ is the pixel currently being processed. The previously captured pixel sample vectors are marked by solid grey lines. Pixels that have not been captured and processed are marked by dashed grey lines.

An anomalous pixel vector has a significant spectral vector difference from its surroundings. Since ACAD is causal, the surroundings are defined as the n_{ACAD} previously processed pixels. ACAD defines the variable u_k, used to evaluate if a pixel vector is anomalous, as shown in equation 2.6:

$$u_k = \frac{1}{n_{ACAD}} \sum_{i-1}^{n_{ACAD}} \delta^{ACAD}(\mathbf{x}_{k-i}). \tag{2.6}$$

In order to classify if a pixel vector is anomalous the variable t_k is introduced, defined in equation 2.7:

$$t_k = \delta^{ACAD}(\mathbf{x}_k) - u_k. \tag{2.7}$$

If t_k is greater than a predetermined value τ, the pixel vector is considered to be an anomaly and added to the set of anomalous targets. If not, it is used in subsequent data processing. The anomaly map created by ACAD is shown in equation 2.8:

$$map_{ACAD}(t_k) = \begin{cases} 1, \text{ if } t_k > \tau \\ 0, \text{ otherwise.} \end{cases} \tag{2.8}$$

The four issues labelled Q1, Q2, Q3 and Q4 still remain. For a pixel vector to be considered anomalous it has to be relatively small compared to the size of the image. The relationship between the size of an anomaly and the size of the entire image is β, shown in equation 2.9:

$$\beta = \frac{Image\_size}{size\_of\_anomaly}. \tag{2.9}$$

Empirical results show that β will be ≈ 100 [7]. The relationship between β and n_{ACAD} is shown in equation 2.10:

$$n_{ACAD} = \frac{N\_PIXELS\_TOT}{\beta}, \tag{2.10}$$

where N_PIXELS_TOT is the total number of pixels in the image. In RX and LRX, an earlier detected anomaly with a strong spectral signature may influence the detection of subsequent anomalies, as the anomalies are used for calculation of the correlation or covariance matrix. This is shown in Figure 12 in [7], where an anomaly with a strong spectral signature influences the RX detector to such a degree that it fails to detect four subsequent anomalous pixels. This problem is solved in ACAD by removing previously detected anomalous pixels from the sample spectral correlation matrix, as shown in equation 2.4.

In [7], noise-immunity tests have been done on different ADs, including RX and ACAD. These tests add Gaussian noise with a Signal-to-noise ratio (SNR) of 20:1, 10:1 and 5:1 to a test image. One of the conclusions is that noise has less effects on ACAD compared to the RX detector as shown in Figure 2.9 [7].

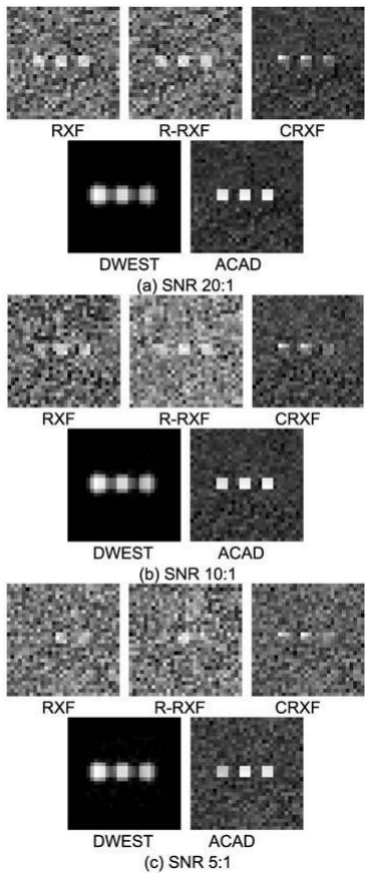

Figure 2.9: Results of noise tests [7].

2.3.4 Adaptive Local RX

A modification of the LRX algorithm is the Adaptive Local RX (ALRX) presented by the author. To the author's knowledge, it is not yet described in literature.

ALRX is inspired by the anomaly-map creation and the removal of previously detected anomaly pixel vectors from the causal sample correlation set as done in the ACAD AD. Similarly, ALRX builds an anomaly map and removes previously detected anomaly pixel vectors that are located within the local window from the local sample correlation set. The result of the ALRX AD is shown in equation 2.11:

$$\delta_k^{ALRX}(\mathbf{x}) = \mathbf{x}^T \tilde{\mathbf{R}}_{K\times K}(\mathbf{x})^{-1}\mathbf{x}. \tag{2.11}$$

$$\tilde{\mathbf{R}}_{K\times K}(\mathbf{x}) = \mathbf{R}_{K\times K}(\mathbf{x}) - \sum_{t_j \in \Delta(k_{K\times K})} \mathbf{t}_j \mathbf{t}_j^T, \tag{2.12}$$

where $\Delta(k_{K\times K})$ is the set of previously detected anomalous pixel vectors $\mathbf{t}_j$ located within the local window of size $K \times K$ with center in the image pixel vector currently being processed, $\mathbf{x}$.

2.4 Inverse matrix

The computation of the inverse of a matrix is a part of all the considered ADs. This is a computationally intensive task. There exist multiple algorithms for computing the matrix inverse. One option is to do QR factorization [18], and compute the inverse of QR. In hardware (HW), the QR-factorization is most often computed using Givens rotation enabled by a trigonometric algorithm called COrdinate Rotation DIgital Computer (CORDIC) [19].

Another option is to implement the inverse matrix computation by doing Gauss-Jordan elimination [8]. The Gauss-Jordan elimination is highly parallelizable [8] and less complex than the QR-factorization enabled by CORDIC. A pseudo-code for computing the Gauss-Jordan elimination is shown in Figure 2.10. The Gauss-Jordan elimination can be tiled into three parts: forward elimination, backward elimination and last division, marked by black, red and green squares in Figure 2.10, respectively.

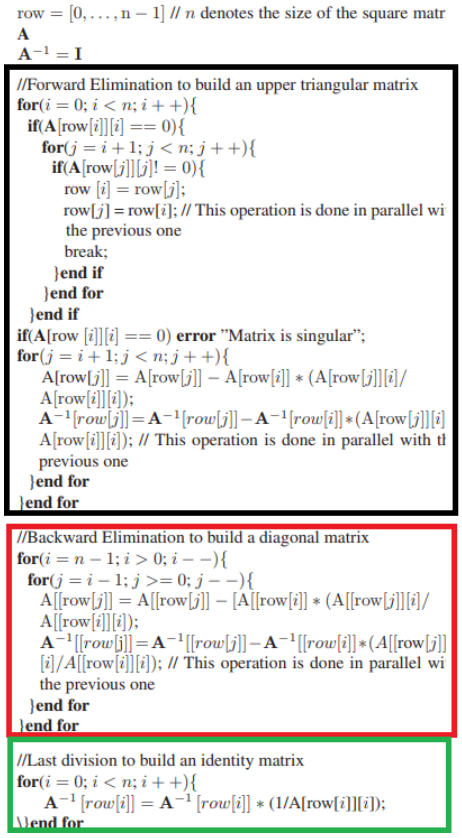

Figure 2.10: Pseudo-code for computing the inverse of a matrix by Gauss-Jordan elimination [8].

2.4.1 Gauss-Jordan elimination

There are three different types of row operations performed on the rows of a matrix in Gauss-Jordan elimination:

1. Swap the positions of two rows.
2. Multiply a row by a nonzero scalar.
3. Adding a scalar multiple of one row to another.

2.4.1.1 Forward elimination

The first part of the Gauss-Jordan elimination reduces the matrix to row echelon form (upper triangular matrix) by using row operations, starting at the topper-most row of the matrix, which is denoted as index zero, and iterating downwards. Two indexes are used, the outer index i and the inner index j. This is shown in Figure 2.11. Forward elimination may use all of the different row operations listed, depending upon the existence of a zero element in the diagonal of the matrix.

```
//Forward Elimination to build an upper triangular matrix
for(i = 0; i < n; i++){
  if(A[row[i]][i] == 0){
    for(j = i + 1; j < n; j++){
      if(A[row[j]][j]! = 0){
        row [i] = row[j];
        row[j] = row[i]; // This operation is done in parallel with
         the previous one
        break;
      }end if
    }end for
  }end if
if(A[row [i]][i] == 0) error "Matrix is singular";
for(j = i + 1; j < n; j++){
    A[row[j]] = A[row[j]] − A[row[i]] ∗ (A[row[j]][i]/
    A[row[i]][i]);
    A^-1[row[j]] = A^-1[row[j]] − A^-1[row[i]] ∗ (A[row[j]][i]/
    A[row[i]][i]); // This operation is done in parallel with the
    previous one
  }end for
}end for
```

Figure 2.11: Pseudo-code for computing the forward elimination in Gauss-Jordan elimination [8].

2.4.1.2 Backward elimination

Backward elimination utilizes row operation two and three mentioned in the list on the previous page, in order to create a diagonal matrix. It starts at the bottom of the matrix, denoted index $P\_bands - 1$ and iterates upwards. Two indexes are used, the outer index i and the inner index j. This is shown in Figure 2.12.

```
//Backward Elimination to build a diagonal matrix
for(i = n − 1; i > 0; i − −){
  for(j = i − 1; j >= 0; j − −){
    A[[row[j]] = A[[row[j]] − [A[[row[i]] ∗ (A[[row[j]][i]/
    A[[row[i]][i]);
    A^-1[[row[j]] = A^-1[[row[j]] − A^-1[[row[i]] ∗ (A[[row[j]]
    [i]/A[[row[i]][i]); // This operation is done in parallel with
    the previous one
  }end for
}end for
```

Figure 2.12: Pseudo-code for computing the backward elimination in Gauss-Jordan elimination [8].

2.4.1.3 Last division

Last division is the last step of the Gauss-Jordan elimination. The last division starts at the topper-most index of the matrix and iterates downwards as shown in Figure 2.13. It creates the matrix A^{-1} by utilizing the second type of row operations. A^{-1} is the inverse matrix of A which fulfills the property $A \times A^{-1} = I$. I is the identity matrix of size $P\_bands \times P\_bands$ containing zero elements, except for the diagonal of the matrix, which contains ones.

```
//Last division to build an identity matrix
for(i = 0; i < n; i++){
    A^-1[row[i]] = A^-1[row[i]] * (1/A[row[i]][i]);
\}end for
```

Figure 2.13: Pseudo-code for computing the last division in Gauss-Jordan elimination [8].

Chapter 3

Review of state of the art anomaly detectors

To evaluate the performance of the ADs considered in this thesis, models of the algorithms described in Section 2.3 were developed in MATLAB, and tested on both hyperspectral image data from the Cuprite site [3] captured by the AVIRIS imager and synthetic images created by the author. The creation of the synthetic images is described in Section 3.1.

The MATLAB hyperspectral toolbox [20] was used for image preprocessing, visualization and for having a good starting point for developing further functionality. The toolbox included an implementation of the RX algorithm. A fork of the toolbox was made, available at [21], to be able to do MATLAB implementations of LRX, ALRX and ACAD in order to evaluate the performance of the ADs. The most important scripts and functions from the forked toolbox are also located in Appendix A.

3.1 Experiments on synthetic images

To make an objective analysis of the considered ADs performance, synthetic hyperspectral images were created. These images contain inserted pixels flagged as anomalies, with known spectral signature and position, to be able to create a reference anomaly map. The anomalies are of various sizes, to test the ADs ability to detect variably sized anomalies. A similar test is also done in [9]. Figure 3.1 [9] shows that the RX (referred to as Global RX (GRX) [9]) algorithm was not able to detect anomalies in the third and four column (anomalous pixels made up of $> 50\%$ abundance of the anomaly signature). The LRX exhibits slightly better anomaly detection accuracy in this test.

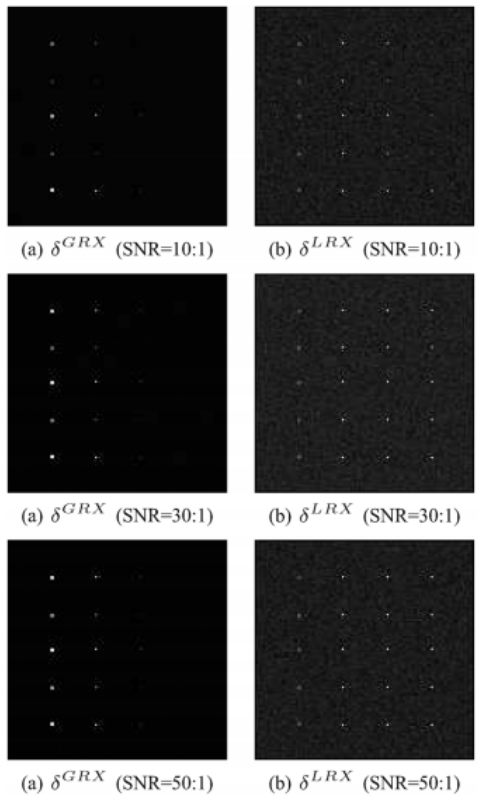

Figure 3.1: Test of RX (GRX) and LRX algorithms [9].

The purpose of the synthetic images used in this thesis was to provide means for doing objective evaluations of the performance of the considered ADs. Two different metrics are important in order to evaluate the performance of the ADs: $false\_anomalies$ and $correctly\_predicted\_anomalies$. These metrics are defined in equation 3.1 and 3.2;

$$false\_anomalies = predicted\_anomalies - true\_anomalies, \tag{3.1}$$

in which $predicted\_anomalies$ is the number of predicted anomalies by the anomaly detector and $true\_anomalies$ are actual anomalies found by the anomaly detector.

$$correctly\_predicted\_anomalies = \frac{predicted\_anomalies\_in\_reference\_map}{reference\_anomalies} \tag{3.2}$$

Parameter $predicted\_anomalies\_in\_reference\_map$ is the number of predicted anomalies that are also found in the reference anomaly map. Parameter $reference\_anomalies$ is the number of anomalies in the reference anomaly map. These metrics are important as they provide an objective way to evaluate the performance of the ADs, something that can not be done with real hyperspectral image data, unless one possesses a reference anomaly map to the real image data, which the author has not been able to find.

Synthetic images with different sizes and anomaly sizes were created to evaluate the performance of the ADs. Tests were performed to get an objective evaluation of the

considered ADs. To be able to compare to the test done on images with a size of 200 × 200 as described in chapter 5.5.1 in [10] by Hsueh, this test were mimicked. The synthetic image used in this mimicked test can be seen in Figure 3.2.

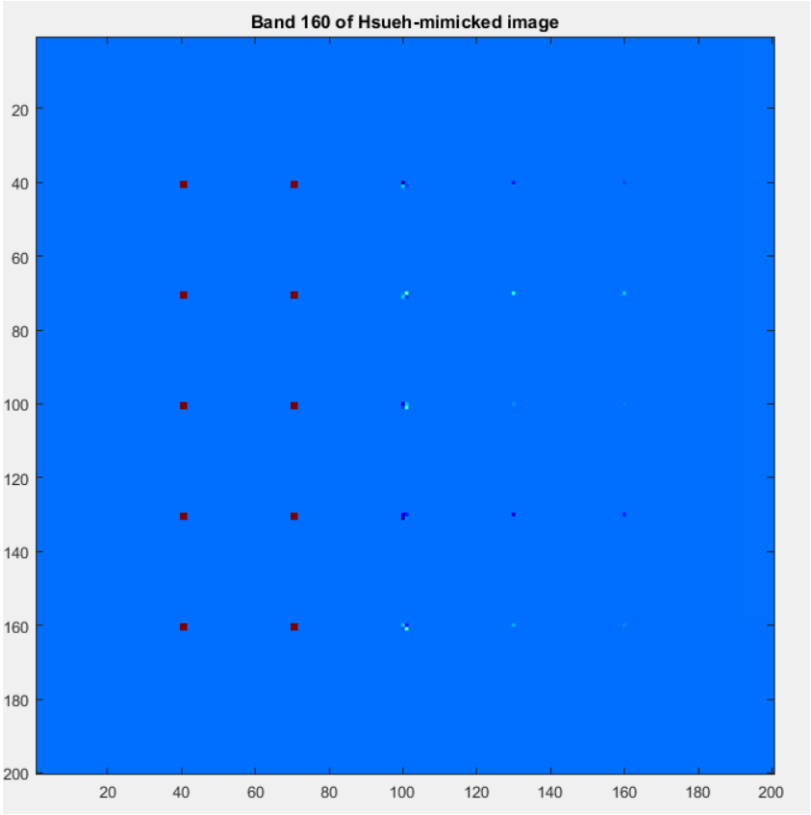

Figure 3.2: First class of synthetic images: 200 × 200 synthetic image with 25 inserted anomaly panels as describe by Hsueh in [10].

Additionally, synthetic images with a size of 30 × 30 pixels were created with an anomalous panel of size 2×2 inserted into the center of the images. This image scene is labelled $Sim30\_30AVIRIS$. The anomalous pixels are pure pixels with a spectral signature of Buddingtonite. These synthetic images has a background consisting of 33% Alunite, 33% Kalonite and 33% Pyrope. Such a generated synthetic image can be seen in Figure 3.3, displaying spectral band 160.

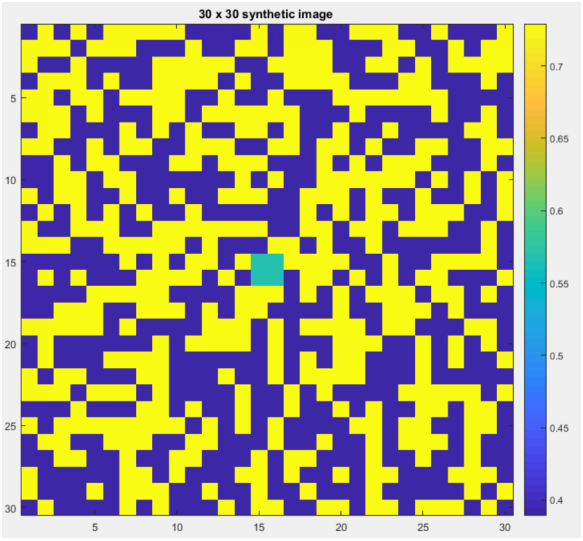

Figure 3.3: Second class of synthetic images: Synthetic 30 × 30 image with an inserted 2x2 anomalous panel inserted into the center.

A third class of synthetic images with a size of 100 × 614 pixels was also created, labelled *SimAviris*01. These images have a background consisting of 33% Alunite, 33% Kalonite and 33% Pyrope. The anomalous pixels are pure pixels with a spectral signature of Buddingtonite, extracted from the Cuprite mining scene [22]. Six different sized anomalous target kernels were made, with a size of 1×1, 2×2, 5×5, 10×10, 15×15, 20×20 and 25×25 pixels. Table 3.1 describes the position and size of anomalous areas of *SimAviris*01.

Table 3.1: Properties of *SimAviris*01. Row and column locations are location of the center pixel in the kernel of size $K \times K$.

Scene	**Row**	**Column**	**Anomaly size [pixels x pixels]**
SimAviris01	35	50	1x1
SimAviris01	70	50	1x1
SimAviris01	35	100	2x2
SimAviris01	70	100	2x2
SimAviris01	35	150	5x5
SimAviris01	70	150	5x5
SimAviris01	35	250	10x10
SimAviris01	70	250	10x10
SimAviris01	35	350	15x15
SimAviris01	70	350	15x15
SimAviris01	35	450	20x20
SimAviris01	70	450	20x20
SimAviris01	35	550	25x25
SimAviris01	70	550	25x25

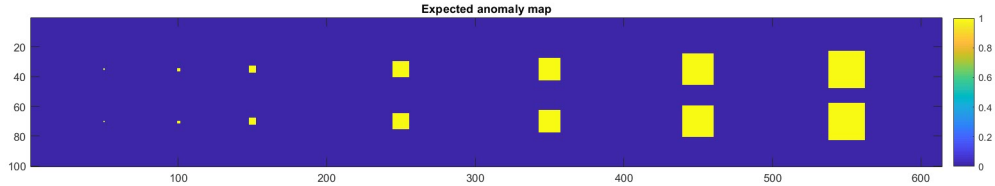

Figure 3.4: Expected anomaly map for the third class of synthetic images created.

Since the RX and LRX AD do not build an anomaly map, the author defines anomalous pixels as pixels having a score of $\geqslant 75\%$ of the maximum value outputted from the RX AD or the LRX AD. This is done in order to be able to set the objective metrics $false\_anomalies$ and $correctly\_predicted\_anomalies$.

3.1.1 RX detection results

3.1.1.1 Hsueh-mimicked image

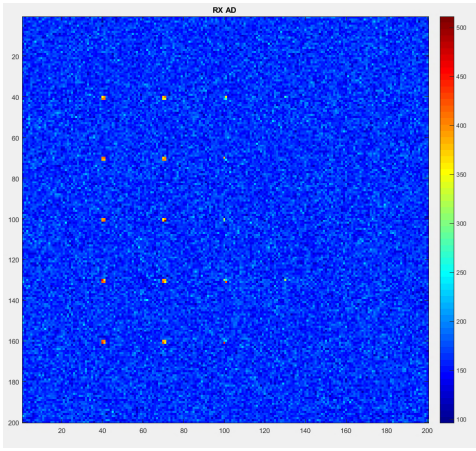

Figure 3.5: RX AD result.

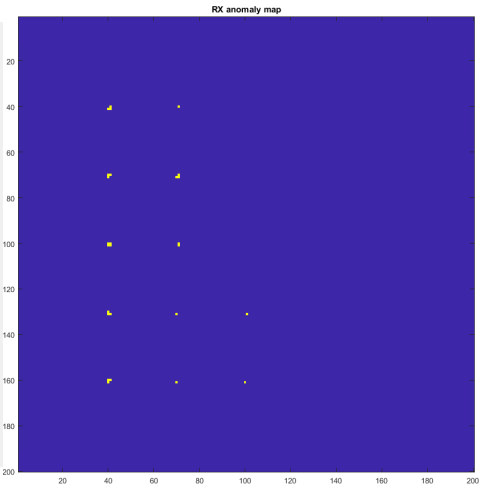

Figure 3.6: Generated anomaly map.

Figure 3.7: RX AD test on synthetic image based on Hsueh's description. The map in Figure 3.6 was created to provide a way of computing $false\_anomalies$ and $correctly\_predicted\_anomalies$.

For this test, $false\_anomalies = 0$, and $correctly\_predicted\_anomalies = 0.3714$. A score of 0.3714 for $correctly\_predicted\_anomalies$ is reasonable when comparing to the tests done in [9], where no anomalies were detected in the fourth and fifth column, shown in Figure 3.1. As can be seen in 3.7, the RX detector is not able to detect the smallest anomalous targets, which leads to the poor score.

3.1.1.2 $Sim30\_30AVIRIS$ scene

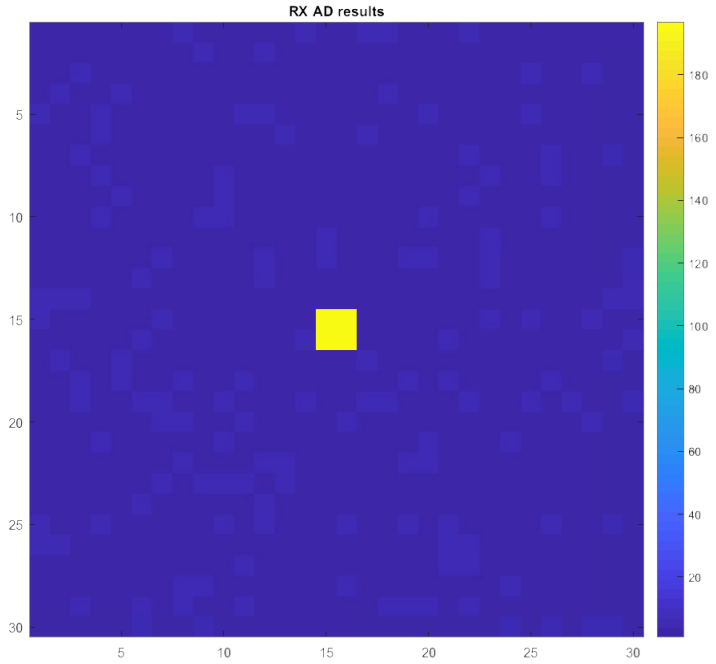

Figure 3.8: RX AD results for the $Sim30\_30AVIRIS$ scene.

The RX detector performs well for the $Sim30\_AVIRIS$ scene, which can be seen in Figure 3.8. Parameter $false\_anomalies = 0$ and $correctly\_predicted\_anomalies = 1$ for this test.

3.1.1.3 $Sim\_Aviris01$ scene

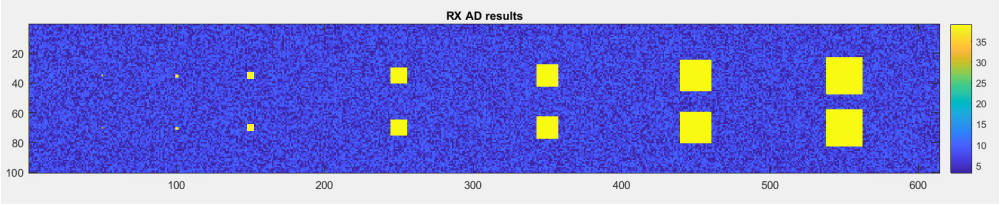

Figure 3.9: RX AD results for the $Sim\_Aviris01$ scene.

The RX detector performs well for the $Sim\_Aviris01$ scene. RX AD results, shown in Figure 3.9, are quite similar to the expected anomaly map, shown in Figure 3.4. $false\_anomalies = 0$ and $correctly\_predicted\_anomalies = 1$ for this test.

3.1.2 LRX detection results

LRX was tested for different kernel sizes K=[5, 10, 15, 20, 23, 25 and 30]. These values were chosen as [9] tested LRX and optimized the value K empirically, in the range of K =[3,30].

3.1.2.1 Hsueh mimicked image

K	$false\_anomalies$	$correctly\_predicted\_anomalies$
5	967	0.0429
10	62	0.1143
15	58	0.1714
20	50	0.2857
23	46	0.4782
25	34	0.5143
30	38	0.4571

Table 3.2: LRX results on Hsueh scene.

The LRX yields better results for *correctly_predicted_anomalies* on the Hsueh test than the RX AD with a top score of 0.5143 for kernel size K =25. It does however have a higher number of $false\_anomalies$, with the best score of 34.

3.1.2.2 $Sim30\_30AVIRIS$ scene

K	$false\_anomalies$	$correctly\_predicted\_anomalies$
5	12	0.33
10	0	1
15	0	1
20	0	1
23	0	1
25	0	1
30	0	1

Table 3.3: LRX detection results on $SIM30\_30AVIRIS$ scene.

LRX performs well for the $Sim30\_30AVIRIS$ scene, and yields perfect results for $K>=$ 10, as shown in Table 3.3.

3.1.2.3 *SimAviris*01

K	$false\_anomalies$	$correctly\_predicted\_anomalies$
5	703	0.3562
10	2584	0.5106
15	2247	0.5621
20	2167	0.5710
23	1379	0.6737
25	1126	0.7192
30	1056	0.7208

Table 3.4: LRX results on *SimAviris*01 scene.

Table 3.4 shows that the LRX AD struggles more on the *SimAviris*01 scene. The LRX produces a significant number of $false\_anomalies$, with a score of 1379 and 1126 for kernels sizes K of 23 and 25 respectively. The best score for $correctly\_predicted\_anomalies$ is 0.7208 for $K = 30$.

3.1.3 ALRX detection results

The threshold τ used in the ALRX algorithm was tested in range [0.5, 100]. The values of τ yielding best results for K are presented in Table 3.5 and 3.6. The results gathered from the ALRX detection are worse than those for RX, LRX and ACAD. The work done on improving the algorithm was stopped to be able to prioritize hardware implementation of an anomaly detector.

3.1.3.1 $SIM30\_30AVIRIS$

τ	K	$false\_anomalies$	$correctly\_predicted\_anomalies$
3.5	5	50	0
3.5	10	46	0
90	15	0	0.5
90	20	0	0.5
90	23	0	0.5
90	25	0	0.5
90	30	0	0.5

Table 3.5: ALRX results on $SIM30\_30AVIRIS$ scene.

3.1.3.2 *SimAviris*01

τ	K	$false\_anomalies$	$correctly\_predicted\_anomalies$
3.5	5	4760	0.3447
3.5	10	2953	0.1553
3.5	15	2871	0.1252
3.5	20	2985	0.0888

Table 3.6: ALRX results on *SimAviris*01 scene.

3.1.4 ACAD

The ACAD algorithm was extensively tested by Hsueh *et al* [10]. In this thesis, ACAD was tested on the $SIM\_AVIRIS\_30\_30$ scene and the *SimAviris*01 scene, to be able to do a comparison to the other ADs performances.

3.1.4.1 $SIM\_AVIRIS\_30\_30$

τ	$false\_anomalies$	$correctly\_predicted\_anomalies$
0.1	8	0.33
0.3	6	0.4
0.5	4	0.5
0.7	2	0.6667
0.8	1	0.25
0.9	0	0
1	0	0

Table 3.7: ACAD results on $SIM\_AVIRIS\_30\_30$ scene.

Table 3.7 shows that ACAD yields poorer results than both RX and LRX for the $SIM\_AVIRIS\_30\_30$ scene. The best performance of ACAD is for $\tau = 0.7$, resulting in $false\_anomalies = 2$ and $correctly\_predicted\_anomalies = 0.667$.

3.1.4.2 *SimAviris*01

τ	$false\_anomalies$	$correctly\_predicted\_anomalies$
0.1	552	1
0.2	491	1
0.3	429	1
0.4	368	1
0.5	306	0.6727
0.6	245	0.3457
0.7	184	0.1685
0.8	122	0.0631
0.9	61	0.0420
1	0	0

Table 3.8: ACAD results on *SimAviris*01 scene.

Table 3.7 shows that ACAD is able to accomplish a score of 1 for $correctly\_predicted\_anomalies$, but produces a high number of $false\_anomalies$. The best performance of the ACAD for this scene is for τ =0.4, yielding $correctly\_predicted\_anomalies = 1$ and $false\_anomalies = 368$, which is better than the results accomplished by the LRX detector on this scene.

3.2 Testing on real image data

The ADs considered in this thesis were tested on hyperspectral image data from the Cuprite mining area captured by the AVIRIS hyperspectral camera to evaluate their performance. Band 220 from Cuprite scene 02 can be seen in Figure 3.10. As no reference anomaly map for this data has been found, it is not possible to calculate $false\_anomalies$ and $correctly\_predicted\_anomalies$. The real image data provides a subjective method of evaluating the performance of the ADs.

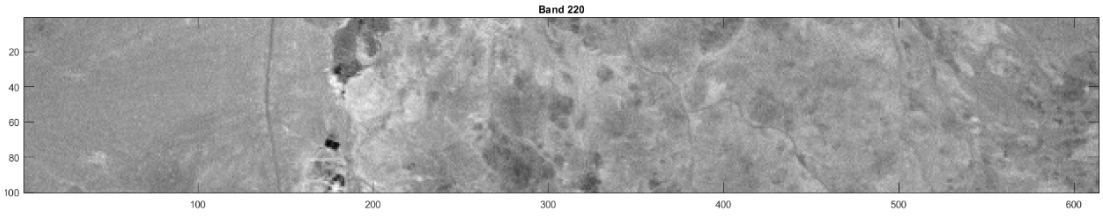

Figure 3.10: Band 220 from the Cuprite scene 02[3].

3.2.1 RX

Figure 3.11 shows the result of the RX AD on the Cuprite scene 02. A higher score indicates a higher likelihood for the pixel being anomalous.

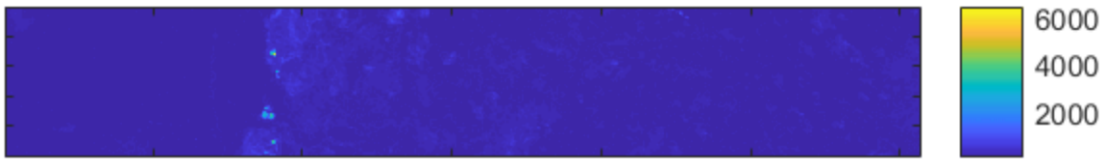

Figure 3.11: Result from RX AD on Cuprite image data.

3.2.2 LRX

Figure 3.12 shows the result of the LRX AD on the Cuprite scene 02. $K = 23$ was chosen due to the results presented in Table 3.4 and the evaluation done in [9], which concluded that K=23 yielded the best trade-off between detection accuracy and computational burden.

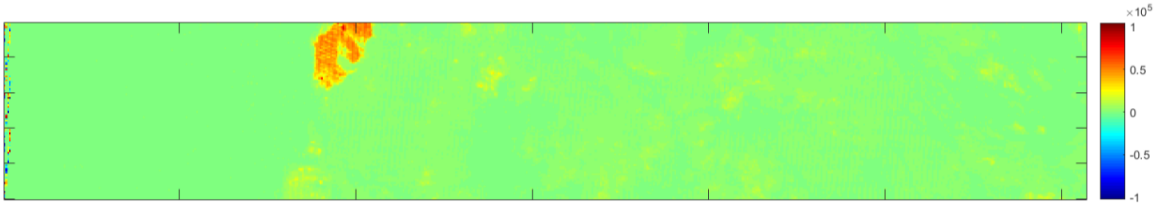

Figure 3.12: Result from LRX AD with a kernel size of K=23 on Cuprite image scene 02.

3.2.3 ACAD

The resulting anomaly map from ACAD AD on the Cuprite scene 02 is shown in Figure 3.13. τ is set to 250. The anomaly map is overlayed over Figure 3.10.

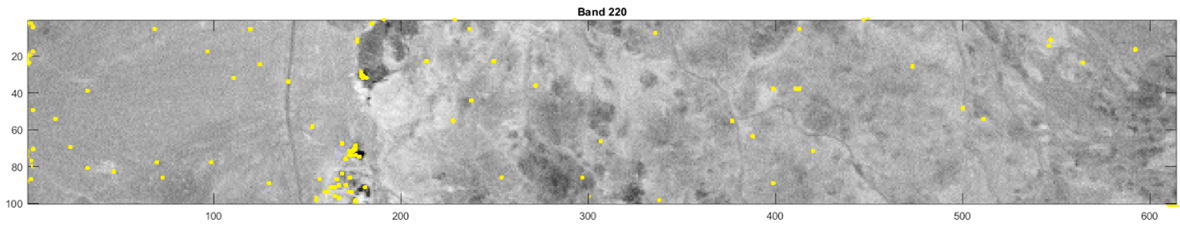

Figure 3.13: Anomaly map created by ACAD (yellow dots) overlayed over Figure 3.10.

3.2.4 Choice of anomaly detector algorithm

Table 3.9 summarizes the comparison of the different ADs.

AD	*false_anomalies* (best performance)	*correctly_predicted_anomalies* (best performance)	**Performance on real data**	**Possibility of implementing in real time**
RX	Hsueh : 0 $SIM30\_30AVIRIS$: 0 $Sim\_Aviris01$: 0	Hsueh: 0.3714 $Sim30\_30AVIRIS$: 1 $Sim\_Aviris01$: 1	Figure 3.11	Low. Need global covariance matrix before computing inverse.
LRX	Hsueh: 50 $SIM30\_30\_AVIRIS$: 0 $SimAviris01$: 1056	Hsueh: 0.5143 $SIM30\_30\_AVIRIS$: 1 $SimAviris01$: 0.7208	Figure 3.12	Medium. Need to wait for a window of size $K \times K$ to be captured by the imager before processing can start.
ALRX	Hsueh: - $SIM30\_30\_AVIRIS$: 0 $SimAviris01$: 4760	Hsueh:- $SIM30\_30\_AVIRIS$: 0.5 $SimAviris01$: 0.3447	-	Medium. Need to wait for a window of size $K \times K$ to be captured by the imager before processing can start.
ACAD	Hsueh:- $SIM30\_30\_AVIRIS$: 2 $SimAviris01$: 368	Hsueh:- $SIM30\_30\_AVIRIS$: 0.667 $SimAviris01$: 1	Figure 3.13	High. Pixels can be processed as soon as they are captured by the imager

Table 3.9: Summary of comparison of anomaly detectors.

Other metrics are also important to consider in order to choose the best AD for implementation in HW. The ACAD and the ALRX algorithms are beneficial with regards to data transmission requirements as they build a binary anomaly map of size $N_{pixels} \times N_{rows}$ which may be transmitted, as opposed to the RX and LRX algorithms which produce output results of size $Pixel\_data\_width \times 2 \times N_{pixels} \times N_{rows}$. If the result from the AD is to be transmitted via radio to a ground station, this will become an important consideration as the transmission layer is usually the most power-hungry layer in wireless sensor nodes (WSN) [11]. One example of this is shown in Figure 3.14 [11]:

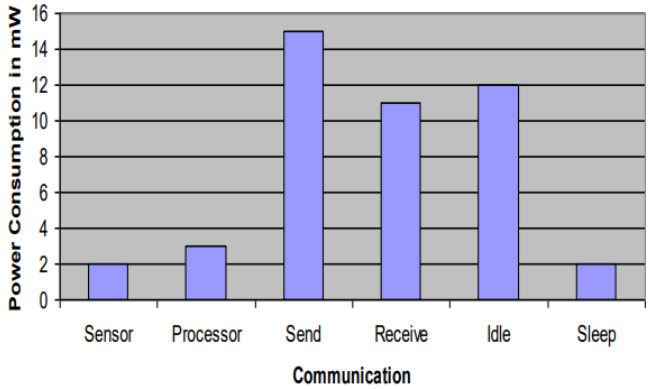

Figure 3.14: Power consumption in a WSN [11].

In Figure 3.14, the different states of operation of the WSN and the operation state power consumption are shown. The two states "Send" and "Receive" consume most and third-most power respectively. These states are part of the transmission layer in a WSN. A satellite may also be considered as a WSN as it is wireless and connected via a radio link to a ground station.

The ACAD algorithm's causality and the use of correlation matrix makes it easier to process in real-time compared to the LRX and GRX algorithms. ACAD can immediately start processing when the first pixel is captured. RX, utilizing the global covariance matrix, can not start computing the global covariance matrix until the entire image is captured by the hyperspectral imager. LRX needs an image tile of size $K \times K$ to be captured by the imager before computation of the local correlation matrix can start.

Chan *et al.* [7] conclude that ACAD has advantages over the RX AD in several ways. It can be processed in real time. "It detects various anomalies regardless of whether they are of the same type or distinct types" [7]. The findings in [7] and the above comparison made by the author lead to the ACAD algorithm being chosen as the AD for implementation in HW.

Chapter 4

Proposed hardware implementation

The following chapter describe the proposed implementation of the Adaptive Causal anomaly detection (ACAD) algorithm on a Zynq Z-7030 or Z-7035 device, used by the NTNU Smallsat project. The fact that the initial prototype of the ACAD algorithm is to implemented on the Zedboard and Development kit (referred to as Zedboard) is taken into account.

4.1 Memory considerations

As the ACAD AD is to be implemented on a Zynq Z-7030 or Z-7035 device, care must be taken in the design process regarding the logic and memory usage. The hyperspectral pixel data inputted to the AD might have number of spectral bands, $P\_bands = N_{BANDS}$, depending upon if preprocessing steps such as Principal Component Analysis (PCA) is done on the image cube. $Pixel\_data\_width$ is the data width per spectral band of the input pixel to the AD. This will be up to 16 bit in the SmallSat project. The large size of P_bands and $Pixel\_data\_width$ make memory usage an important consideration.

4.1.1 Storing and updating matrices in ACAD

The ACAD algorithm requires storage of the following matrices: $\mathbf{R}(\mathbf{x}_k)$, $\tilde{\mathbf{R}}(\mathbf{x}_k)$ and $\sum_{t_j \in \Delta(k)} \mathbf{t}_j \mathbf{t}_j^T$. Additionally, the matrices A and A^{-1} used in Gauss-Jordan elimination as shown in Figure 4.1 must be stored in memory. These matrices have a size of $P\_bands \times P\_bands$, with matrix elements of size $Pixel\_data\_width \times 2$.

A[row[j]] = A[row[j]] − A[row[i]] ∗ (A[row[j]][i]/
A[row[i]][i]);
$\mathbf{A}^{-1}[row[j]] = \mathbf{A}^{-1}[row[j]] - \mathbf{A}^{-1}[row[i]] * (\mathrm{A}[\mathrm{row}[j]][i]/$
A[row[i]][i]); // This operation is done in parallel with the

Figure 4.1: Matrix A and A^{-1} used in Gauss-Jordan elimination.

.

Storing and updating matrices of size $P\_bands \times P\_bands$ with matrix elements of size $Pixel\_data\_width \times 2$ require a lot of memory resources. One of the matrices is the causal correlation matrix $\mathbf{R}(\mathbf{x}_k)$. Update of this matrix needs to be done for each pixel in the image, and the memory used for this operation is therefore important in order to make the AD real-time. As $\mathbf{R}(\mathbf{x}_k)$ is the product of $\mathbf{x} \times \mathbf{x}^T$, the resulting data width will be $2 \times Pixel\_data\_width$. For $Pixel\_data\_width = 16$, using spectral information from all N_{BANDS} would require $P\_bands \times P\_bands \times 32 = 100 \times 100 \times 32$ bit $= 320kbit$ of memory storage.

There exist two alternatives for storage of all this information on the Field-Programmable Gate Array (FPGA) in the Zynq device: storing it in block RAM (BRAM) or in registers. The initial prototype for the SmallSat project will be developed on the Zedboard. Later stage prototypes will contain the Zynq Z-7030 or the Zynq Z-7035. The FPGAs contain the memory resources as shown in Figure 4.2. The Z-7030 and the Z-7035 contain 265 and 500 36kbit BRAM blocks respectively. The number of DSP Slices is 400 for the Z-7030 and 900 for the Z-7035.

Table 1: **Zynq-7000 and Zynq-7000S All Programmable SoCs *(Cont'd)***

	Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100
	Part Number	XC7Z007S	XC7Z012S	XC7Z014S	XC7Z010	XC7Z015	XC7Z020	XC7Z030	XC7Z035	XC7Z045	XC7Z100
Programmable Logic	Xilinx 7 Series Programmable Logic Equivalent	Artix®-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Kintex®-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA
	Programmable Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200	78,600	171,900	218,600	277,400
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400	157,200	343,800	437,200	554,800
	Block RAM (# 36 Kb Blocks)	1.8 Mb (50)	2.5 Mb (72)	3.8 Mb (107)	2.1 Mb (60)	3.3 Mb (95)	4.9 Mb (140)	9.3 Mb (265)	17.6 Mb (500)	19.2 Mb (545)	26.5 Mb (755)
	DSP Slices (18x25 MACCs)	66	120	170	80	160	220	400	900	900	2,020
	Peak DSP Performance (Symmetric FIR)	73 GMACs	131 GMACs	187 GMACs	100 GMACs	200 GMACs	276 GMACs	593 GMACs	1,334 GMACs	1,334 GMACs	2,622 GMACs

Figure 4.2: Zynq memory resources [12].

.

4.1.1.1 Using registers

The Zynq Z-7030 and the Zynq Z-7035 contain 157, 200 and 343, 800 flip flops (registers), respectively. By using equation 4.1:

$$max\_bands = \sqrt{\frac{number\_of\_registers}{2 \times Pixel\_data\_width \times number\_of\_matrices}}, \tag{4.1}$$

it is possible to do an estimation of the maximum value of P_bands if using registers for storage of the matrices. Parameter $number\_of\_registers$ is the total number of registers (flip flops) available in the device. $number\_of\_matrices$ is the number of matrices of size $P\_bands \times P\_bands$ with matrix elements of size $Pixel\_data\_width \times 2$ that need to be stored in memory. max_bands is the maximum number of P_bands for the matrices used in ACAD. As the ACAD algorithm needs to store five matrices, $number\_of\_matrices = 5$, which yields $max\_bands = [31, 46]$ for the Z-7030 and Z-7035 respectively.

However, using this amount of registers is unrealistic as it leaves no registers free for other use in the design. As the AD implemented in this thesis is a part of a larger processing pipeline, it is not acceptable to use all of the available registers. Assuming that it is acceptable to use 15% of the available registers, the number of spectral bands that can be used is 12 and 17 for the Zynq Z-7030 and the Zynq Z-7035 respectively. Dimensional reduction to reduce P_bands from 100 to 12 or 17 can be done through pre-prossesing of the data by for example PCA. The benefit of using registers to store matrices is the ability of instantaneous update.

4.1.1.2 Using BRAM

The Z-7030 and the Z-7035 contain 265 and 500 36kbit BRAM - blocks respectively. In order to store the largest matrix of $320kbit$, a minimum of nine BRAM blocks are needed. Each 36 kbit BRAM block consists of two 18 kbit BRAM blocks. In true dual port (TDP) mode [23], it is possible to do two writes and two reads per 36kbit BRAM per clock cycle, with each write and read being maximum 36 bits. BRAMs in TDP mode have only one address input, the same address for reads and writes. This makes it hard to use for the correlation module as the ACAD correlation needs to read previously stored data from the BRAM before writing to the same address. Therefore, it is necessary to have a separate read and write address. By inferring two separate Simple Dual Port (SDP) 18kbit BRAMs, by the code shown in Listing 4.1, it is possible to get two writes and two reads per cycle per 36kbit BRAM, with separate read and write addresses.

Listing 4.1: Code for inferring a SDP 18 kbit BRAM.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity block_ram is
  generic(
    B_RAM_SIZE      : integer := 100;
    B_RAM_BIT_WIDTH : integer := 32
    );
  port (
    clk           : in  std_logic;
    aresetn       : in  std_logic;
    data_in       : in  std_logic_vector(B_RAM_BIT_WIDTH-1 downto 0);
    write_enable  : in  std_logic;
    read_enable   : in  std_logic;
    read_address  : in  integer range 0 to B_RAM_SIZE-1;
    write_address : in  integer range 0 to B_RAM_SIZE -1;  -- added
    data_out      : out std_logic_vector(B_RAM_BIT_WIDTH-1 downto 0)
    );
end block_ram;
architecture Behavioral of block_ram is
```

```
  type bus_array is array(0 to B_RAM_SIZE-1) of
                        std_logic_vector(B_RAM_BIT_WIDTH-1 downto 0);
  signal b_ram_data : bus_array;
begin
  process(clk)
  begin
    if(rising_edge(clk)) then
      if(write_enable = '1') then
        b_ram_data(write_address) <= data_in;
      end if;
    end if;
  end process;
  process(clk)
  begin
    if(rising_edge(clk)) then
      if(read_enable = '1') then
        data_out <= b_ram_data(read_address);
      end if;
    end if;

  end process;
end Behavioral;
```

To be able to evaluate if it is possible to store a matrix of size $P\_bands \times P\_bands$ with matrix elements of size $Pixel\_data\_width \times 2$ in BRAM, with acceptable update characteristics, the time spent updating $\tilde{\mathbf{R}}(\mathbf{x}_k)$ has been used as a benchmark. Equation 4.2 shows the calculation of number of clock cycles needed to update $\tilde{\mathbf{R}}(\mathbf{x}_k)$ of size $P\_bands \times P\_bands$, $n\_clk\_update\_corr\_BRAM$:

$$n\_clk\_update\_corr\_BRAM = \frac{P\_bands \times P\_bands}{2 \times N\_bram\_correlation}. \tag{4.2}$$

The update is done for each pixel in the image. $N\_bram\_correlation$ is the number of 36 kbit BRAMs used to store $\tilde{\mathbf{R}}(\mathbf{x}_k)$. The total time spent updating $\tilde{\mathbf{R}}(\mathbf{x}_k)$ for the entire image is given by equation 4.3:

$$clk\_corr\_image\_BRAM = N\_pixels \times N\_rows \times n\_clk\_update\_corr\_BRAM. \tag{4.3}$$

Updating a matrix with $P\_bands = 100$ using nine BRAMs would require 556 clock cycles. For the entire image, having $N\_pixels = 578$ and $N\_rows = 1088$, the total amount of clock cycles spent updating the correlation matrix would be 349,648,384. At a target clock frequency of 100 MHz this would require 3.49648 seconds.

Figure 4.3 shows the estimated total time spent updating $\tilde{\mathbf{R}}(\mathbf{x}_k)$ for an image of N_rows =1088 $\times$ N_pixels = 578, with a target clock frequency of 100 MHz, plotted as a function of $N\_bram\_correlation$. The BRAMs are assumed written to in parallel. Figure 4.3 is plotted for P_bands =[20, 30, 40, 50, 60, 70, 80, 90, 100].

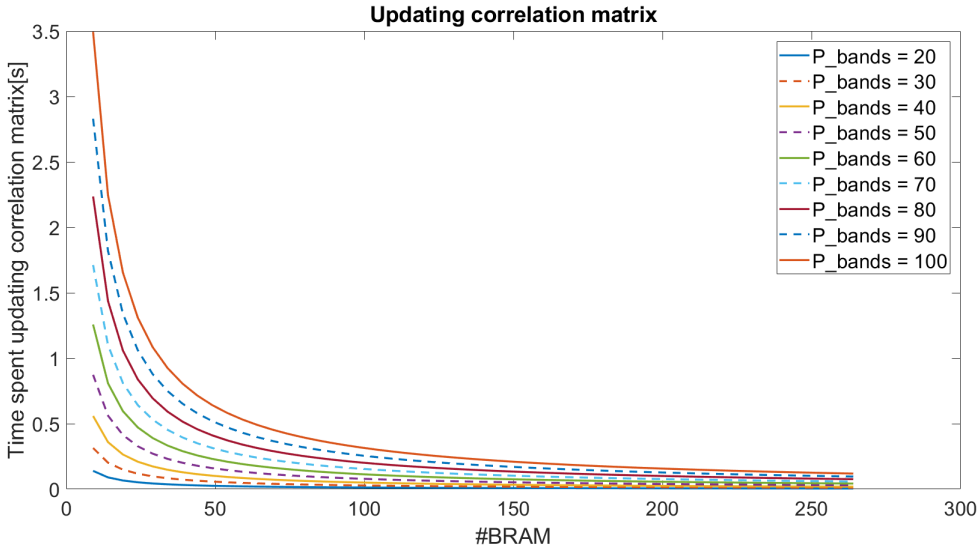

Figure 4.3: Estimated time spent updating $\tilde{\mathbf{R}}(\mathbf{x}_k)$.

As shown in Figure 4.3, the time spent updating $\tilde{\mathbf{R}}(\mathbf{x}_k)$ can be reduced by increasing $N\_bram\_correlation$. One column of $\tilde{\mathbf{R}}(\mathbf{x}_k)$ will maximum contain $100 \times 32 = 3200$ bit of data. By setting $N\_bram\_correlation = P\_bands$ it is possible to store one column of the correlation matrix in each BRAM, and enable writing of P_bands number of columns at the same time. As it is possible to write two 32 bit elements per cycle for each 36 kbit BRAM block, the total correlation matrix update time per pixel is $\frac{P\_bands}{2}$. Storing one column in each 36 kbit BRAM block simplifies the control logic while achieving an acceptable trade-off between speedup as a function of the number of BRAMs used and resources used.

In order for each of the inner-most for-loops in Figure 2.10 to be executed within one clock cycle, it is possible to determine the maximum memory requirements of the Gauss-Jordan elimination. The maximum memory requirement of the Gauss-Jordan elimination is when executing the operations showed in Figure 4.4.

row $[i]$ = row$[j]$;
row$[j]$ = row$[i]$; // This operation is done in parallel with the previous one

Figure 4.4: Maximum memory accessing requirement by the Gauss-Jordan elimination.

The operations shown in Figure 4.4 need to swap two rows of A. By choosing to have P_bands number of 36 kbit BRAMs for storage of A and A^{-1} it is possible to write and read two rows of each matrix per clock cycle, and thereby execute the operations shown in Figure 4.4 in one clock cycle.

The Zynq-Z7030 and the Zynq-Z7035 contain 265 and 500 BRAM blocks respectively. By utilizing P_bands to store each of the five matrices, this means maximum P_bands will be 53 for the Z7030 and 100 for the Z7035.

$\tilde{\mathbf{R}}(\mathbf{x}_k)$ is written to $N\_bram\_correlation$ in parallel, where the leftmost column (column zero) of $\tilde{\mathbf{R}}(\mathbf{x}_k)$ is written to BRAM_0, column one to BRAM_1, ... and column P_bands-1 to BRAM_P_bands-1. For each 36 kbit BRAM, two 18kbit BRAM blocks are accessed, one for even row indices of the column and one for odd row indices of the column. In Figure 4.5, the addressing scheme for each 36kbit BRAM is presented, exemplified by BRAM_0 and BRAM_P_bands-1. As shown in the figure, elements of column zero are stored in BRAM_0, while elements of column P_BANDS-1 are stored in BRAM_P_bands-1.

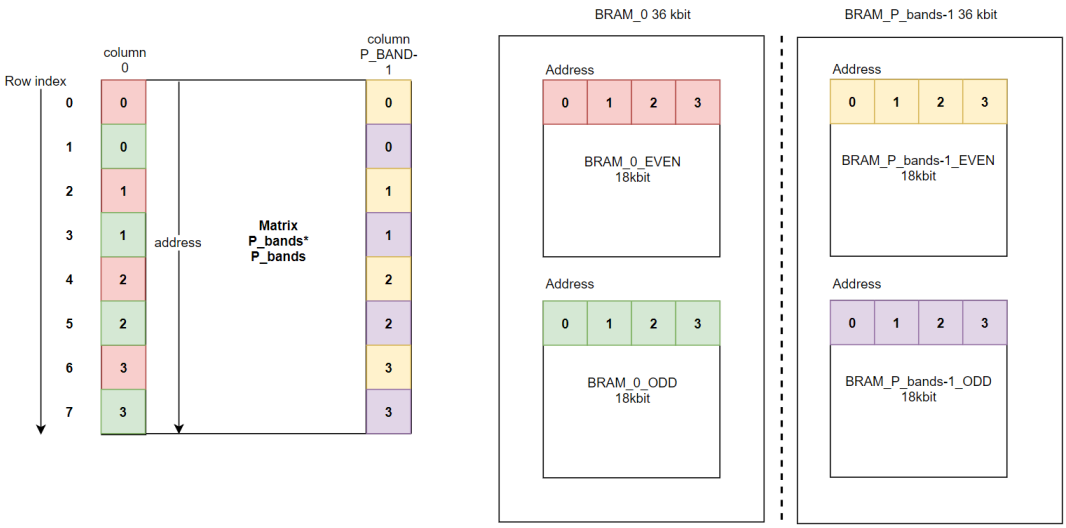

Figure 4.5: BRAM addressing scheme for storage of matrices utilized by ACAD. One column of the matrix is stored per 36kbit BRAM.

By using the same addressing logic for $\mathbf{R}(\mathbf{x}_k)$, $\sum_{t_j \in \Delta(k)} \mathbf{t}_j \mathbf{t}_j^T$, A and A^{-1} as for $\tilde{\mathbf{R}}(\mathbf{x}_k)$, the control logic of the design is simplified.

Figure 4.6 shows a 36 kbit BRAM block used in the design and the dataflow within.

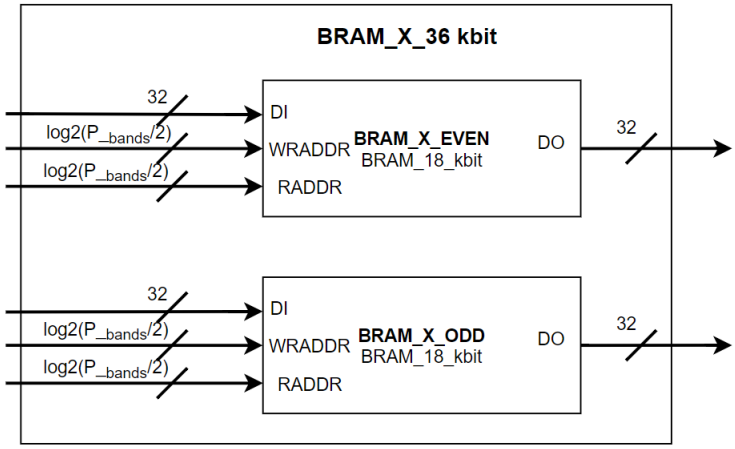

Figure 4.6: BRAM hierarchy, showing two 18kbit BRAM blocks contained within one 36kbit BRAM block.

BRAM_X_EVEN contains even row index elements for column X. **BRAM_X_ODD** contains odd row index elements of the same column. The width of the addresses $WRADDR$ and $RADDR$ will be $log_2(\frac{P\_bands}{2})$. This is because a maximum of $\frac{P\_bands}{2}$ addresses is needed as one 36kbit BRAM contains two 18kbit BRAM blocks, storing even and odd indexes of the column as shown in Figure 4.5. This means half the column is stored in the **BRAM_X_EVEN** and the other half in the **BRAM_X_ODD**.

4.2 Proposed implementation

The top level architecture of the ACAD Anomaly detector is presented in Figure 4.7. It consists of five blocks: **FSM ACAD**, **Shiftregister**, **ACAD correlation**, **ACAD inverse** and **dACAD module**. The matrices $\mathbf{R}(\mathbf{x}_k)$, $\sum_{t_j \in \Delta(k)} \mathbf{t}_j \mathbf{t}_j^T$, A, A^{-1} and $\tilde{\mathbf{R}}(\mathbf{x}_k)$ are all stored in BRAM. The ACAD anomaly detector interfaces the Cube DMA [16] via an AXI-Stream interface. The output of the ACAD anomaly detector, *anomaly map*, is a binary anomaly map.

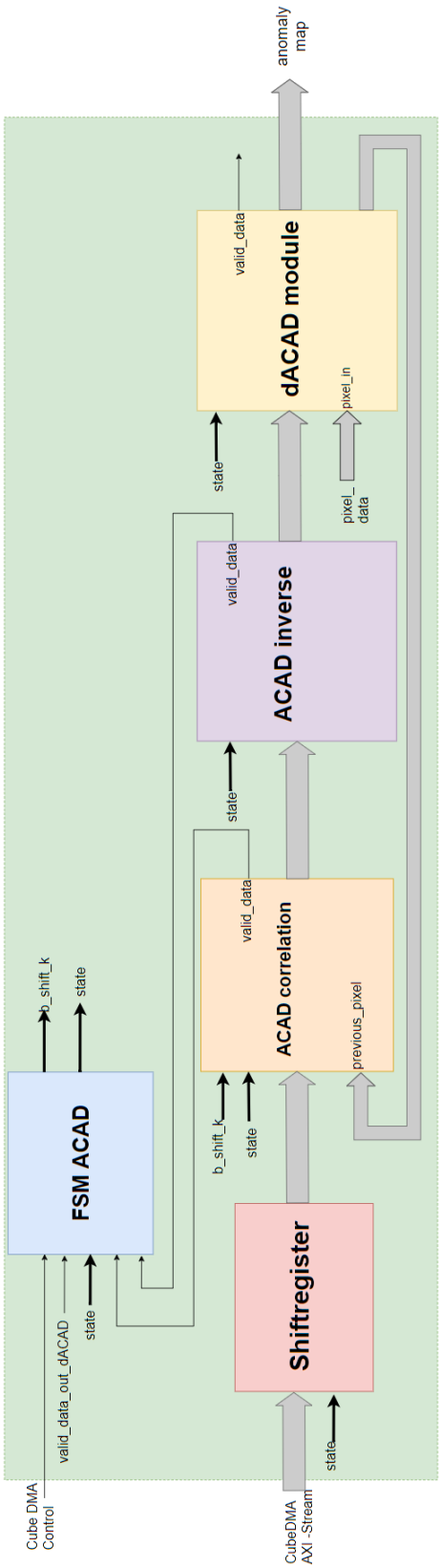

Figure 4.7: Top level architecture of the ACAD anomaly detector.

The **Shiftregister** is a Serial-in Parallel-Out (SIPO) shiftregister which interfaces the SmallSat's Cube DMA. It is AXI-Stream compatible. 64-bit data is shifted in per clock cycle. For $Pixel\_data\_width$ of 16, four bands are shifted in per cycle. When a complete pixel is shifted in, it is sent to **ACAD correlation**, which computes the matrix $\tilde{\mathbf{R}}(\mathbf{x}_k)$. After $\tilde{\mathbf{R}}(\mathbf{x}_k)$ is computed, it is sent to **ACAD inverse**. Two rows of $\tilde{\mathbf{R}}(\mathbf{x}_k)$ are outputted per clock cycle to **ACAD inverse**, which computes $\tilde{\mathbf{R}}^{-1}(\mathbf{x}_k)$.

Two rows of $\tilde{\mathbf{R}}^{-1}(\mathbf{x}_k)$ are then sent to the **dACAD module** per clock cycle. This block computes δ^{ACAD}, as given by:

$$\delta^{ACAD}(\mathbf{x}_k) = \mathbf{x}_k^T \tilde{\mathbf{R}}^{-1}(\mathbf{x}_k)\mathbf{x}_k. \tag{4.4}$$

u_k and t_k are calculated in this block to decide if the pixel is anomalous. A binary anomaly map is created. When all pixels have been processed, the generated anomaly map is outputted and the ACAD anomaly detection is finished.

The **FSM_ACAD** block controls the state of the ACAD anomaly detector. It chooses which of the blocks should have its clock enabled and controls the general behaviour of the anomaly detector.

4.3 Shiftregister

The width of the the data input to **Shiftregister** is 64, while the width of the output is $P\_bands \times Pixel\_data\_width \times 2$. **Shiftregister** is designed to function for P_bands dividable by four, i.e. the remainder of $modulo(\frac{P\_bands}{4})$ is zero. Figure 4.8 shows the architecture of **Shiftregister**, while the output of **Shiftregister** is shown in Figure 4.9.

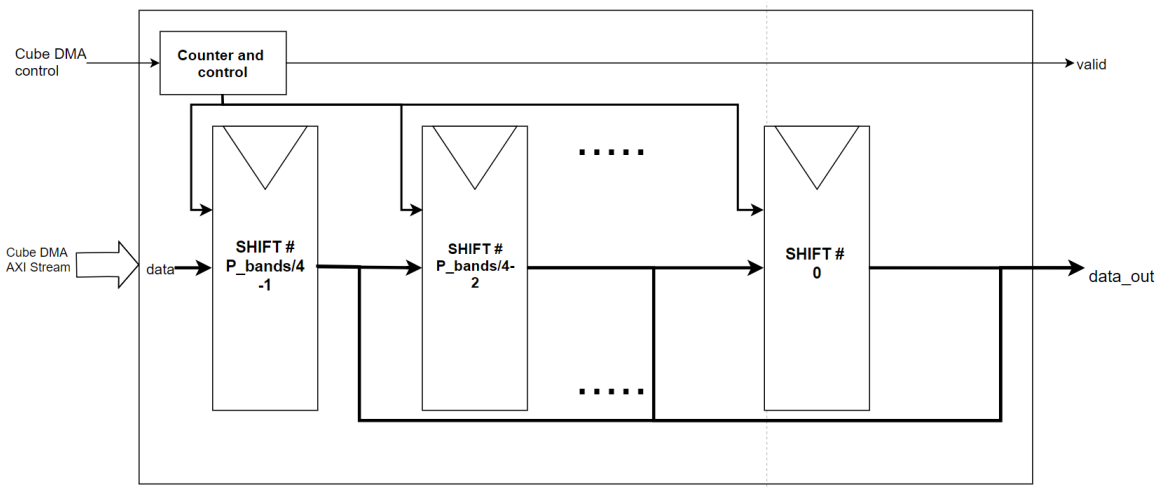

Figure 4.8: Architecture of the **Shiftregister** block.

The depth of **Shiftregister** is $\frac{P\_bands}{4}$. The **Counter and control** block in Figure 4.8 controls the shiftregister. It counts the number of shifts executed with the counter

count_number_of_shifts. For each clock cycle that valid data is inputted from the Cube DMA, *count_number_of_shifts* increments. When *P_bands* shift operations have been executed, *valid* is asserted and *data_out* is sent to **ACAD correlation**. *data_out* will be a $P\_bands \times Pixel\_data\_width$ wide signal. The signal *data_out* is shown in Figure 4.9 for $Pixel\_data\_width = 16$.

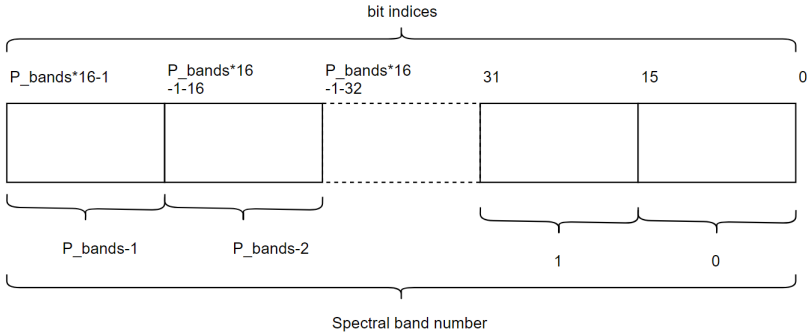

Figure 4.9: Data output of the **Shiftregister** block.

4.4 ACAD correlation

The **ACAD correlation**, as shown in Figure 4.7, computes the causal correlation matrix:

$$\mathbf{R}(\mathbf{x}_k) = \frac{1}{k}\sum_{i=1}^{k}\mathbf{x}_i\mathbf{x}_i^T. \tag{4.5}$$

ACAD correlation is designed for an even number of *P_bands*. If an odd number of *P_bands* is used, a band with zero values has to be inserted or the matrix needs to be re-scaled to an even number of *P_bands* before inputting to the **ACAD correlation**.

Data flow and architecture of **ACAD correlation** can be seen in Figure 4.10.

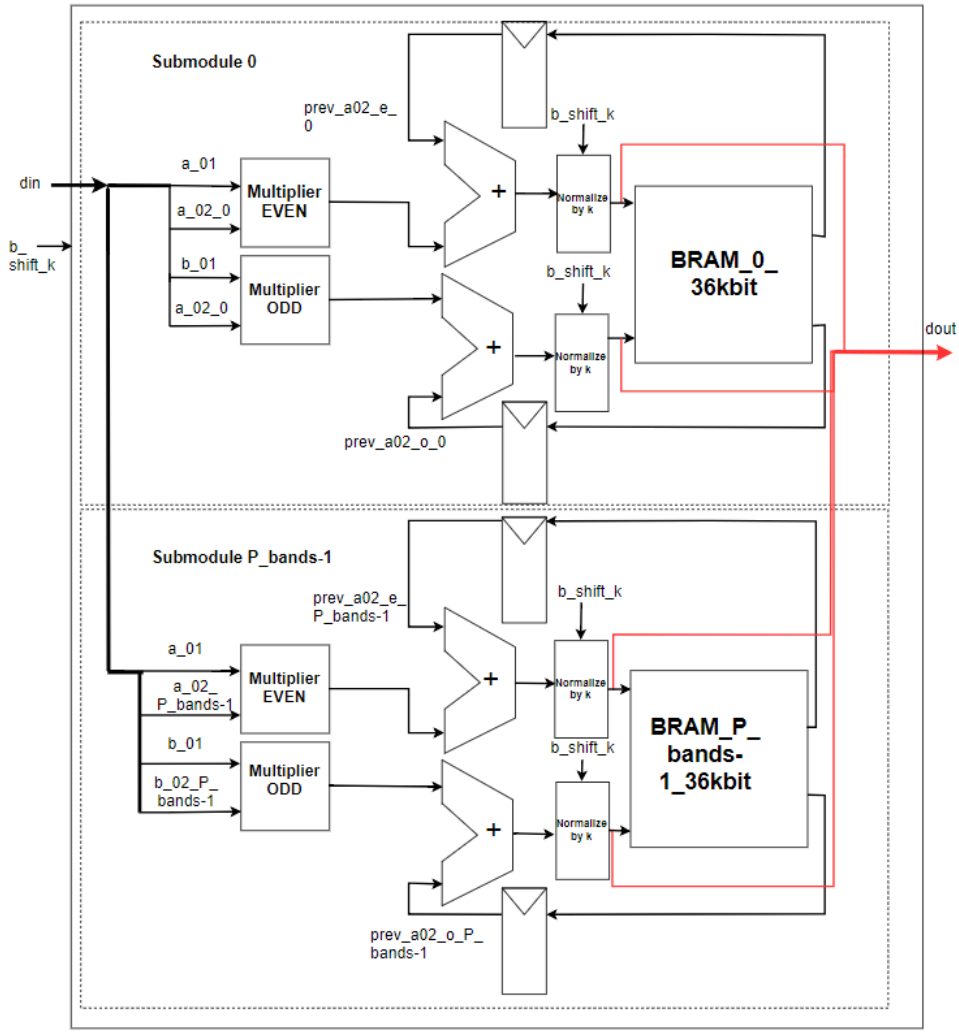

Figure 4.10: Data flow within the **ACAD correlation** module.

The dotted squares mark one correlation sub-module. One correlation sub-module computes two elements of $\tilde{\mathbf{R}}(\mathbf{x}_k)$. P_bands correlation sub-modules are synthesized in the design. Input signal *din* is a $P\_bands \times Pixel\_data\_width$ wide bus. It is sent from the **Shiftregister** block and contains data from one pixel vector. Signal b_shift_k is sent from **FSM ACAD** and indicates the number of shift operations to be done by the **Normalize by k** blocks.

The elements from the previously computed $\tilde{\mathbf{R}}(\mathbf{x}_{k-1})$, $prev\_a02\_e/o\_x$, are stored in BRAMs. In the signals $prev\_a02\_e/o\_x$, e/o marks that it is for an even or odd element, and x is the column index of the element computed. $prev\_a02\_e/o\_x$ are added to the output of the multipliers before being normalized by the block **Normalize by k**.

The block **Normalize by k** approximates the operation of multiplying the result after addition with $\frac{1}{k}$, where k is the index of the currently processed pixel. This block performs a right hand shift operation, dependent upon k. The architecture and functionality of **Normalize by k** is further described in section 4.4.1. The outputs of the **Normalize by k** blocks drive the output signal *dout* and the data inputs of the BRAMs. Signal *dout*, marked by red, is a $P\_bands \times Pixel\_data\_width \times 2 \times 2$ wide bus containing two rows of $\tilde{\mathbf{R}}(\mathbf{x}_k)$.

Figure 4.11 shows an example of the operations done by the **ACAD correlation** block, and how the results are stored in BRAMs. In this example, *din* is a spectral pixel vector with $P\_bands = 4$. $din.'$ is the transposed vector of *din*.

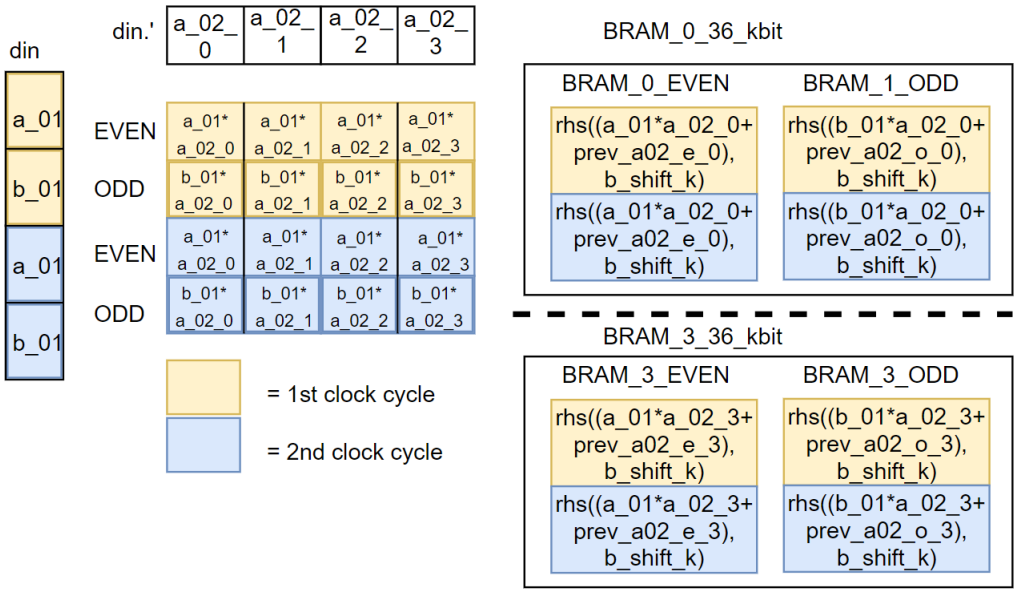

Figure 4.11: An example of the data handling done by **ACAD correlation**. For this example P_bands =4.

The yellow blocks in Figure 4.11 are blocks that are executed, utilized or written to within the first clock cycle. In the first clock cycle, a_01 and b_01 are set to the first and second element of the input signal *din*. These elements have a width of $Pixel\_data\_width$. a_01 in this cycle is $din[Pixel\_data\_width$-1:0] and b_01 is $din[Pixel\_data\_width \times 2$ -1 :$Pixel\_data\_width]$. Each of the sub-modules shown in Figure 4.10 computes two elements of $\tilde{\mathbf{R}}(\mathbf{x}_k)$ before inputting the results to BRAMs. Sub-module 0 computes column zero, which is written to BRAM_0_36kbit, sub-module 1 computes column one, which is written to BRAM_1_36_kbit, ..., while sub-module P_bands-1_36kbit computes column P_bands-1, which is written to $BRAM\_P\_bands$-1_36kbit. Figure 4.11 illustrates these operations. The *rhs* operation is a right shift operation by b_shift_k spaces.

The blue blocks in Figure 4.11 are blocks that are executed, utilized or written to within the second clock cycle. a_01 in this cycle is $din[Pixel\_data\_width \times 3$-1: $Pixel\_data\_width \times 2]$ and b_01 is $din[Pixel\_data\_width \times 4$ -1 : $Pixel\_data\_width$

$\times$ 3].

4.4.1 Normalizing with k

The **Normalizing with k** block approximates the operation of multiplying $\sum_{i=1}^{k} \mathbf{x}_i \mathbf{x}_i^T$ with $\frac{1}{k}$, where k =[1,2,3...,N_PIXELS_TOT] is the index of the pixel currently being processed. N_PIXELS_TOT is the total amount of pixels in the hyperspectral image, $N\_PIXELS\_TOT = N_{rows} \times N_{pixels}$. Due to division being an operation that is computationally intensive, a shifting approach is proposed instead of doing actual division by utilizing the division operator "/".

Doing shifting instead of actual division will lead to a precision error for k that is not power of two.

The signal b_shift_k shown in Figure 4.10 is the number of shifts that best approximates the division $\frac{1}{k}$. The signal is driven by **FSM ACAD** and inputted to **ACAD correlation**.

The author propose to store an array called $b\_shift\_k\_array(k)$ in LUTs. $b\_shift\_k\_array(k)$ contains the best shifting approximation for pixel indexes k =[1,2,3... N_PIXELS_TOT]. The array can be generated by the MATLAB-script shown in Listing 4.2.

Listing 4.2: Code for creating the array b_shift_k_array.

```
clear; clc;

N_pixels = 578;
N_rows = 1088;
N_PIXELS_TOT = N_pixels*N_rows;
b_shift_k_array = zeros(1,N_PIXELS_TOT);
file = fopen('b_shift_array.txt', 'w');


for k =1: N_PIXELS_TOT
    if mod(k,2) == 0
        % k is power of two
        b_shift_k_array(k) = log2(k); % Number of shifts is log2(k)
    else
        b_shift_k_array(k) = ceil(log2(pow2(floor(log2(k)))));
    end
    fprintf(file, '%d,\n',round(b_shift_k_array(k)));
end

fprintf('\n');
fclose(file);
```

4.5 Inverse computation

Due to its low complexity, the Gauss-Jordan elimination was chosen for implementation of inverse matrix computation.

The top level architecture of **ACAD inverse** is presented in Figure 4.12. This is an implementation of the Gauss-Jordan elimination shown in Figure 2.10. **ACAD inverse** interfaces **ACAD correlation**. The outputs from **ACAD inverse** are sent to **dACAD**.

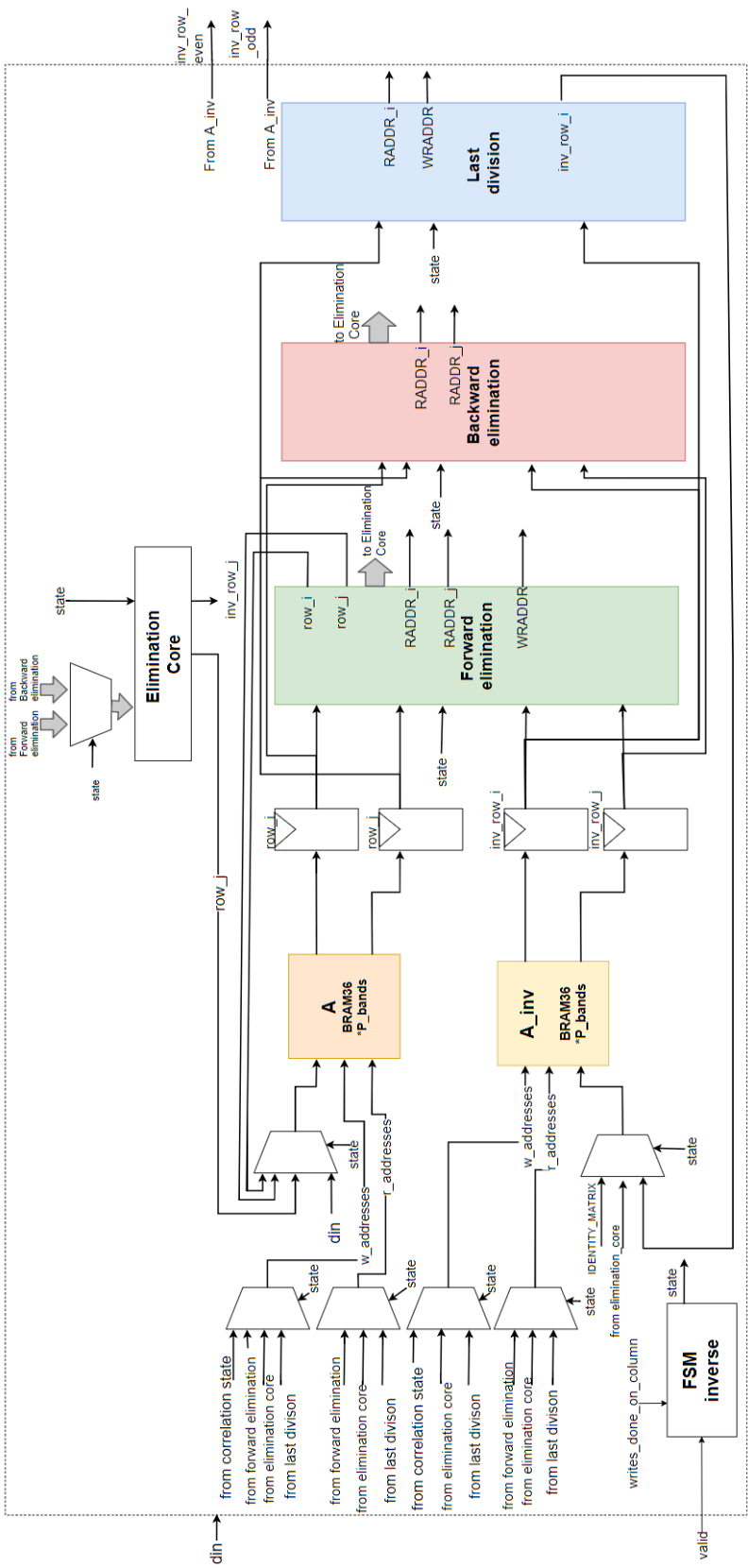

Figure 4.12: Top level architecture of the inverse module.

The **Forward elimination**, **Backward elimination** and **Last division** execute the operations done in the forward elimination, backward elimination and last division part of Gauss-Jordan elimination as described in Section 2.4.1, with an exception to the operations shown in Figure 4.13. These operations are part of both the forward elimination and backward elimination blocks in the Gauss-Jordan inverse. They are therefore put in an external process, called **Elimination core**. A and A_inv are two arrays of BRAM36 of size P_bands, in which $\mathbf{A}$ and $\mathbf{A}^{-1}$ are stored.

$$\begin{aligned}&\mathrm{A[row}[j]] = \mathrm{A[row}[j]] - \mathrm{A[row}[i]] * (\mathrm{A[row}[j]][i]/\\ &\mathrm{A[row}[i]][i]);\\ &\mathbf{A}^{-1}[row[j]] = \mathbf{A}^{-1}[row[j]] - \mathbf{A}^{-1}[row[i]] * (\mathrm{A[row}[j]][i]/\\ &\mathrm{A[row}[i]][i]);\ \text{// This operation is done in parallel with the previous one}\end{aligned}$$

Figure 4.13: The operations computed by the **Elimination core**, utilized by both the **Forward elimination** and the **Backward elimination** block.

4.5.1 Elimination core

Elimination core is utilized by both **Forward elimination** and **Backward elimination**. Its input are driven by **Forward elimination** or **Backward elimination**, depending upon which of the states in **FSM inverse** that are active (states are presented in Section 4.5.2). **Elimination core** does not use the division operator "/" to compute division, but rather use a combination of the LUT approach and the adaptive-shifting approach. See Section 4.5.9 for more details about these approximations to division.

The architecture of **Elimination core** can be seen in Figure 4.14 and Figure 4.15. The input signals $control\_BRAM$, row_j, row_i, $index\_i$, $index\_j$ and $state$ are driven by **Forward elimination** or **Backward elimination**. The signal $divisor\_inv$ is the inverse of the divisor used in the operations shown in Figure 4.13. $best\_approx$ is the best adaptive-shifting approximation to the divisor. The division is computed using $divisor\_inv$ if $Div\_Precision >=$ MSB of the divisor (signal msb_index in Figure 4.14). The blocks **rhs(DIV_PRECISION)** and **rhs(best_approx)** perform a right shift operation by $Div\_Precision$ and $best\_approx$ spaces respectively. If $Div\_Precision <$ MSB of the divisor, the adaptive shifting approach is utilized to approximate division.

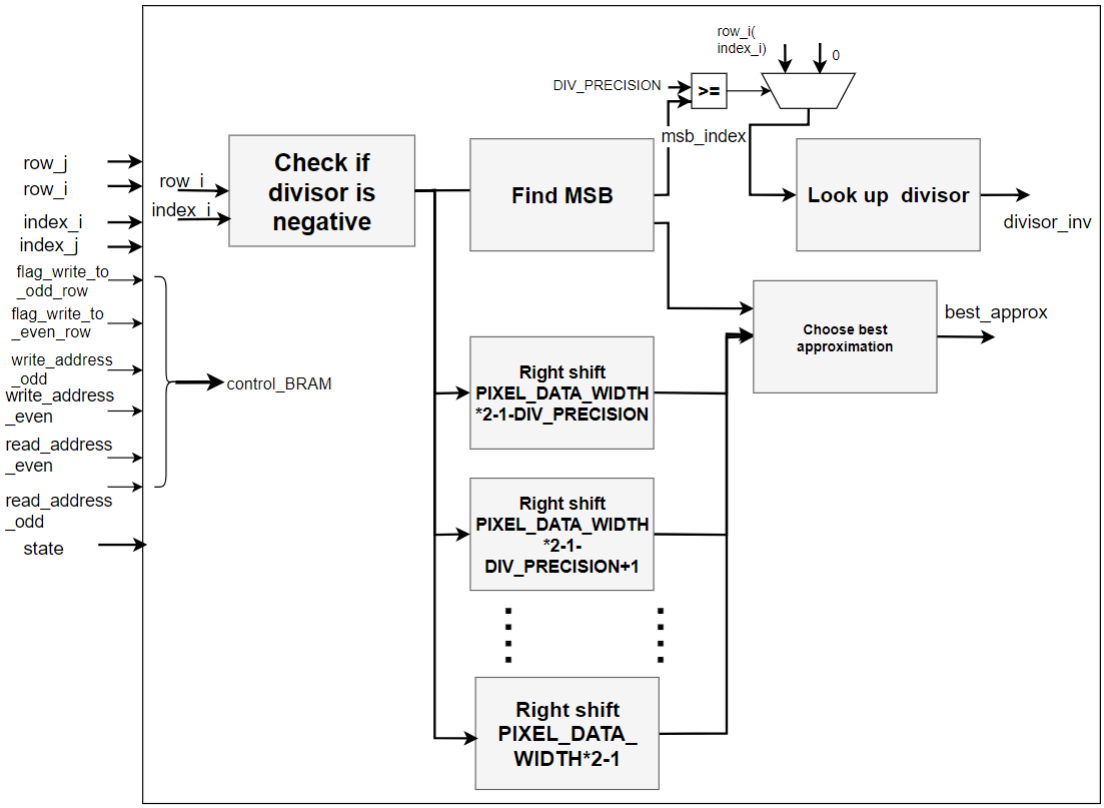

Figure 4.14: Elimination core part one.

The output signal *data_out* consists of new matrix data for matrices A and A^{-1}, stored in **A** and **A_inv**, for *index_j*. The bus *control_BRAM* contains control signals to **A** and **A_inv**. The control signals include read and write addresses and write and read-enabling signals. **Elimination core** contains *P_bands* sub-modules marked by the dotted squares in Figure 4.15. One such sub-module computes one element of *new_row_j* and one element of *new_inv_row_j*. *new_row_j* and *new_inv_row_j* are updated row data for the rows indexed by index j in matrices A and A^{-1}, which are written to **A** and **A_inv** respectively. As shown in Figure 4.15, Submodule(0) computes *new_row_j*(0) and *new_inv_row_j*(0), Submodule(1) computes *new_row_j*(1) and *new_inv_row_j*(1),..., and Submodule(*P_bands*−1) computes *new_row_j*(*P_bands*−1) and *new_inv_row_j*(*P_bands* − 1). As such, the operations shown in Figure 4.13 are implemented.

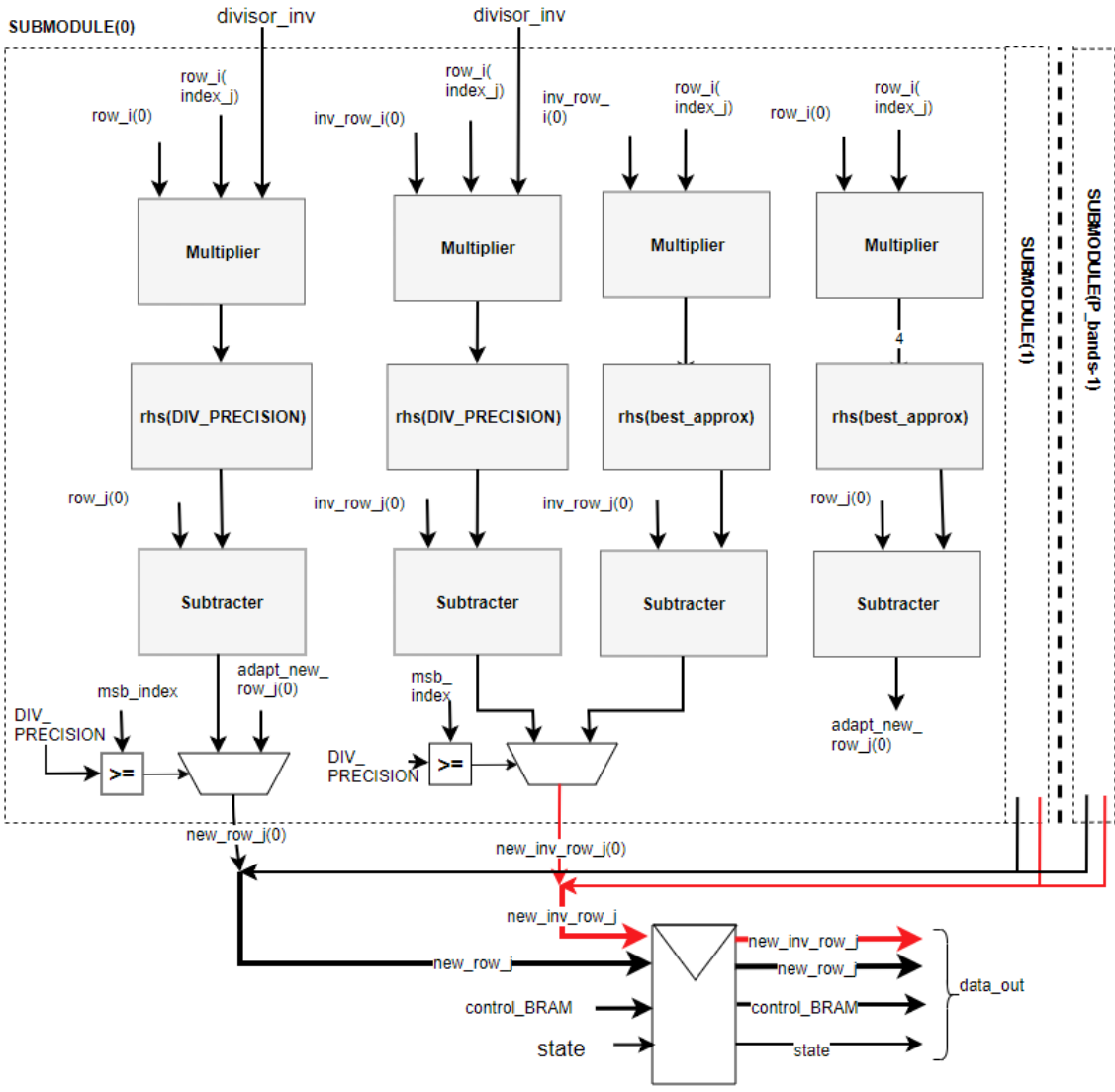

Figure 4.15: Elimination core part two.

4.5.2 FSM inverse

The finite state machine (FSM) for the **ACAD inverse** shown in Figure 4.12 is illustrated in Figure 4.16. Its possible states are described in Table 4.1.

State	Description
Unknown state	An unknown state. The behaviour of the **ACAD inverse** is unknown. The FSM should transition to state **Idle**.
Idle	The **ACAD inverse** is not performing any operations.
Store_correlation_matrix_in_BRAM	Writing data inputted from **ACAD correlation** to **A**. Two rows are written to BRAMs per clock cycle.
Forward_elimination	Computing the forward elimination of the Gauss-Jordan elimination.
Backward_elimination	Computing the backward elimination of the Gauss-Jordan elimination.
Last_division	Computing the last division of the Gauss-Jordan elimination.
Output_inverse_matrix	Outputting the completed inverse matrix for the pixel. Two rows are outputted per clock cycle.

Table 4.1: States of the inverse FSM.

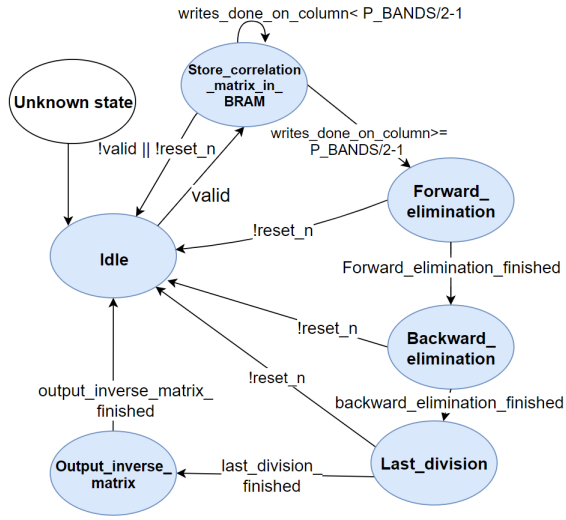

Figure 4.16: FSM controlling **ACAD inverse** shown in Figure 4.12.

In state **Idle**, **ACAD inverse** is not performing any operations. The outputs of **ACAD inverse** are not valid in this state. All states transition to **Idle** if signal *reset_n* is asserted.

When valid data is written to **ACAD inverse** from **ACAD correlation**, signal *valid* is asserted, and the FSM transitions from **Idle** to **Store_correlation_matrix_in_BRAM**. **Store_correlation_matrix_in_BRAM** stores two rows per clock cycle of the causal anomaly-removed sample spectral correlation matrix outputted from **ACAD correlation** in BRAMs **A**. It also writes two rows per clock cycle of the identity matrix of size $P\_bands \times P\_bands$ with matrix elements of size $Pixel\_data\_width \times 2$ to **A_inv**.

write_done_on_column is a signal indicating the number of writes done on 36 kbit BRAMs. The number of writes per BRAM per clock cycle is two. When $write\_done\_on\_column = \frac{P\_bands}{2}$, all P_bands elements of the different BRAMs have been written, and the entire causal anomaly-removed sample spectral correlation matrix

is stored in BRAM.

Forward_elimination and **Backward_elimination** compute the forward and backward elimination of the Gauss-Jordan elimination.

In **Last division**, the last division of the Gauss-Jordan elimination is computed.

Output_inverse_matrix outputs the matrix A^{-1} stored in **A_inv**. Two rows of the matrix are outputted per clock cycle and sent to **dACAD**.

4.5.3 Forward elimination

Forward elimination contains a FSM with the following valid states: **Idle**, **Check_diagonal_element_is_zero**, **Swap_rows**, **Even_j_write** and **Odd_j_write**. The states are described in Table 4.2.

State	Description
Unknown state	An unknown state. The behaviour of **Forward elimination** is unknown.The FSM should transition to state **Idle**.
Idle	**Forward elimination** is not performing any operations.
Check_diagonal_element_is_zero	Checking if element row_i[index_i] = 0 as done in Gauss-Jordan elimination.
Swap_rows	Swapping row_i and row_j of A stored in **A**.
Even_j_write	Updating an even indexed row of A and A^{-1}.
Odd_j_write	Updating an odd indexed row of A and A^{-1}.

Table 4.2: States of the forward elimination FSM

The FSM controlling **Forward elimination** can be seen in Figure 4.17. $flag\_prev\_row\_i\_at\_odd\_row$ is a flag used as control to indicate whether or not the previous row_i was located at an odd indexed row.

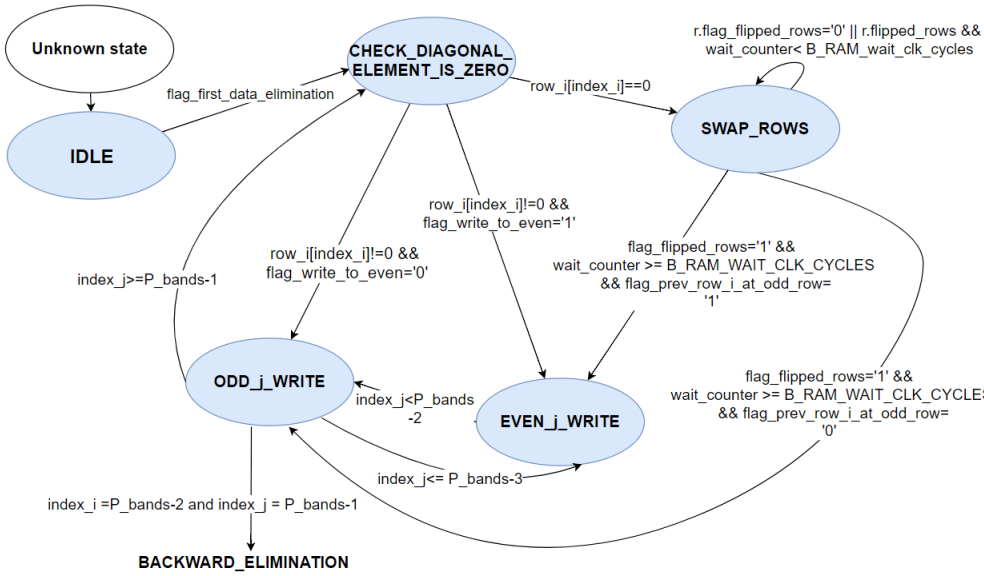

Figure 4.17: FSM controlling **Forward elimination**.

State **Check_diagonal_element_is_zero** executes the check shown in Figure 4.18. If the check evaluates to true, the FSM transitions to state **Swap_rows**. If it evaluates to false, it transitions to either **Even_j_write** or **Odd_j_write**, depending upon if the location of the outer loop index i is at an odd or even index.

$$\textbf{if}(\mathbf{A}[\text{row}[i]][i] == 0)\{$$

Figure 4.18: The check done in state **Check_diagonal_element_is_zero**.

Swap rows executes the operations given in Figure 4.19. When two rows are swapped, the FSM needs to wait for $B\_RAM\_wait\_clk\_cycles$ before issuing transitioning to another state and issuing new reads. This is to ensure that data read is valid, and that the swap has been executed in **A**.

```
for(j = i + 1; j < n; j++){
  if(A[row[j]][j]! = 0){
    row [i] = row[j];
    row[j] = row[i]; // This operation is done in parallel with
     the previous one
     break;
  }end if
}end for
```

Figure 4.19: Operations done in **Swap rows**.

Even_j_write issues writes for an even indexed row of A and A^{-1} to **A** and **A_inv**, by driving the control and data signals to **Elimination core**. It also issues reads to **A** and **A_inv**. Data is structured and sent to **Elimination core**. The operation of the state is illustrated in Figure 4.20. In this example $P\_bands = 6$, $index\_i$=0, $index\_j$=2, $w\_address$=1 and $r\_address$=2. The green row marks row_i. Elements marked by red are writes being issued. Yellow elements are reads being issued.

$index\_i$ and $index\_j$ correspond to the loop indexes i and j of the **Forward elimination** and **Backward elimination**. row_i and row_j are the rows of the matrices A and A^{-1} indexed by $index\_i$ and $index\_j$.

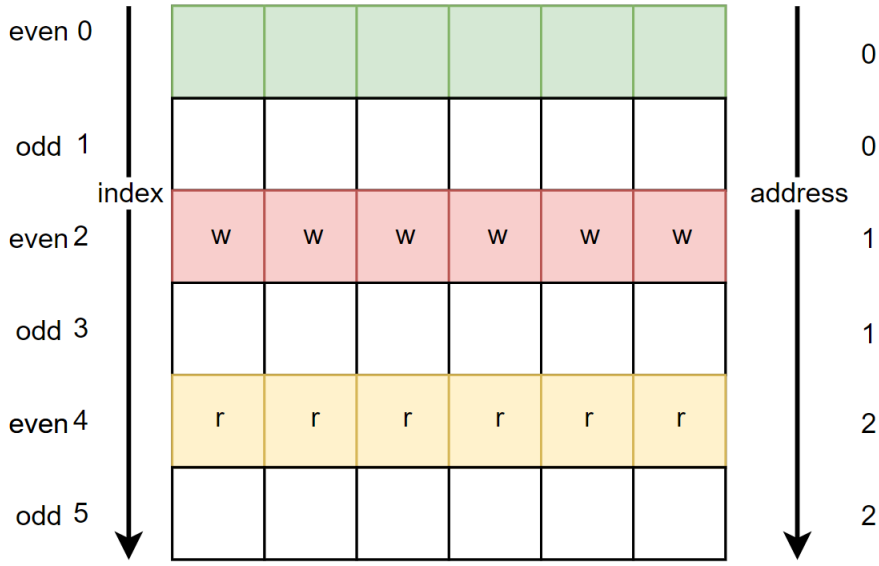

Figure 4.20: Even_j_write in the Backward elimination state.

Odd_j_write issues writes for an odd indexed row of A and A^{-1} to **A** and **A_inv**, by driving the control and data signals of **Elimination core**. It also issues reads to to **A** and **A_inv**. Data is structured and sent to **Elimination core**. The operation of the

state is illustrated in Figure 4.21. In this example $P\_bands = 6$, $index\_i$=0, $index\_j$=3, $w\_address$=1 and $r\_address$=2.

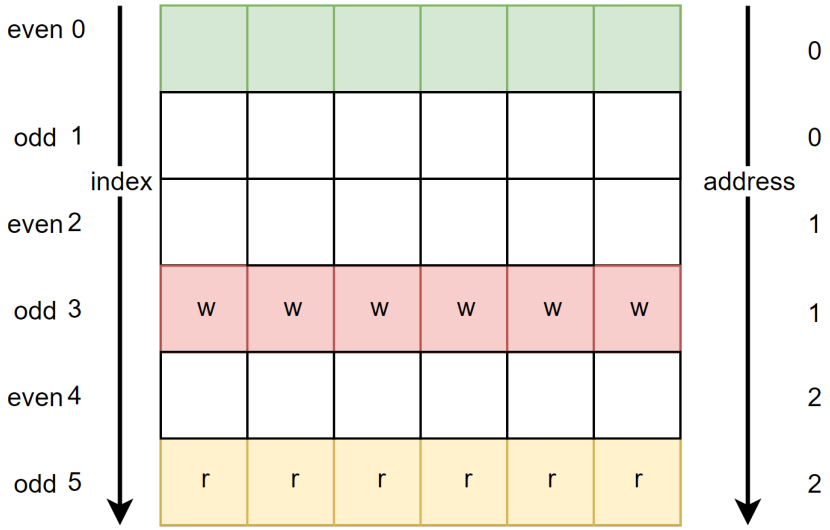

Figure 4.21: Odd_j_write in the forward elimination state.

4.5.4 Backward elimination

Backward elimination contains a FSM with the following valid states: **Idle**, **First_elimination**, **Even_i_start**, **Odd_i_start**, **Even_j_write** and **Odd_j_write**. These are shown in Table 4.3.

State	Description
Unknown state	An unknown state. The behaviour of the **Backward elimination** is unknown. The FSM should transition to state **Idle**.
Idle	**Backward elimination** is not performing any operations.
First_elimination	Doing the first backward elimination iteration in the inverse computation. The first row written will be to an even indexed row.
Odd_i_start	Starting at a new iteration of the outermost loop of the backward elimination loop in the Gauss-Jordan elimination. Starting at an odd indexed row $index\_i$ of the matrix. Computing $A[row\_j]$ and $A^{-1}[row\_j]$, which is at an even row index.
Even_i_start	Starting at a new iteration of the outermost loop of the backward elimination loop in the Gauss-Jordan elimination. Starting at an even row $index\_i$ of the matrix. Computing $A[row\_j]$ and $A^{-1}[row\_j]$, which is at an odd row index.
Even_j_write	Updating an even indexed row of A and A^{-1}.
Odd_j_write	Updating an odd indexed row of A and A^{-1}.

Table 4.3: States of the backward elimination FSM.

The FSM controlling **Backward elimination** can be seen in Figure 4.16. $flag\_index\_i\_at\_odd\_row$ is a flag used to signal if previous row_i was located at an

odd indexed row.

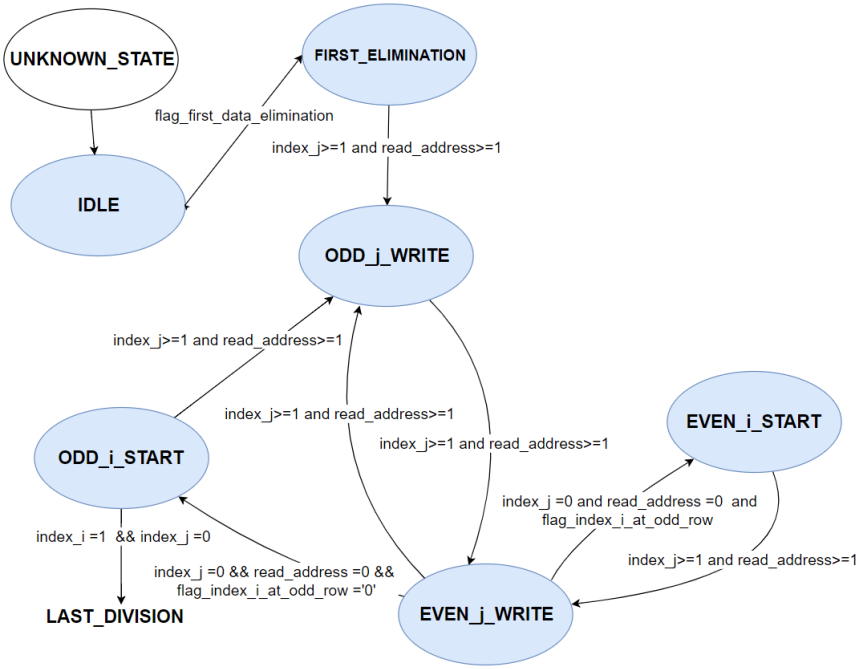

Figure 4.22: FSM controlling **Backward elimination**.

First_elimination is the first iteration of the backward elimination. The flag $flag\_first\_data\_elimination$ is asserted once the two rows with index P_bands-1 and P_bands-2 have been read from memory. **First_elimination** will always issue a write to an even row.

Even_j_write issues writes for an even indexed row of A and A^{-1} to **A** and **A_inv**, by driving the control and data signals to **Elimination core**. It also issues reads to **A** and **A_inv**. Data is structured and sent to **Elimination core**. The operation of the state is illustrated in Figure 4.23. In this example $P\_bands = 6$, $index\_i$=P_bands-1, $index\_j$=4, $w\_address$=2 and $r\_address$=1.

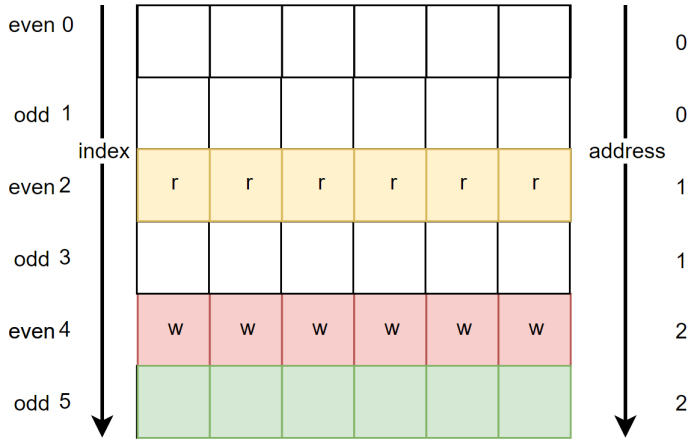

Figure 4.23: Even_j_write in backward elimination.

Odd_j_write issues writes for an odd indexed row of A and A^{-1} to **A** and **A_inv**, by driving the control and data signals to **Elimination core**. It also issues reads to **A** and **A_inv**. Data is structured and sent to **Elimination core**. The operation of the state is illustrated in Figure 4.24. In this example $P\_bands = 6$, $index\_i$=P_bands-1, $index\_j$=3, $w\_address$=1 and $r\_address$=0.

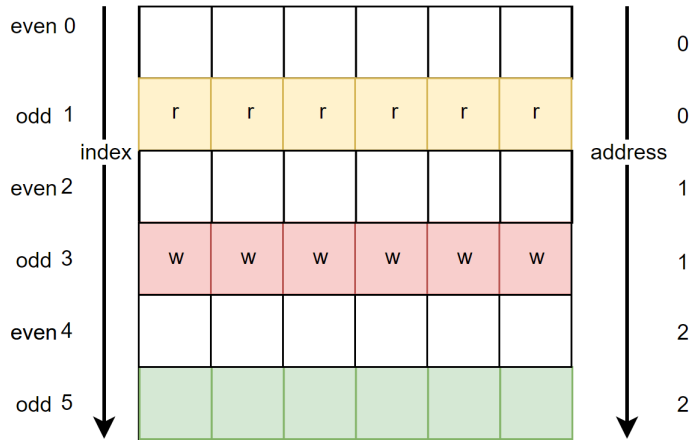

Figure 4.24: Odd_j_write in backward elimination.

Odd_i_start is a new iteration of the outermost loop in backward elimination. $index\_i$ is located at an odd indexed row. In this state, a write is issued to an even indexed row by driving the control and data signals to **Elimination core**. An example is shown in Figure 4.25. For this example P_bands=6, $index\_i$=3, $index\_j$=2, $w\_address$=1 and $r\_address$=0.

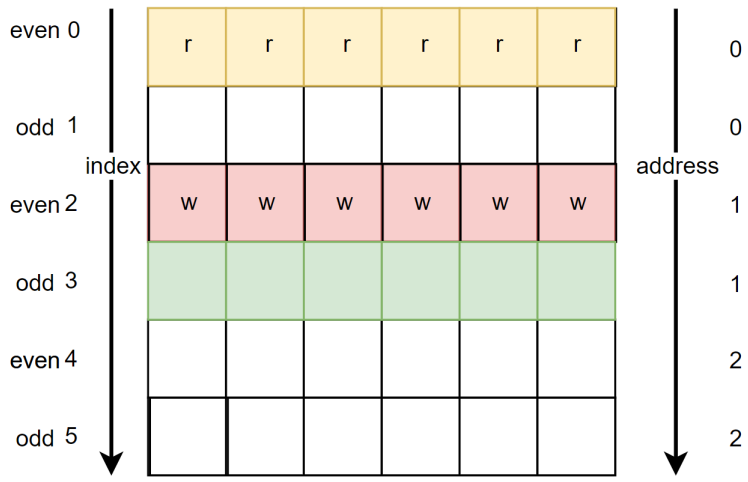

Figure 4.25: Odd_i_start.

Even_i_start is a new iteration of the outermost loop in backward elimination. *index_i* is located at an odd indexed row. In this state, a write is issued to an even indexed row by driving the control and data signals to **Elimination core**. An example is shown in Figure 4.26. For this example *P_bands*=6, *index_i*=4, *index_j*=3, *w_address*=1 and *r_address*=0.

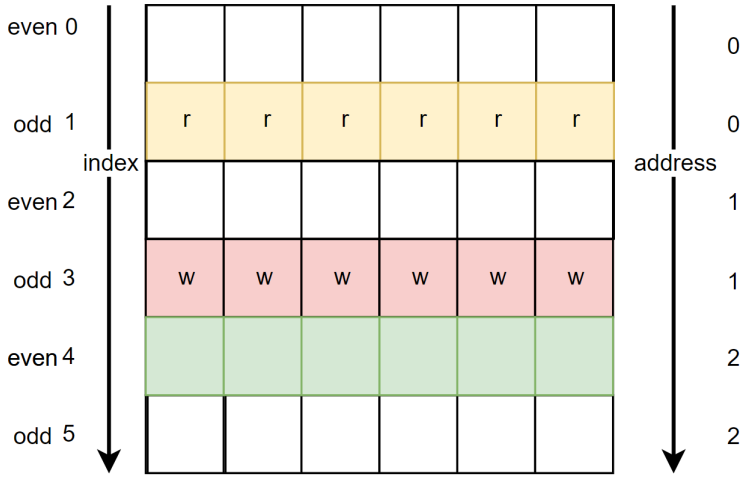

Figure 4.26: Even_i_start.

4.5.5 Last division

Last division contains a FSM with the following valid states: **Idle**, **Even_i_write** and **Odd_i_write**. These are described in Table 4.4.

State	Description
Unknown state	An unknown state. The behaviour of **Last division** is unknown. The FSM should transition to state **Idle**.
Idle	**Last division** is not performing any operations.
Even_i_write	Updating an even indexed row of A^{-1}.
Odd_i_write	Updating an odd indexed row of A^{-1}.

Table 4.4: States of the last division FSM.

In **Even_i_write**, an even indexed row of A^{-1} is updated. A read is issued for the next even indexed row.

In **Odd_i_write**, an odd indexed row of A^{-1} is updated. A read is issued for the next odd indexed row.

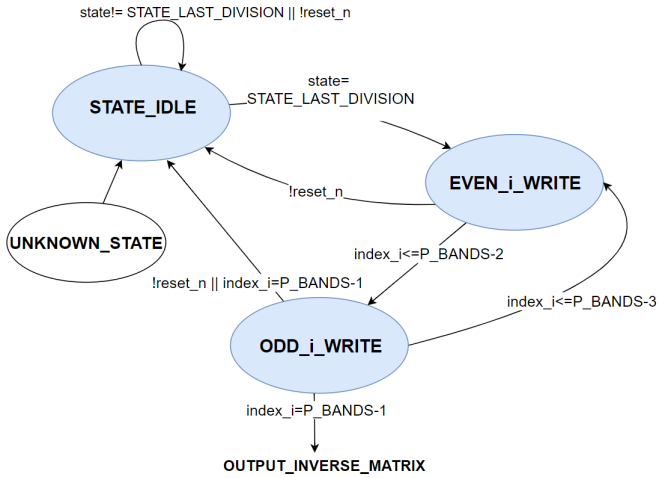

Figure 4.27: FSM controlling **Last division**.

4.5.6 Output inverse matrix

In the **Output_inverse_matrix** state, the contents of **A_inv** is read and outputted to **dACAD**. Reading the contents of **A_inv** takes $\frac{P\_bands}{2}$ clock cycles.

4.5.7 Inverse pipeline stages

The inverse module is pipelined into four stages in order to achieve high throughput. The pipeline can be seen in Figure 4.28, Figure 4.29 and Figure 4.30. The green squares mark processes in which data is written to **A**/**A_inv**. Blue squares represent reading of data from **A**/**A_inv**. Purple squares mark inputs being set from **FSM inverse** and **A** and **A_inv** to **Forward elimination**, **Backward elimination**, **Elimination core** and **Last division**. Yellow squares mark calculation of new data for row_j by either **Forward elimination**, **Backward elimination** or **Last division**.

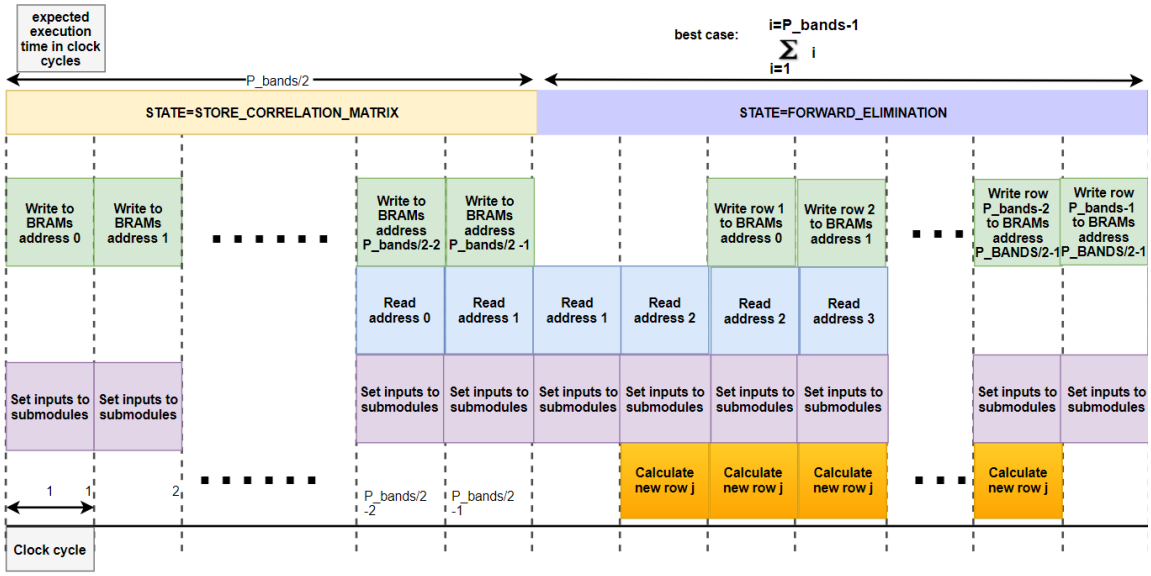

Figure 4.28: Showing pipeline operations in the Store_correlation_matrix and Forward_elimination states.

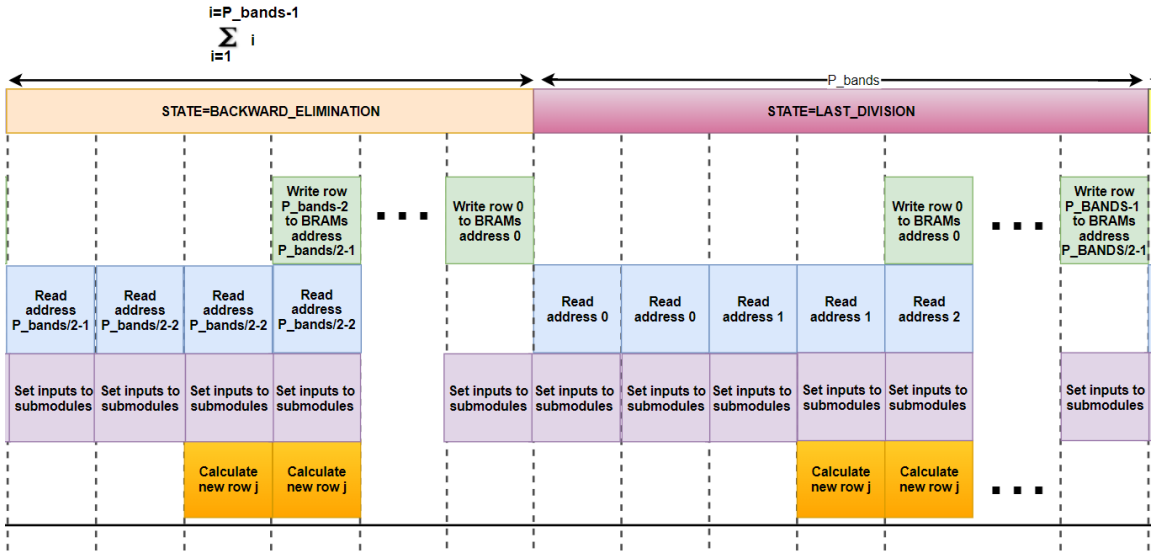

Figure 4.29: Showing pipeline operations in the Forward_elimination and Last_division states.

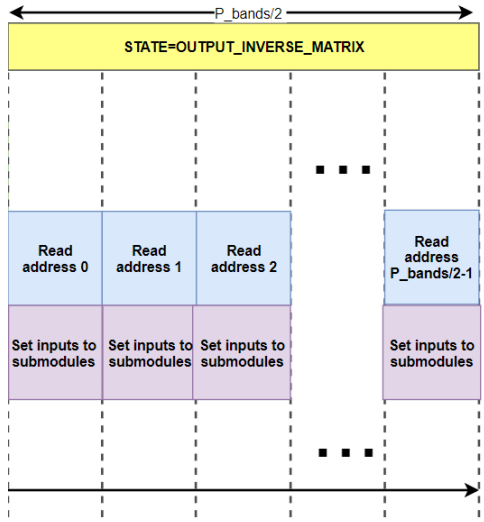

Figure 4.30: Showing pipeline operations in the *Output_inverse_matrix* state.

4.5.8 Execution time expectations inverse computation

Using *P_bands* 36 kbit BRAMs for storage of A and A^{-1} enables to read and write a maximum of two rows to and from **A** and **A_inv** per clock cycle. Assuming that each of the row-operations in the Gauss-Jordan elimination can be calculated within one clock cycle, it is possible to do an estimation of the expected execution time in clock cycles for the inverse computation per pixel. By using that assumption and the fact that a swap of rows can be executed within one clock cycle, this means each of the inner loops in the Gauss-Jordan elimination can be completed within one clock cycle.

Expected execution time for the different states is shown in Figure 4.28, Figure 4.29 and Figure 4.30. For state *Forward_elimination*, the execution time will be greater if it is necessary to swap rows, which is done in the state **Swap rows**. The worst case execution time of *Forward_elimination* is assumed to be when the first element of the matrix A has a zero element at row(i,i) and all other rows, except the last row, which has a zero element at row(j,j).

A worst case and a best case execution time, *inv_worst_case* and *inv_best_case*, for the computation of the inverse per pixel are estimated. The estimations are shown in equations 4.6 and 4.7. *N_STATES_INV* is the number of valid states in the inverse top level FSM, shown in Figure 4.16. *worst_case_ex_state* is the set of expected worst case execution times for the states. *best_case_ex_state* is the set of expected best case

execution times for the states.

$$
\begin{aligned}
inv\_worst\_case &= \sum_{i=0}^{N\_STATES\_INV} worst\_case\_ex\_state(i) \\
&= \underbrace{\frac{P\_bands}{2}}_{\text{STORE\_CORRELATION\_MATRIX}} + \overbrace{\sum_{i=0}^{P\_bands-1} i + P\_bands}^{\text{STATE\_FORWARD\_ELIMINATION}} \\
&+ \underbrace{\sum_{i=0}^{P\_bands-1} i}_{\text{STATE\_BACKWARD\_ELIMINATION}} + \overbrace{P\_bands}^{\text{LAST\_DIVISION}} + \underbrace{P\_bands/2}_{\text{OUTPUT\_INVERSE\_MATRIX}} \\
&= 3P\_bands + 2 \sum_{i=0}^{P\_bands-1} i
\end{aligned}
\tag{4.6}
$$

$$
\begin{aligned}
inv\_best\_case &= \sum_{i=0}^{N\_STATES\_INV} best\_case\_ex\_state(i) \\
&= \underbrace{\frac{P\_bands}{2}}_{\text{STORE\_CORRELATION\_MATRIX}} + \overbrace{\sum_{i=0}^{P\_bands-1} i}^{\text{STATE\_FORWARD\_ELIMINATION}} \\
&+ \underbrace{\sum_{i=0}^{P\_bands-1} i}_{\text{STATE\_BACKWARD\_ELIMINATION}} + \overbrace{P\_bands}^{\text{LAST\_DIVISION}} + \underbrace{P\_bands/2}_{\text{OUTPUT\_INVERSE\_MATRIX}} \\
&= 2P\_bands + 2 \sum_{i=0}^{P\_bands-1} i
\end{aligned}
\tag{4.7}
$$

Figure 4.31 shows the estimated execution time in seconds for computing $\tilde{\mathbf{R}}^{-1}(\mathbf{x}_k)$ for all $\mathbf{x}_k$ in the hyperspectral image, for an image size of 1088x576, with an operating clock frequency of 100MHz.

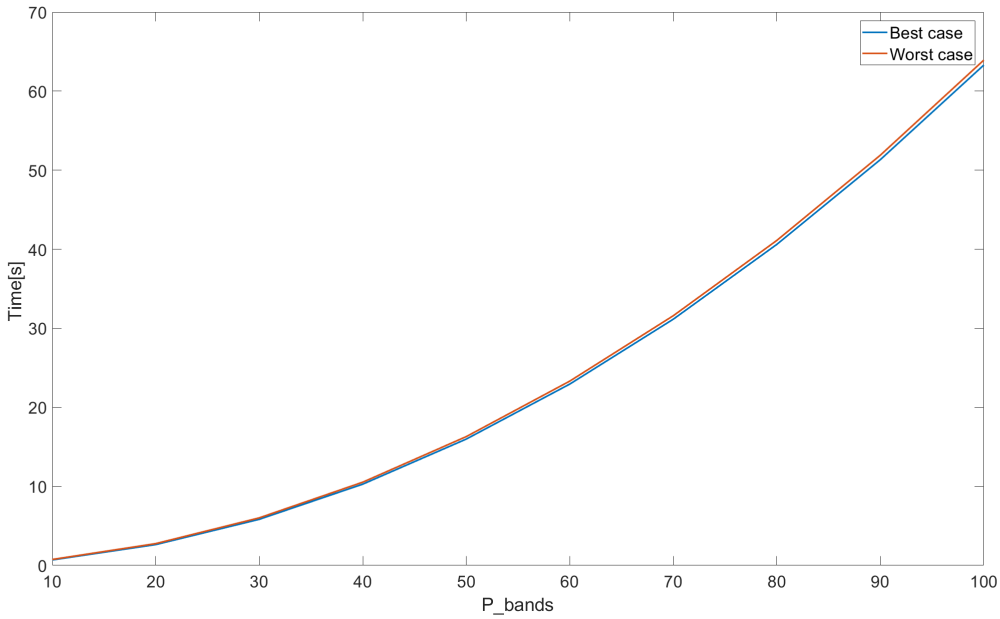

Figure 4.31: Estimated execution time for computation of $\tilde{\mathbf{R}}^{-1}(\mathbf{x}_k)$ for an image of size 1088x576 in seconds.

4.5.9 Division

A drawback with the Gauss-Jordan algorithm is that it uses division. Division is an operation that is computationally intensive, and it requires a large amount of logic to be implemented. An early implementation of the Gauss-Jordan elimination by the author utilized the division operator "/". This is further described in Section 4.5.9.1.

An approach to implement division by adaptive shifting is described in Section 4.5.9.2.

A third approach for computing division was made. This approach utilizes LUTs to store an array containing the inverse of the divisor in the division. It is further described in Section 4.5.9.3.

The semantics used to described the division operation will be $C = B * \frac{1}{a}$ where C, B and a are integers in the range of s=[-$2^{Pixel\_data\_width \times 2}$-1,..., $2^{Pixel\_data\_width \times 2}$-1].

4.5.9.1 Using the division operator "/"

To evaluate if division could be implemented by using the "/" operator for signed datatypes, the **Last division** block was synthesized, and the worst negative slack (WNS) was used as a criteria to see if the design met the system clock target constraint of 100 MHz. The

max frequency, f_{max}, is calculated by equation 4.8:

$$f_{max} = \frac{1}{-WNS + 10ns}. \tag{4.8}$$

Results for different divisor-and-divident bit widths are presented in Table 5.3. The data flow for block **Last division** using the division operator can be seen in Figure 4.32.

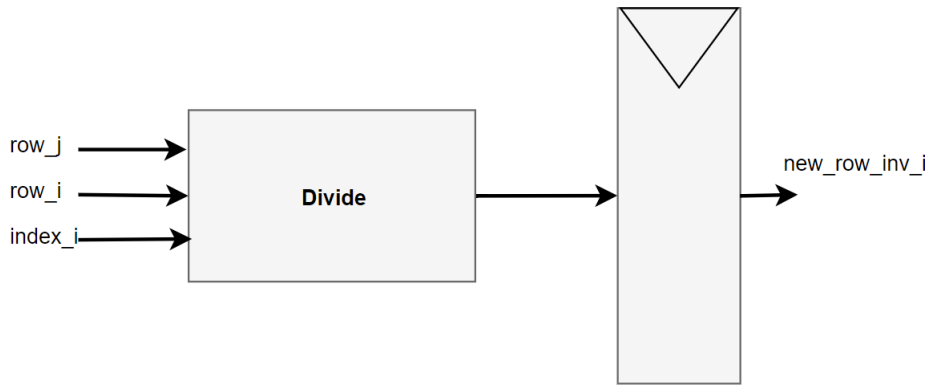

Figure 4.32: Dataflow of block **Last division** using the division operator "/" for division.

4.5.9.2 Adaptive shifting

To avoid using the division operator, the adaptive shifting approach shown in Figure 4.33 has been implemented. It approximates the divisor by an adaptive number of shift operations as the divisor is not constant. To achieve this, the most significant bit (MSB) of the divisor is first checked to evaluate if the divisor is a negative number. If it is, the divisor is negated. The block **Find MSB** finds the MSB of the unsigned divisor. In parallel, $Pixel\_Data\_Width$ × 2-1 numbers of shift-operation processes shifts the unsigned divisor by n_shifts=[1,2...$Pixel\_Data\_Width$ × 2 - 1]. These shift-operation processes are illustrated in Figure 4.33 by the blocks **Right shift one**, **Right shift two** and **Right shift PIXEL_DATA_WIDTH*2-1**. The remainders after shifting are sent to the **Choose best approximation** block. This block chooses the best approximation depending upon the index of the MSB and the remainders after shifting. The best approximation to the divisor will be a shift operation by MSB or MSB+1 number of shifts. Each element of the row inv_row_i is then shifted in parallel to compute the approximate division. If the divisor is a negative number, the row is negated before outputting data to register.

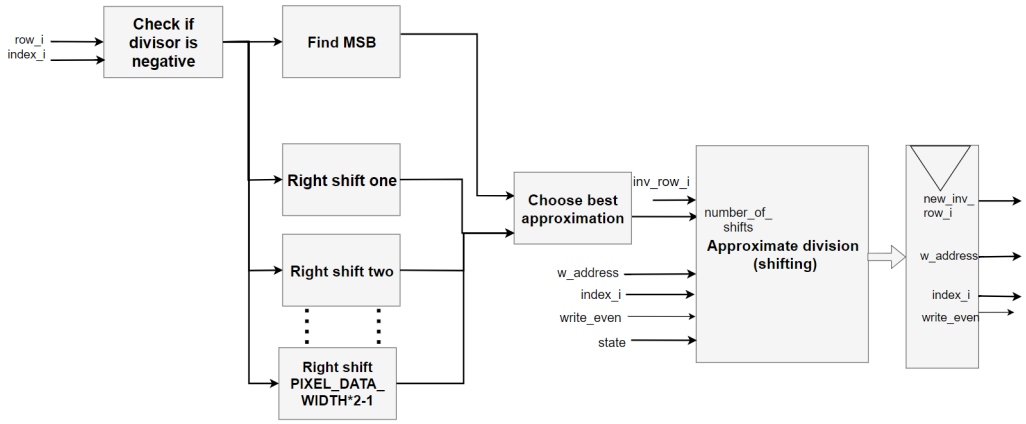

Figure 4.33: Architecture of block **Last division**, approximating division with an adaptive number of shifts.

4.5.9.3 LUT approach

Instead of computing the division in the operation $C = B \times \frac{1}{a}$, an approach based on the solution in [24] was used. This approach utilizes LUTs to store the array $divisor\_inv = \frac{2^{Div\_Precision}}{a}$, where $a = [1, 2...2^{Div\_Precision}]$. $Div\_Precision$ is the bit width of the divisor possible to represent with this approach. If the MSB of the divisor a is at an index $> Div\_Precision$, the adaptive shifting approach is used. If not, a is used as an index to look up in LUTs storing $divisor\_inv$. Then, $divisor\_inv(a)$ is multiplied by B, which yields product C. C is right shifted $Div\_Precision$ spaces. This can be seen in equation 4.9.

$$C = shift\_right(B \times divisor\_inv(a), Div\_Precision) \tag{4.9}$$

The code for inferring LUTs for storage of $divisor\_inv$ is shown in Listing 4.3, exemplified for $Div\_Precision$=4.

Listing 4.3: LUT division approach exemplified for Div_Precision = 4.

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity division_lut is
port(
  y          : in  std_logic_vector(3 downto 0);
  y_inv      : out std_logic_vector(3 downto 0));
end division_lut;
architecture rtl of division_lut is
constant C_NY       : integer:= 4;
constant C_NDY      : integer:= 4;
type t_divition_lut is array (0 to 2**C_NY-1) of
integer range 0 to 2**C_NDY-1;
```

```
constant C_DIV_LUT  : t_divition_lut := (
  16,
  8,
  5,
  4,
  3,
  3,
  2,
  2,
  2,
  2,
  1,
  1,
  1,
  1,
  1,
  1
   );
begin
  y_inv <= std_logic_vector(to_unsigned(
    C_DIV_LUT(to_integer(unsigned(y))),C_NDY));

end rtl;
```

The values of *divisor_inv* can be generated by the MATLAB script shown in Listing 4.4.

Listing 4.4: MATLAB code for generating the values of *divisor_inv*, for *Div_Precision* = 17.

```
clear; clc;

 DIV_PRECISION =17;
 division_lut_values = zeros(1,2^DIV_PRECISION);
 file = fopen('generated_luts_17_bit.txt', 'w');

for i =1: 2^DIV_PRECISION
    division_lut_values(i)=(2^DIV_PRECISION *1)/i;
   str = num2str(division_lut_values(i));
    fprintf(file, '%8.0f,\n',division_lut_values(i));
end

fclose(file);
```

The architecture of the **Last division** block, utilizing this LUT approach, is shown in Figure 4.34. If *Div_Precision*< *Pixel_Data_Width* × 2, an adaptive shifting approach is added in parallel. If the MSB is located at an index > *Div_Precision*, then the adaptive shifting approach is used.

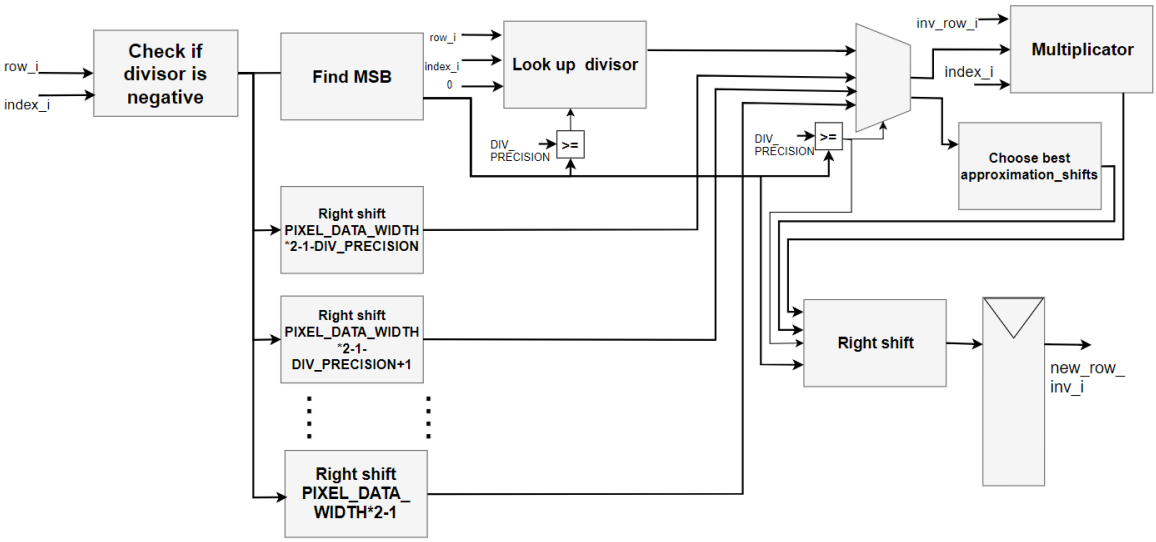

Figure 4.34: Architecture of block **Last division**, computing division using the LUT approach.

Look up divisor looks up the inverse of the unsigned divisor a by using the absolute value of $row\_i[index\_i]$ as an index to the LUTs storing the array $divisor\_inv$. If $Div\_Precision >=$ MSB of the unsigned divisor a, then $divisor\_inv(a)$ is sent to **Multiplicator**, which performs a multiplication, else 1 is sent to **Multiplicator** and **Last division** utilizes the adaptive shifting approach. If $Div\_Precision >=$ MSB the LUT approach is used by **Last division**. The **Choose best approximation_shifts** block choose the best approximation made by the adaptive shifting approach. In block **Right shift**, a right shift operation of b_shift_spaces is performed on each element of the row inv_row_i, which is the row indexed by i in matrix A^{-1} stored in **A_inv**. b_shift_spaces is dependent upon if the adaptive shifting or LUT approach is used.

Chapter 5

Results

5.1 Synthesis

All synthesis results in this chapter are synthesized for $Pixel\_data\_width$ of 16, unless another value is especially mentioned. Results presented in this chapter are gathered from synthesis utilization reports. "Vivado Synthesis Defaults" was used as the "Strategy" in the Option field for the synthesis project settings, in order to get the best trade-off between performance and area.

The designs were synthesized in Vivado. As Zynq-7000 - Z7030/Z7035 was not eligible for synthesis, the Zedboard was used. This kit contains less logic than the Z7030/Z7035, and it contains only 220 DSPs. This leads to the **ACAD inverse** over-utilizing DSPs when running synthesis with $P\_bands >= 20$ and $Pixel\_data\_width = 16$, when utilizing the LUT approach to approximate division. When over-utilizing DSPs, the logic gets mapped to LUTs instead, as described in [25], and will produce unusable synthesis results. Therefore, **ACAD inverse** was synthesized for xc7k160tiffv676-2L for $P\_bands >= 20$ and $Pixel\_data\_width = 16$, as this device contains 600 DSPs as well as having a similar architecture as the Zedboard (has Slice Registers and Slice LUTs, as opposed to Configurable Logic Block (CLB) Registers and CLB LUTs). **ACAD correlation** also over-utilizes DSPs for $P\_bands >= 60$ and $Pixel\_data\_width = 16$. Therefore, **ACAD correlation** is synthesized for xc7k160tiffv676-2L for $P\_bands >= 60$ and $Pixel\_data\_width = 16$.

Timing results when synthesizing for xc7k160tiffv676-2L are not considered usable as the performance of the device logic is different to the Zedboard's. The Zedboard contains an Artix-7 device, which is a slower device than the Z-7030/Z-7035, which are Kintex-7 devices. Designs that meet timing demands for the Zedboard will therefore also most likely meet timing requirements for the Z-7030/Z-7035 devices. In addition to this, the initial test prototype is to be implemented on a Zedboard. As such, it is valuable to see if the design meet timing when running on Zedboard. Therefore, only timing results from synthesis on the Zedboard are presented.

5.1.1 Shiftregister

Shiftregister was synthesized for $P\_bands = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]$. The numbers of synthesized Slice registers and Slice LUTs are shown in Figure 5.1, plotted as a function of P_bands.

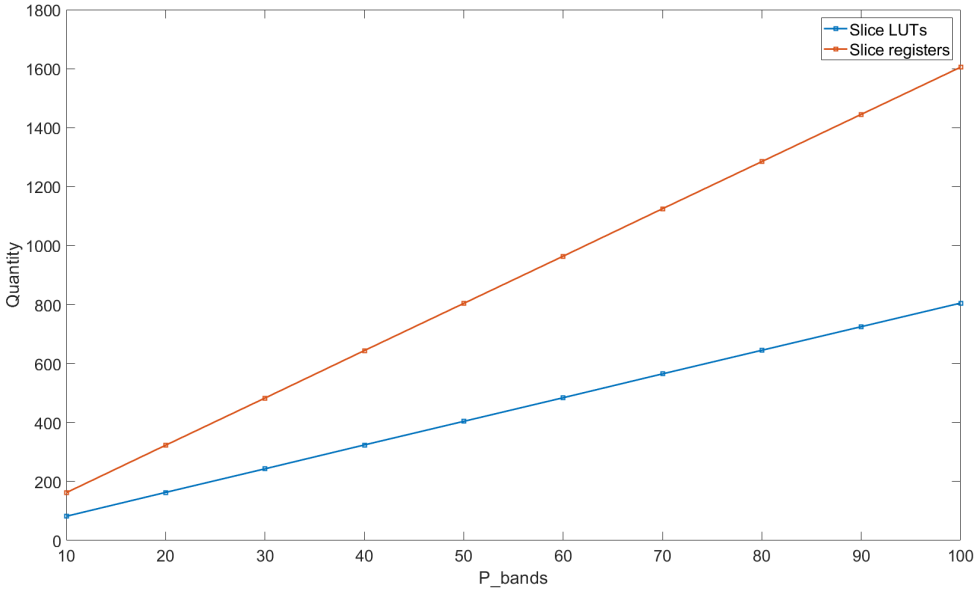

Figure 5.1: Shiftregister synthesis results.

5.1.2 ACAD correlation

The design was synthesized for $P\_bands = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]$. Figure 5.3 shows the number of synthesized BRAM36E1 and DSP48E1. Figure 5.4 shows the number of synthesized Slice Registers and Slice LUTs as a function of P_bands.

The block **Normalize by k**, which is a sub-module of **ACAD correlation** that performs a shift operation depending upon the index k, has not been implemented. The architecture of **ACAD correlation** synthesized is illustrated in Figure 5.2. The synthesis and timing results for **ACAD correlation** without **Normalize by k** do produce relatively accurate results as a shift operation is "free" in hardware, meaning that it does not lead to an increase in logic usage and delay worth mentioning.

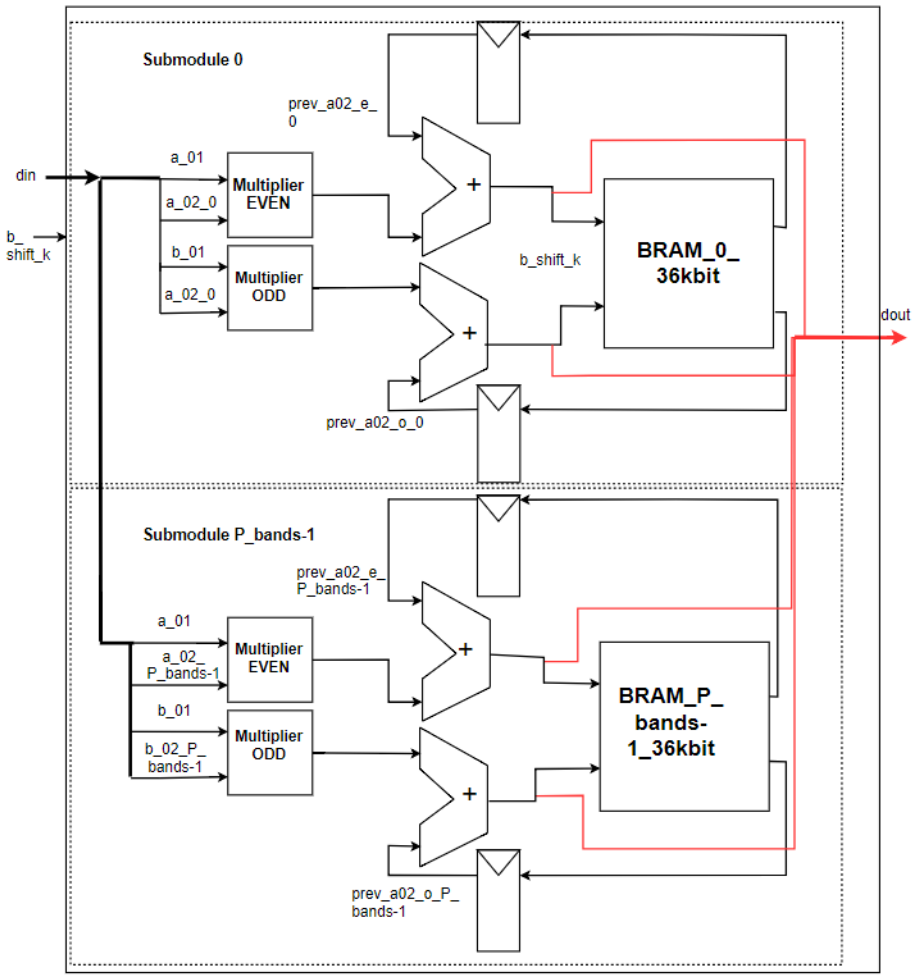

Figure 5.2: Architecture of the implemented version of **ACAD correlation**, without normalization.

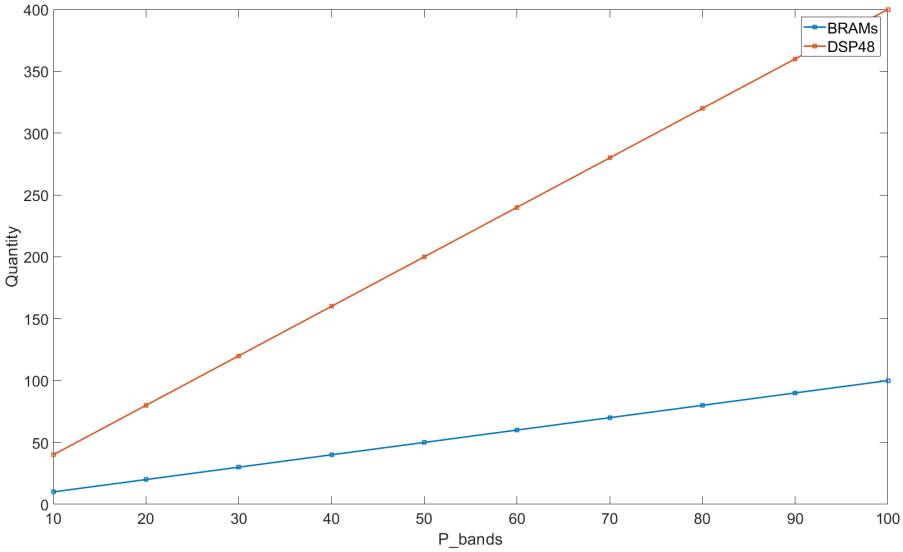

Figure 5.3: Number of synthesized BRAM36E1 and DSP48E1 as a function of *P_bands* for the **ACAD correlation** block.

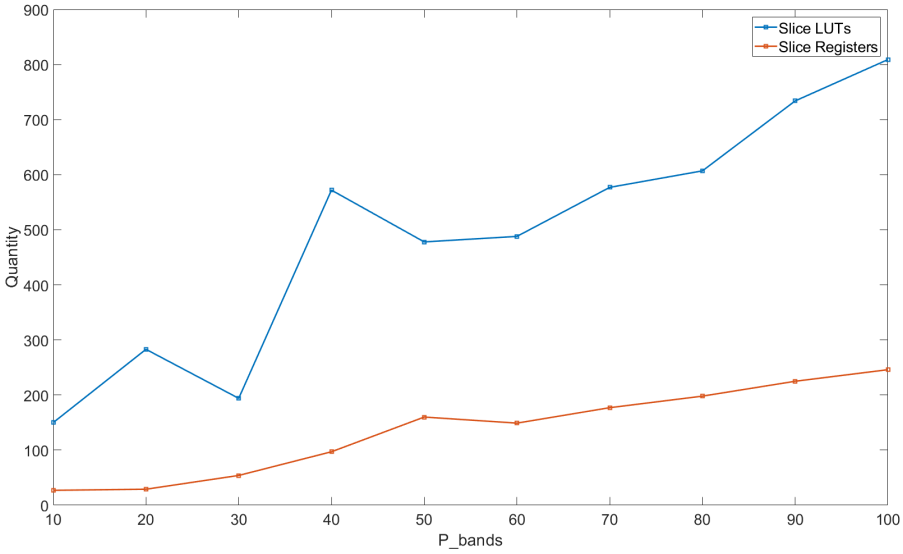

Figure 5.4: Number of synthesized Slice Registers and Slice LUTs as a function of *P_bands* for the **ACAD correlation** block.

5.1.2.1 $Pixel\_data\_width = 10$

As **ACAD correlation** inferred a large number of DSPs, $Pixel\_data\_width$ was lowered to see if the number of DSPs inferred would be decreased. The design inferred DSPs for $Pixel\_data\_width >= 11$, but for $Pixel\_data\_width = 10$ the synthesis tool did not infer any DSPs. Instead, the logic was mapped to LUTs. When varying $Pixel\_data\_width$, the number of BRAMs synthesized are unchanged. The numbers of LUTs and registers synthesized as a function of P_bands are shown in Figure 5.5.

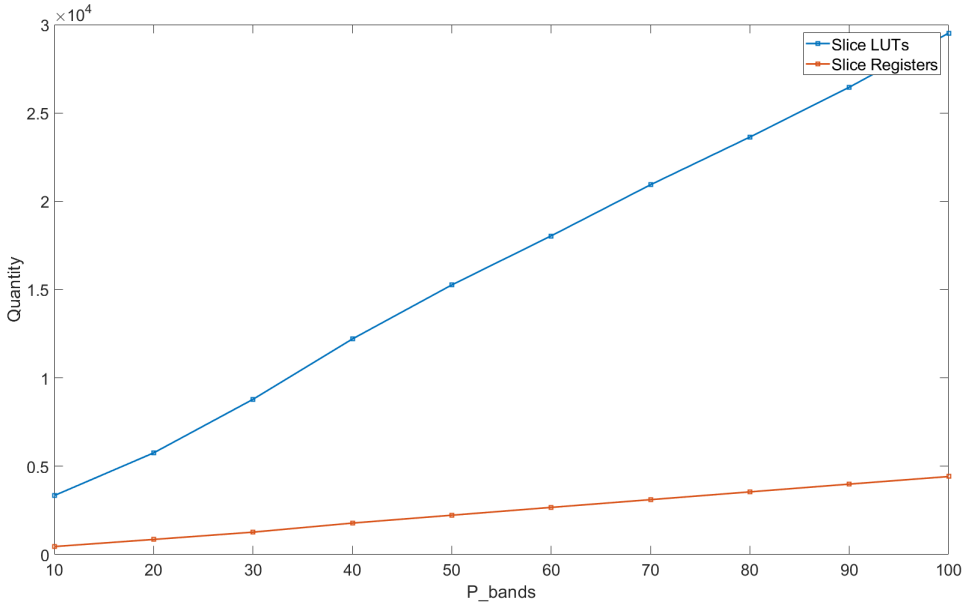

Figure 5.5: The numbers of synthesized Slice Registers and Slice LUTs as a function of P_bands for the **ACAD correlation** block for $Pixel\_data\_with$ =10.

5.1.3 ACAD inverse

ACAD inverse was synthesized using the three different division-approaches. The numbers of BRAMs synthesized for the three approaches are equal as shown in Figure 5.6.

DSPs inferred for the three approaches can be seen in Figure 5.7. The LUT-approach infers more DSPs than the two other approaches.

The numbers of LUTs synthesized for the three approaches are illustrated in Figure 5.8, while the numbers of registers synthesized can be seen in Figure 5.9.

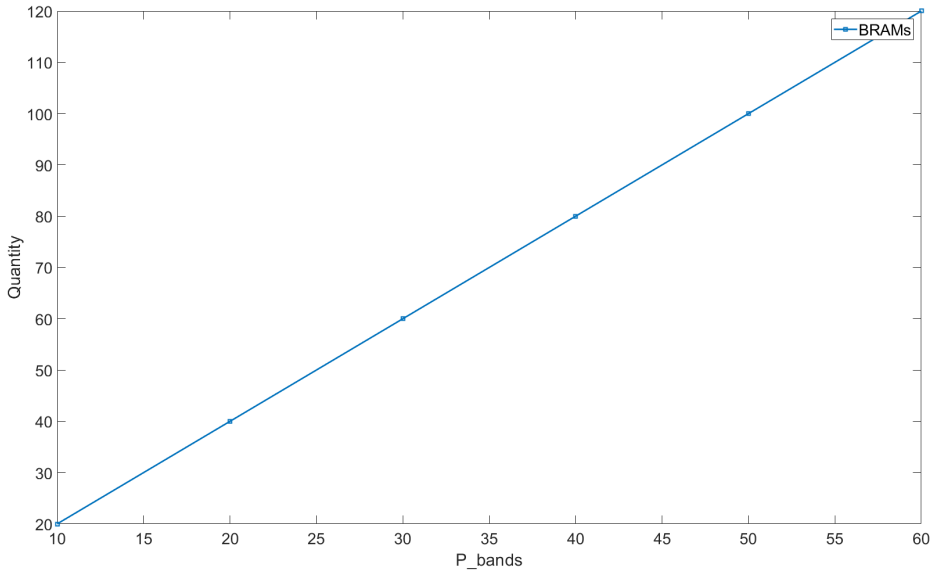

Figure 5.6: Number of BRAMs synthesized for the **Inverse** block.

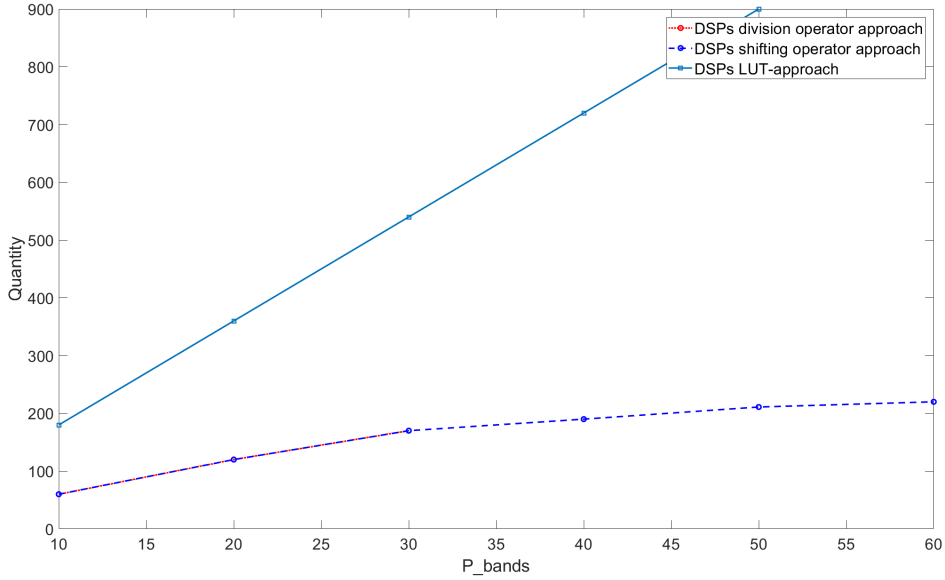

Figure 5.7: Numbers of DSP48E1 synthesized for the **Inverse** block.

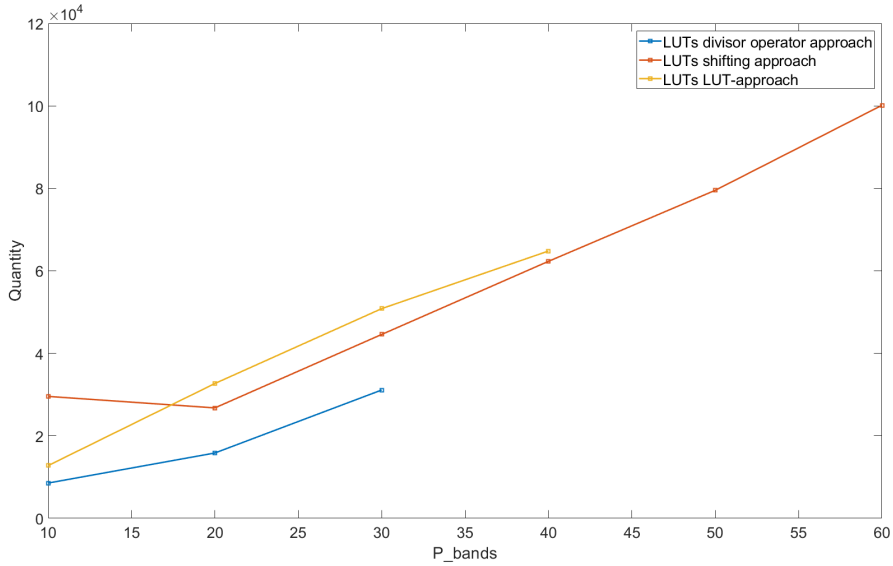

Figure 5.8: Numbers of LUTs synthesized for the **Inverse** block.

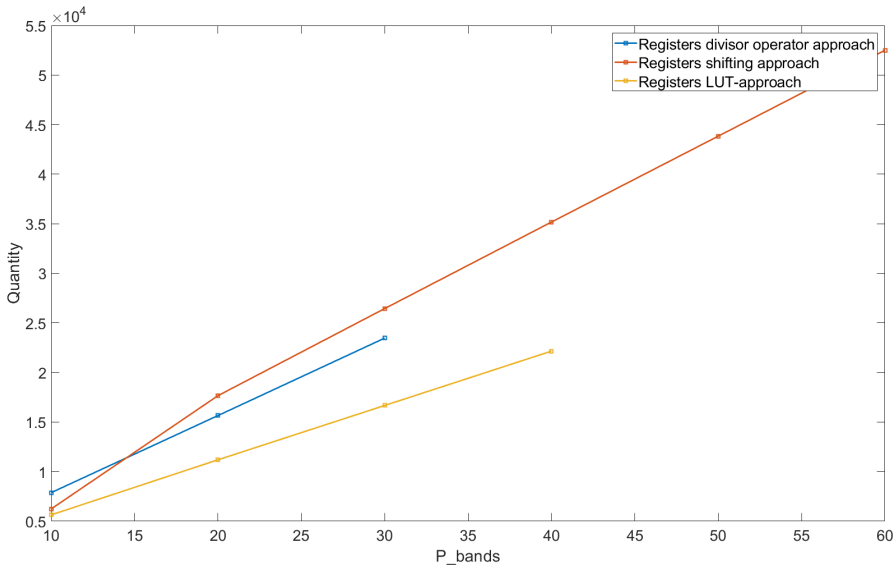

Figure 5.9: Numbers of registers synthesized for the **Inverse** block..

5.1.4 Timing results

To check if the design met timing requirements, the WNS of the synthesized designs was checked. The target clock constraint was set to 100 MHz.

5.1.4.1 WNS ACAD correlation

ACAD correlation was synthesized for $P\_bands = [10, 20, 30, 40, 50]$ and $Pixel\_data\_width = 16$ on the Zedboard. The timing results are presented in Table 5.1.

P_bands	**WNS [ns]**
10	1.272
20	0.721
30	1.446
40	0.845
50	0.509

Table 5.1: Timing results for **ACAD correlation** $Pixel\_data\_width = 16$.

ACAD correlation was synthesized for $P\_bands = [10, 20, 30, 40, 50, 60, 70, 80, 90]$ and $Pixel\_data\_width = 10$ on the Zedboard. The timing results are presented in Table 5.2.

P_bands	**WNS [ns]**	**Net delay [ns]**	**Logic delay [ns]**
10	-3.074	7.860	5.078
20	-3.309	7.868	5.305
30	-3.109	6.732	6.241
40	-3.319	7.698	5.485
50	-3.324	7.703	5.485
60	-3.331	7.518	5.677
70	-4.923	8.563	6.224
80	-4.881	8.521	6.224
90	-5.224	9.373	5.735

Table 5.2: Timing results for **ACAD correlation** $Pixel\_data\_width = 10$.

5.1.4.2 WNS division operator

Implementing division by the use of the division operator "/" yielded the timing results presented in Table 5.3 when synthesizing block **Last division**, computing the product $C = B * \frac{1}{A}$. Width of B is 32 bit. The design was synthesized for dividend and divisor data width of 32, 16, 12 and 10, with P_bands =10 and a target clock constraint of 100 MHz. The target device was the Zedboard.

Data width divisor and dividend	WNS [ns]	f_{max}[MHz]
32	-80.524	11.046
16	-11.776	45.934
12	-7.071	58.578
10	-5.730	63.572

Table 5.3: Synthesis results for Zedboard for **Last division** using the division operator "/".

5.1.4.3 Worst Negative Slack adaptive shifting approach

The design shown in Figure 4.33 was synthesized for Zedboard to check timing for P_bands = [10, 20, 30, 40, 50, 60]. The worst WNS was 2.034 nanoseconds (ns), which yields f_{max} of $\approx 125MHz$.

5.1.4.4 Worst Negative Slack LUT approach

The **Inverse** block was synthesized for Zedboard, with $Div\_Precision = 17$ and P_bands =10. The synthesis results yielded a WNS of -5.972 ns. Net delay accounts for 4.847 of this.

P_bands =10 was chosen for testing of WNS due to the LUT approach over-utilizing DSP48E1s in the Zedboard for a higher number of P_bands.

5.2 Simulation

The designs have been tested on simulation runs in Vivado. The testbenches used for simulation are available at `https://github.com/marthauk/Anomaly-detection/tree/invert_matrix_computation/FPGA_implementation/Anomaly_detection/Anomaly_detection.srcs/sim_1/new`.

5.2.1 Shiftregister

Shiftregister has been simulated and tested for constrained random inputs of *din*, and for values of P_bands dividable by 4 satisfying the condition $modulo(P\_bands, 4) = 0$. Figure 5.10 shows a simulation for $P\_bands = 12$.
dout is the output signal and outputs a spectral pixel vector of size $P\_bands \times Pixel\_data\_width$. *din* is input pixel data of size 64 bit. *valid* and $valid\_out$ are control signals signalling the validity of the input and output signals, while $shift\_counter$ is an internal counter which signalizes how many shifts that have been done.

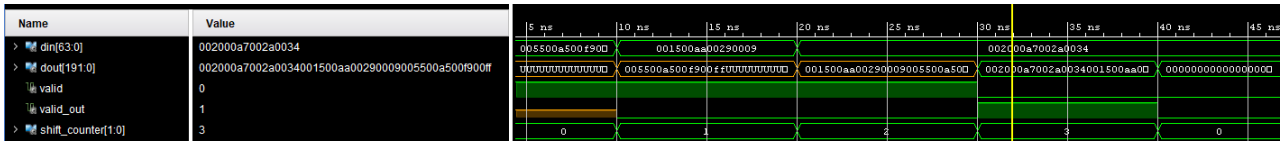

Figure 5.10: Simulation of **Shiftregister** for $P\_bands = 12$.

5.2.2 ACAD correlation

Constrained random input simulation of **ACAD correlation** has been done in Vivado, and the captured waveforms have been visually inspected. The waveforms shown in Figure 5.11 and Figure 5.12 show simulations of input pixel vectors of size $P\_bands = 4$, $Pixel\_data\_width = 16$.

Figure 5.11 shows a simulation for a data input pixel vector = [0x00ff, 0x00f9, 0x00a5, 0x0055]. This is simulated to be the first pixel of the hyperspectral image. *valid* is an input signal signalizing if the input *din* is valid. The output *data_out* outputs two rows of the causal anomaly-removed correlation matrix per clock cycle.

Figure 5.12 shows a simulation for a data input pixel vector = [0x0015, 0x00aa, 0x0029, 0x0009]. This is simulated to be the second pixel of the hyperspectral image. As can be seen by the output signal *data_out*, the contents of the BRAMs, which store the causal anomaly-removed correlation matrix for the previous pixel, are added to the causal anomaly-removed correlation matrix of the current pixel.

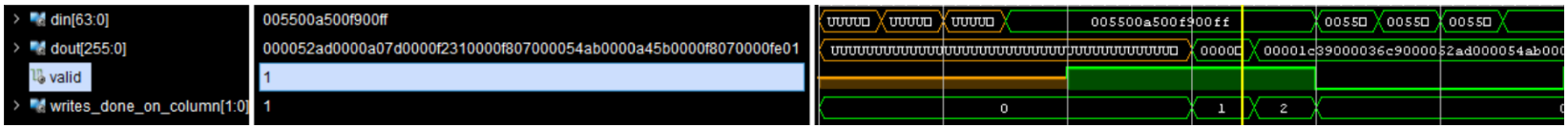

Figure 5.11: Simulation of the **ACAD correlation** block.

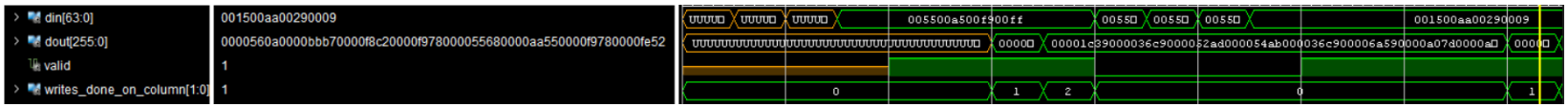

Figure 5.12: Simulation of the **ACAD correlation** block.

5.2.3 Inverse

Simulation has been done for P_bands = [4,6] and $Pixel\_data\_width$ = 16, with constrained random input. Figure 5.13 shows a simulation with P_bands = 4. For this simulation the division operator "/" is utilized. Data input signal *din* is a P_bands × $Pixel\_data\_width$ × 2 wide signal. The output $inverse\_rows$ is a P_bands × 2 × $Pixel\_data\_width$ × 2 wide signal which outputs two rows of the inverse matrix A^{-1}, stored in **A_inv**, per clock cycle, while the signals $data\_out\_brams\_M\_inv$ and $data\_out\_brams\_M$ are data that are read from **A_inv** and **A** respectively. The signal *state* is the state of the top level inverse FSM.

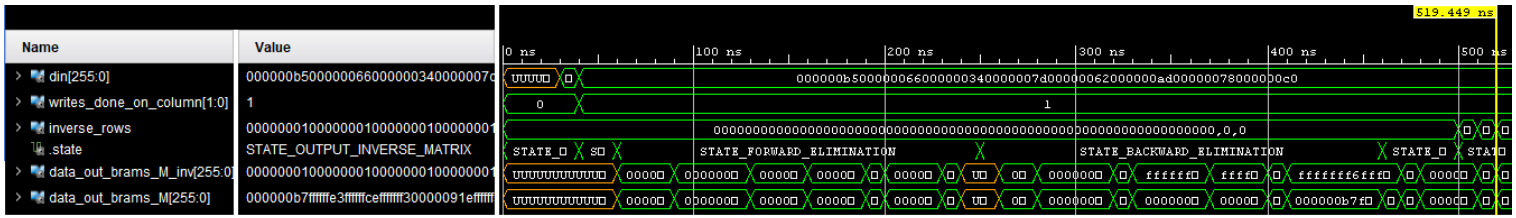

Figure 5.13: Simulation of the **Inverse** block.

Chapter 6

Discussion

6.1 Resource usage

ACAD is more suited for hardware implementation than the RX and LRX algorithms, due to its causality, creation of an anomaly map and use of causal correlation matrix. Additionally, it enables real-time or near real-time performance. It is however computationally intensive and requires a lot of resources.

6.1.1 DSP usage $Pixel\_data\_width = 16$

Synthesis results show that **ACAD correlation** infers $P\_bands \times 4$ DSP48E1s and P_bands BRAMs for $Pixel\_data\_width = 16$.

When using the LUT approach, **ACAD inverse** also infers a large number of DSPs. The Zynq Z-7030 contains 400 DSP Slices, while the Z-7035 contains 900 DSP Slices. According to synthesis results, the LUT approach for implementation of **ACAD inverse** utilizes 540 DSPs for $P\_bands = 30$ and $Div\_Precision = 17$. As **ACAD correlation** infers $P\_bands \times 4$ DSPs for $Pixel\_data\_width >= 11$, the total number of DSPs synthesized for these two modules for $P\_bands = 30$, $Pixel\_data\_width = 16$ and $Div\_Precision = 17$ will be 660. This is a high number, especially considering that **dACAD** computes $\delta^{ACAD}(\mathbf{x}_k) = \mathbf{x}_k^T \tilde{\mathbf{R}}^{-1}(\mathbf{x}_k)\mathbf{x}_k$, where $\mathbf{x}_k$ is a pixel vector of size $P\_bands \times Pixel\_data\_width$ and $\tilde{\mathbf{R}}^{-1}(\mathbf{x}_k)$ is a matrix of size $P\_bands \times P\_bands \times Pixel\_data\_width \times 2$. Depending upon the implementation, this computation will most likely also utilize DSPs. The large number of DSPs used constrains the size of the pixel vector possible to input to the ACAD AD. For the Zynq Z-7030, while the maximum number of spectral bands of the pixel vector will be ≈ 20. It will be ≈ 40 for the Zynq Z-7035. The initial prototype is to be implemented on the Zedboard Zynq Evaluation and Development kit (referred to as Zedboard), which only contains 220 DSPs. Therefore, the maximum size of P_bands is ≈ 10 for the Zedboard.

6.1.2 $Pixel\_data\_width = 10$

When synthesizing **ACAD correlation** for $Pixel\_data\_width = 10$, no DSPs are inferred. Instead, the logic gets mapped to LUTs as shown in Figure 5.5. By doing this, the number of DSPs used by the ACAD AD is heavily reduced. This might be an important consideration for the SmallSat project as the number of DSPs inferred by the ACAD AD is high, which constrains the value of P_bands.

6.2 Timing results

6.2.1 ACAD correlation

ACAD correlation meets timing demands for $Pixel\_data\_width = 16$. For $Pixel\_data\_width = 10$, the synthesized design infers no DSPs, but maps the logic to LUTs. The WNS of the design is negative for $Pixel\_data\_width = 10$ as can be seen in Table 5.2, meaning **ACAD correlation** fails to meet timing demands.

But, as can also be observed in Table 5.2, the net delay is high and increasing as a function of P_bands. This is due to the output ports of **ACAD correlation** getting mapped to physical output pins on the synthesized device. However, this will not be the case in implementation as the output ports of **ACAD correlation** are connected to **ACAD inverse** and **FSM ACAD**. As such, the net delay is most likely unrealistically large, as mapping to output pins scattered on the physical interface of the device will result in a higher delay than mapping to internal buses located inside the device. Therefore, the author believe that **ACAD correlation** will meet timing demands once the design is a sub-module of the ACAD anomaly detector, and the output ports are mapped to an internal bus instead of actual output pins.

6.2.2 ACAD inverse

Implementing division using the division operator "/" is not viable as the **Last division** block fails to meet timing requirements when using this approach. This holds for dividend-and-divisor bit width down to 10.

The adaptive shifting approach is an interesting approach for implementation of division, and the approach meets timing requirements. A big uncertainty however, is the effect of precision error when utilizing this approach.

Implementing division through the LUT approach reveals promising results with regards to timing, especially when taken into account that the author has not focused on optimizing the LUT approach with regards to timing as the approach was implemented late in the process of writing this thesis. Precision errors are most likely less probable in this approach as opposed to the adaptive-shifting approach, especially when using a large value for $Div\_Precision$. The synthesis results for the **ACAD inverse** block when using LUT approach with $Div\_Precision = 17$ yielded a WNS of -5.972ns, in which 4.847 of this is net delay. As the outputs of **ACAD inverse** is mapped to output pins when running synthesis using **ACAD inverse** as top module, the net delay is most likely unrealistically large. However, this will not be the case for the complete implementation of the ACAD anomaly detector as the output from the **ACAD inverse** module will be

mapped to an internal bus connected to the **dACAD** block. The net delay will therefore be considerably lower. The additional -1.25 ns WNS owing to logic delay may be reduced when running implementation instead of synthesis, as implementation results typically reduce the number of LUTs inferred. Still, it it uncertain if the design will meet timing requirements. If the requirements are not met, registers should be inserted into the critical path, which most likely goes through **Elimination core**. If insertion of registers is necessary, the estimated inverse computation execution times, inv_worst_case and inv_best_case, need to be re-estimated.

6.2.3 Simulation results

The simulations of **Shiftregister**, **ACAD correlation** and **ACAD inverse** proved to be successful, and the designs acts as expected. The simulations done is on a limited range of possible inputs to the designs. Simulations done by the author should act as a proof of concept of the simulated designs.

A wider set of data-inputs and test cases should be created for simulation in order to test the designs more thoroughly. Testing on the Zedboard should be done once the ACAD AD is completed.

Chapter 7

Conclusion

A proposed implementation of the Adaptive Causal anomaly detection (ACAD) algorithm has been made in this thesis. The implementation is to be implemented on the Zynq-Z7030 or the Zynq-Z7035. ACAD was chosen after a comprehensive review of existing anomaly detector (AD) algorithms. The ACAD algorithm has been tested on real and synthetic image data by the author and Chang *et al.* [7], and it shows promising results.

The causality of the ACAD algorithm is beneficial for hardware implementation as it enables real-time anomaly detection. ACAD builds a binary anomaly map of size N_TOT_PIXELS, which it is possible to transmit to a ground station instead of transmitting $\delta^{ACAD}(\mathbf{x}_k)$ of size $N\_TOT\_PIXELS \times Pixel\_data\_width \times 2$, where $Pixel\_data\_width$ is the data width of an input pixel per spectral band. This is advantageous with regards to data transmission as it lowers transmission time and thereby also transmission energy.

The Gauss-Jordan elimination was chosen for implementation of inverse. One of the main drawbacks of this algorithm is the usage of division. Usage of the division operator "/" leads to the design failing to meet timing requirements. Therefore, other approximations have been made, including the adaptive-shifting and LUT approaches. As these approaches are approximations, there might be precision errors leading to errors in the outputted anomaly map from the ACAD.

The proposed implementation is made to be scalable in order to handle large values of spectral bands, P_bands. To be able to read and write two rows of a matrix of size $P\_bands \times P\_bands$ per clock cycle, a parallel memory structure consisting of BRAM-arrays of size P_bands for storage of matrices utilized by the ACAD algorithm is made. Zynq-Z7030 and Zynq-7035 contain enough BRAMs to store the matrices needed in ACAD of sizes 53×53 and 100×100 respectively. These matrices have matrix elements of size $Pixel\_data\_width \times 2$.

The correlation and inverse modules have a large degree of parallelism, computing and updating up to two rows of the correlation and inverse matrix respectively, both of size $P\_bands \times P\_bands$, per clock cycle. This computation is largely done by DSP-blocks for $Pixel\_data\_width$ of 16. For this data width, the correlation module, **ACAD correlation**, infers $P\_bands \times 4$ DSPs. By setting $Pixel\_data\_width = 10$, no DSPs are inferred by **ACAD correlation**. The inverse module, **ACAD inverse**, also utilizes a

high number of DSPs. For $P\_bands = 30$, $Div\_Precision = 17$ and $Pixel\_data\_width$ of 16, **ACAD inverse** utilizes 540 DSPs. As the Z-7030 and Z-7035 have 400 and 900 DSPs respectively, the high number of DSPs utilized by the ACAD AD constrains the value of the parameter P_bands.

One of the main bottlenecks of the processing pipeline in the ACAD is the inverse computation. The estimated worst case execution time per pixel for the inverse computation is $3P\_bands + 2\sum_{i=0}^{P\_bands-1} i$.

The largest uncertainty of the proposed implementation of ACAD is the effect of the approximation to the division operation done in both the inverse and correlation modules. This effect should be heavily tested to investigate whether it is possible to implement ACAD using the approach proposed in this thesis.

7.1 Future work

For future work, the ACAD AD should be completed. To complete it, **dACAD** should be implemented as well as **FSM ACAD**. When the ACAD AD is completed it should be tested on a Zedboard Zynq Evaluation and Development kit.

Verification of the design should be done. As of now, the only form of verification done has been constrained random input simulation on the blocks **Shiftregister**, **ACAD correlation** and **ACAD inverse**. The designs should be further simulated for a wider range of inputs. An automatic test-setup should be made, with a golden reference model, possibly by using MATLAB or other high-level tools or languages.

The consequences of precision errors when doing an approximation to division in both **ACAD correlation** and **ACAD inverse** should be investigated. This must be tested on real hyperspectral data, preferably from the hyperspectral imager used by the SmallSat project.

7.1.1 Optimization

One way of optimizing the ACAD AD is by finding a suited methodology for setting the parameter τ. An option is to set τ based on empirical results. The experiments should contain real hyperspectral image data from coastal areas with algae that are interesting for the SmallSat project. To be able to make a correct anomaly map, the value of τ is important.

Power optimization should be also done. This is especially important as the ACAD AD is to be implemented on an energy-limited satellite. One of the most efficient and easiest power optimization techniques is the usage of clock enable signals for sub-modules in the design, for instance **ACAD inverse**, **ACAD correlation**, **dACAD**. Their respective sub-modules could also have clock enable signals.

Bibliography

[1] What is imaging spectroscopy (hyperspectral imaging)? [Online]. Available: http://www.markelowitz.com/Hyperspectral.html

[2] Øystein Antonsen. Alger drepte trolig 38.000 laks på oppdrettsanlegg. [Online]. Available: https://www.nrk.no/troms/alger-drepte-trolig-38.000-laks-pa-oppdrettsanlegg-1.13633680

[3] NASA. Aviris data- ordering free aviris standard data products. [Online]. Available: https://aviris.jpl.nasa.gov/data/free_data.html

[4] L. Sun. Dataset for classification. [Online]. Available: http://lesun.weebly.com/hyperspectral-data-set.html

[5] M. E. Grøtte. Ntnu smallsat: a hyper-spectral imaging mission. [Online]. Available: https://www.ntnu.edu/documents/20587845/1277298890/022_Gr%C2%A2tte_NTNU_SmallSat_Mission.pdf/f98477c2-4fb3-4fa9-a01d-d732c93b53a6

[6] Xilinx. Zynq-7000. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/imgs/block-diagrams/zynq-mp-core-dual.png

[7] C.-I. Chang and M. Hsueh, "Characterization of anomaly detection in hyperspectral imagery," *Sensor Review*, vol. 26, no. 2, pp. 137–146, 2006.

[8] C. González, S. Bernabé, D. Mozos, and A. Plaza, "Fpga implementation of an algorithm for automatically detecting targets in remotely sensed hyperspectral images," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 9, no. 9, pp. 4334–4343, 2016.

[9] J. M. Molero, E. M. Garzón, I. García, and A. Plaza, "Analysis and optimizations of global and local versions of the rx algorithm for anomaly detection in hyperspectral data," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 6, no. 2, pp. 801–814, 2013.

[10] M. Hsueh, "Reconfigurable computing for algorithms in hyperspectral image processing," 2007.

[11] V. K. Sachan, S. A. Imam, and M. Beg, "Energy-efficient communication methods in wireless sensor networks: A critical review," *International Journal of Computer Applications*, vol. 39, no. 17, 2012.

[12] Xilinx. Zynq-7000 all programmable soc data sheet: Overview(ds190). [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[13] B. Yang, M. Yang, A. Plaza, L. Gao, and B. Zhang, "Dual-mode fpga implementation of target and anomaly detection algorithms for real-time hyperspectral imaging,"

IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, vol. 8, no. 6, pp. 2950–2961, 2015.

[14] E.-U. S. E. P. Agency. Climate change and harmful algal blooms. [Online]. Available: https://www.epa.gov/nutrientpollution/climate-change-and-harmful-algal-blooms

[15] NASA. Aviris airborne visible infrared imaging spectrometer. [Online]. Available: https://aviris.jpl.nasa.gov/index.html

[16] J. Fjelltvedt, "Direct memory access for hyperspectral imaging applications."

[17] I. S. Reed and X. Yu, "Adaptive multiple-band cfar detection of an optical pattern with unknown spectral distribution," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 38, no. 10, pp. 1760–1770, 1990.

[18] M. Karkooti, J. R. Cavallaro, and C. Dick, "Fpga implementation of matrix inversion using qrd-rls algorithm," in *Asilomar Conference on Signals, Systems, and Computers*, 2005.

[19] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. ACM, 1998, pp. 191–200.

[20] D. Kun. Matlab hyperspectral toolbox. [Online]. Available: https://github.com/davidkun/HyperSpectralToolbox

[21] M. Haukali. Matlab hyperspectral toolbox fork. [Online]. Available: https://github.com/marthauk/HyperSpectralToolbox/tree/dev

[22] G. D. intelegencia Computacional. Hyperspectral remote sensing scenes. [Online]. Available: http://lesun.weebly.com/hyperspectral-data-set.html

[23] Xilinx. Vivado design suite 7 series fpga and zynq-7000 all programmable soc libraries guide(ug953). [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug953-vivado-7series-libraries.pdf

[24] S. VHDL. How to implement division in vhdl. [Online]. Available: http://surf-vhdl.com/how-to-implement-division-in-vhdl/

[25] Xilinx. Overutiliznng dsps. [Online]. Available: https://forums.xilinx.com/t5/Implementation/LUT-and-DSP-Utilization/td-p/692728

Appendices

Appendix A

MATLAB hyperspectral

A.1 High level models of algorithms

A.1.1 Gauss-Jordan elimination

Listing A.1: Gauss Jordan inverse

```
function [A_inv,A_mode_elim,A_mode_elim_inv ] =
    gauss_jordan_inverse(A,mode)

% This function implements the Gauss-Jordan method for
    calculating inverse of a square matrix.
% It acts as a high level model for later implementation in
    hardware.
%   Detailed explanation goes here
% USAGE:
% Inputs:
% A         -   Matrix of size p x p
% size_p    -   column size
% Outputs:
% A_inv     -   inverse matrix
[size_p,m]=size(A);
A_inv = eye(size_p);

% Forward elimination to build an upper triangular matrix
if(strcmp(mode,'forward') | strcmp(mode,'all'))
 for (i=1:1:size_p)
   if (A(i,i) == 0)
  for (j =i+1:1:size_p)
          if (A(j,j)~=0)
              % The operations below will be different in
                  hardware, because
              % of parallell operations
              %temp_i = row(i);
```

```
24                  %row(i) = row(j);
25                  %row(j) = temp_i;
26
27                  temp_i = A(i,:);
28                  A(i,:) = A(j,:);
29                  A(j,:) = temp_i;
30
31              end
32          end
33      end
34      if (A(i,i) ==0)
35  %         error('Matrix is singular');
36      end
37      for (j = i +1:1: size_p)
38          % The operations below will be different in hardware,
                because
39          % of parallell operations
40          A_j_i_temp =A(j,i);
41          A_i_i_temp = A(i,i);
42          %A(j,:) = A(j,:)- A(i,:)*A_j_i_temp/A_i_i_temp;
43          %A_inv(j,:) = A_inv(j,:) - A_inv(i,:)*A_j_i_temp/
                A_i_i_temp;
44          for (l= 1:size_p)
45           A(j,l) = A(j,l)- A(i,l)*A_j_i_temp/A_i_i_temp;
46          A_inv(j,l) = A_inv(j,l) - A_inv(i,l)*A_j_i_temp/
                A_i_i_temp;
47          end
48      end
49  end
50  end
51
52  if (strcmp(mode,'forward'))
53       A_mode_elim = A;
54       A_mode_elim_inv = A_inv;
55  end
56
57  % Backward elimination to build a diagonal matrix
58  if(strcmp(mode,'backward') | strcmp(mode,'all'))
59       for(i=size_p:-1:2)
60      for( j=i-1:-1: 1)
61          % The operations below will be different in hardware,
                because
62          % of parallell operations
63          A_j_i_temp =A(j,i);
64          A_i_i_temp = A(i,i);
65          %A(j,:) = A(j,:)-A(i,:)*cast(cast(A(j,i)/A(i,i),'int32')
                ,'double');
66          %A_inv(j,:) = A_inv(j,:) - A_inv(i,:)*cast(cast(
                A_j_i_temp/A_i_i_temp,'int32'),'double');
67
```

```
68          %A(j,:) = A(j,:)-A(i,:)*A(j,i)/A(i,i);
69          %A_inv(j,:) = A_inv(j,:) - A_inv(i,:)*A_j_i_temp/
                A_i_i_temp;
70
71          for (k=1:size_p)
72            A(j,k) = A(j,k)-A(i,k)*A(j,i)/A(i,i);
73            A_inv(j,k) = A_inv(j,k) - A_inv(i,k)*A_j_i_temp/
                  A_i_i_temp;
74          end
75      end
76  end
77
78  end
79  if (strcmp(mode,'backward'))
80  A_mode_elim_inv = A_inv;
81  A_mode_elim = A;
82  end
83  if(strcmp(mode,'identity'))
84       A_mode_elim_inv = zeros(3);
85       A_mode_elim = zeros(3);
86  end
87  if(strcmp(mode,'all'))
88       A_mode_elim_inv = zeros(3);
89       A_mode_elim = zeros(3);
90  end
91
92  % Last division to build an identity matrix
93  for ( i = 1:+1:size_p)
94       A_inv(i,:)= A_inv(i,:)*1/A(i,i);
95
96  end
97
98
99
100
101
102 end
```

A.1.2 RX anomaly detector

Listing A.2: RX AD

```
1  function [result, sigma, sigmaInv] = hyperRxDetector(M)
2  %HYPERRX RX anomaly detector
3  %   hyperRxDetector performs the RX anomaly detector
4  %
5  % Usage
6  %   [result] = hyperRxDetector(M)
7  % Inputs
8  %   M - 2D data matrix (p x N)
```

```
9  % Outputs
10 %    result - Detector output (1 x N)
11 %    sigma - Covariance matrix (p x p)
12 %    sigmaInv - Inverse of covariance matrix (p x p)
13
14 % Remove the data mean
15 [p, N] = size(M);
16 mMean = mean(M, 2);
17 M = M - repmat(mMean, 1, N);
18
19 % Compute covariance matrix
20 sigma = hyperCov(M);
21 sigmaInv = inv(sigma);
22
23 result = zeros(N, 1);
24 for i=1:N
25     result(i) = M(:,i).'*sigmaInv*M(:,i);
26 end
27 result = abs(result);
28
29 return;
```

A.1.3 LRX anomaly detector

Listing A.3: LRX AD

```
1  function [result, autocorr, sigmaInv] = hyperLRxDetectorCorr(M,K
       )
2  %HYPERRX LRX anomaly detector
3  %    hyperLRxDetector performs the Local RX anomaly detector
       using Correlation
4  %    instead of covariance
5  %
6  % Usage
7  %    [result] = hyperRxDetector(M)
8  % Inputs
9  %    M - 2D data matrix (p x N)
10 %    K - Size of the kernel window, K x K
11 % Outputs
12 %    result - Detector output (1 x N)
13 %    sigma - Correlation matrix (p x p)
14 %    sigmaInv - Inverse of correlation matrix (p x p)
15
16 [p, N] = size(M);
17
18
19 % Compute correlation matrix of size K
20 % correlation matrix will be of size p x p
21 result = zeros(N, 1);
22
```

```
23      h = waitbar(0,'Initializing waitbar ..');
24      for j=1:N
25          autocorr = hyperCorrK(M,K,j);
26         % M_inv = gauss_jordan_inverse(autocorr,'all');
27          result(j) = M(:,j).' * pinv(autocorr) * M(:,j);
28          %result(j) = M(:,j).' * M_inv * M(:,j);
29
30        waitbar(j/N,h,'Updated LRX progress');
31
32      end
33
34  return;
```

A.1.4 ALRX anomaly detector

Listing A.4: ALRX AD

```
1  function [result, anomaly_map,location_of_anomalies,
       last_local_anomalies_set] = hyperLRX_anomaly_set_remover(M,K,
       treshold)
2  % LRX anomaly detector, that also removes the detected anomalous
       targets
3  % causaly.
4  %   hyperLRxDetector performs the Local RX anomaly detector
       using Correlation
5  %   instead of covariance
6  %
7  % Usage
8  %   [result] = hyperRxDetector(M)
9  % Inputs
10 %   M - 2D data matrix (p x N)
11 %   K - Size of the kernel window, K x K
12 % Outputs
13 %   result - Detector output (1 x N)
14
15 [p, N] = size(M);
16 %waitbar for progress monitoring
17 h = waitbar(0,'Initializing waitbar ..');
18
19 % Compute correlation matrix of size K
20 % correlation matrix will be of size p x p
21 result = zeros(N, 1);
22 anomaly_map = zeros(N,1);
23 anomalies_detected=zeros(p,N/2);
24 anomalies_detected_transpose_sum = zeros(p,p);
25 %tresh_LRX = 6.0000e+14;
26 tresh_LRX=treshold;
27 location_of_anomalies= zeros(N/2,1);
28
29 local_anomalies_set=0;
```

```
30 last_local_anomalies_set =0;
31 ROWS=100;
32 t_an=1;
33
34
35 flag_local_anomaly_found =0;
36     for j=1:N
37         autocorr = hyperCorrK(M,K,p);
38          %adaptive_autocorr_inv = inv(autocorr -
                anomalies_detected_transpose_sum);
39          adaptive_autocorr_inv = pinv(autocorr -
                local_anomalies_set);
40 %result(j) = M(:,i).' * autocorrInv;
41
42         result(j) = M(:,j).' * adaptive_autocorr_inv * M(:,j);
43         if result(j) > tresh_LRX
44             % This pixel is an anomaly! Add it to the set of
                  anomalies
45             anomalies_detected(:,t_an) = M(:,j);
46             anomaly_map(j)=1;
47             location_of_anomalies(t_an)=j;
48             anomalies_detected_transpose_sum = M(:,j)* M(:,j).'
                   + anomalies_detected_transpose_sum;
49             t_an = t_an + 1;
50         end
51         %if anomalies_detected_transpose_sum contains elements
               from outside
52         %the KERNEL
53
54         lower_limit_matrix = j - floor(K/2);
55         higher_limit_matrix = j + floor(K/2);
56
57         % Check if index is out of bounds
58         if( lower_limit_matrix < 1)
59             % for edges of the matrix, gonna assume that we just
                  throw out points
60             % outside of the edge, and use half the KERNEL
61             lower_limit_matrix = 1;
62         end
63         if ( higher_limit_matrix > N)
64           % M(band, neighbouring_pixels) * (M(band, Neighbouring
                pixels)
65           higher_limit_matrix = N;
66         end
67
68         if(any(local_anomalies_set))
69             %just to check that it works
70             last_local_anomalies_set = local_anomalies_set;
71         end
72         %resetting local_anomalies_set before using it the next
```

```
            iteration
73          local_anomalies_set = 0;
74          flag_local_anomaly_found =0;
75          for i=1:t_an
76              if flag_local_anomaly_found == 0
77                  for k=1:K
78                      if(flag_local_anomaly_found==0)
79                          if(location_of_anomalies(i) >
                                lower_limit_matrix+(k-1)*ROWS &
                                location_of_anomalies(i)<
                                higher_limit_matrix+(k-1)*ROWS)
80                              local_anomalies_set =
                                    local_anomalies_set +
                                    anomalies_detected(:,i)*
                                    anomalies_detected(:,i).';
81                              flag_local_anomaly_found =1;
82                              break;
83                          end
84                      end
85                  end
86              end
87              %if location_of_anomalies(t_an)<j-floor(K/2) |
                    location_of_anomalies(t_an)>j+floor(K/2)
88              %     anomalies_detected_transpose_sum =
                    anomalies_detected_transpose_sum - M(:,t_an)*M(:,
                    t_an).';
89              %end
90          end
91          %result(j)= result(j) *  M(:,i);
92          waitbar(j/N,h,'Updated progress');
93      end
94
95
96  result = abs(result);
```

A.1.5 ACAD anomaly detector

Listing A.5: ACAD

```
1  function [d_acad, anomaly_map,threshold_check_values] =
       hyperACAD(M, tresh)
2  %HYPERRX Adaptive Causal Anomaly detector
3  %   hyperLRxDetector performs the Adaptive Causal detector using
4  %   correlation matrix
5  % It is adaptive in the sense that it removes the previously
       detected
6  % anomalies from the correlation set
7
8  % Usage
9  %   [result] = hyperACAD(M)
```

```
10 % Inputs
11 %   M - 2D data matrix (p x N)
12 %   tresh - Treshold for a pixel to be considered anomaly
13 % Outputs
14 %   d_acad - Detector output (1 x N)
15 %   anomalies_detected - (1 x t_an)
16
17
18
19
20 % t_an is the number of anomalies detected. Since MatLab is 1-
       index, T is
21 % initially set to 1, not 0.
22 t_an=1;
23
24 % bheta is the ratio of the entire image size to the size of
       anomaly
25 bheta = 100;
26 %bheta = 50;
27
28 % p is number of spectral bands, N is number of pixels
29 [p, N] = size(M);
30
31 % anomalies_detected is the growing set of anomalies detected in
       the image.
32 % Numbers of anomalies will not exceed N/2. Even that is way to
       much.
33 % Starting point N/2. Need to include the pixel it was found, j.
       Make some
34 % kind of map
35 anomalies_detected=zeros(p,N/2);
36
37 % anomalies_detected_transpose_sum is the sum of the transposes
       taken on
38 % anomalous pixels
39 anomalies_detected_transpose_sum = zeros(p,p);
40
41 % n_acad is used in the process of setting the threshold for
       finding an
42 % anomaly
43 n_acad = (N/bheta);
44
45 % u_k is the expected value/causal mean in the image. Initial
       value is set
46 % to the first pixel..This is wrong!!
47 %u_k = M(:,1);
48 u_k =0;
49 % tresh is the treshold value used to consider if the pixel is
       an anomaly
50 % or not. I think that it the anomaly detection will be
```

```
    normalized...(???)
% Grubbs test for setting treshold?
%tresh = 50;


% d_acad is the result of Adapative Causal anomaly detection
d_acad = zeros(N, 1);


%waitbar for progress monitoring
h = waitbar(0,'Initializing waitbar ..');

% Since this is causal, it is useful to have the value
    prev_autocorr
prev_autocorr = 0;

% Causality
prev_u_k = 0;

threshold_check_values = zeros(N,1);

%
location_of_anomalies= zeros(N/2,1);

% for all N= m x n pixels
for j=1:N
        % want to store the result of hyperCausalCorr, in case
            this pixel isan
        % anomaly. In that case we need to subtract it from the
            set.
         autocorr = prev_autocorr + hyperCausalCorr(M,j);
         prev_autocorr = autocorr;
         % Normalizing
         %autocorr = autocorr/j;
         % Since anomalies_detected_transpose is firstly
            initialized to
         % zero, this will sum N/2 elements being zero. This is
            not
         % necessary, and will cost computation time. Find fix
         %adaptive_autocorr_inv = inv(autocorr -
            anomalies_detected_transpose_sum);
         %adaptive_autocorr_inv = gauss_jordan_inverse(autocorr -
             anomalies_detected_transpose_sum,'all');
%          if(j>floor(n_acad))
%              adaptive_autocorr_inv = gauss_jordan_inverse((
    autocorr - anomalies_detected_transpose_sum)/(n_acad-t_an),'
    all');
%          else
%              adaptive_autocorr_inv = gauss_jordan_inverse((
    autocorr - anomalies_detected_transpose_sum)/(j-t_an),'all');
```

```
90 %           end
91 %
92 %
93  if(j>floor(n_acad))
94      adaptive_autocorr_inv = pinv((autocorr -
            anomalies_detected_transpose_sum))/(n_acad-t_an));
95  else
96              adaptive_autocorr_inv = pinv((autocorr -
                    anomalies_detected_transpose_sum))/(j-t_an));
97          end
98
99
100
101         %temp_acad = M(:,j).' * adaptive_autocorr_inv * M(:,j);
102         d_acad(j)= M(:,j).' * adaptive_autocorr_inv * M(:,j);
103
104         if(j>floor(n_acad))
105             u_k_un_normalized = prev_u_k + d_acad(j) - d_acad(j-
                    floor(n_acad));
106             %u_k_un_normalized = sum(d_acad(j-n_acad:j));
107         else
108             u_k_un_normalized = prev_u_k + d_acad(j);
109         end
110         u_k  = (1/n_acad) * u_k_un_normalized;
111         %u_k = abs(u_k);
112
113         %disp(d_acad(j)-u_k);
114         threshold_check_values(j) = d_acad(j)-u_k;
115         %if (abs(d_acad(j) - u_k)) > tresh
116         if ((d_acad(j) - u_k)) > tresh
117             % This pixel is an anomaly! Add it to the set of
                    anomalies
118             anomalies_detected(:,t_an) = M(:,j);
119             location_of_anomalies(t_an)=j;
120             anomalies_detected_transpose_sum = M(:,j)* M(:,j).'
                    + anomalies_detected_transpose_sum;
121             t_an = t_an + 1;
122         end
123         prev_u_k = u_k_un_normalized;
124         waitbar(j/N,h,'Updated progress ACAD');
125 end
126 anomaly_map= zeros(1,N);
127 for i=1:1:N/2
128     if (anomalies_detected(1,i)~= 0)
129     pixel_pos_anomaly = location_of_anomalies(i);
130     anomaly_map(pixel_pos_anomaly) = 1;
131     end
132 end
133
134
```

```
135
136
137 return;
```

A.2 Testing

A.2.1 Hyper demo detectors

Listing A.6: Hyper Demo Detector

```
1  function hyperDemo_detectors
2  % HYPERDEMO_DETECTORS Demonstrates target detector algorithms
3  clear; clc; dbstop if error; close all;
4  %
       ------------------------------------------------------------------------

5  %% Parameters
6  %resultsDir ='E:\One Drive\OneDrive for Business\NTNU\Master\
       Forked_MATLAB_hyperspectral_toolbox\
       MATLAB_Hyperspectral_toolbox\results';
7  %dataDir = 'E:\One Drive\OneDrive for Business\NTNU\Master\
       Forked_MATLAB_hyperspectral_toolbox\MATLAB_DEMO_hyperspectral
       \f970619t01p02r02c';
8
9  resultsDir =['E:\One Drive\OneDrive for Business\NTNU\Master\
       Anomaly detection results\MATLAB\LRX\real image data Cuprite
       scene\',datestr(now, 'dd-mmm-yyyy')];
10 dataDir = 'E:\One Drive\OneDrive for Business\NTNU\Master\
       MATLAB_DEMO_hyperspectral\f970619t01p02r02c';
11 %
       ------------------------------------------------------------------------

12
13 mkdir(resultsDir);
14
15 %% Read part of AVIRIS data file that we will further process
16 M = hyperReadAvirisRfl(sprintf('%s\\f970619t01p02_r02_sc02.a.rfl
       ', dataDir), [1 100], [1 614], [1 224]);
17 %M = hyperReadAvirisRfl(sprintf('%s\\f970619t01p02_r02_sc04.a.
       rfl', dataDir), [1 100], [1 614], [1 224]);
18
19 M = hyperNormalize(M);
20 %% Read AVIRIS .spc file
21 lambdasNm = hyperReadAvirisSpc(sprintf('%s\\f970619t01p02_r02.a.
       spc', dataDir));
22
23 %% Isomorph
24 [h, w, p] = size(M);
25 M = hyperConvert2d(M);
26 %KSC_2d = hyperConvert2d(KSC);
```

```
27 %M=KSC_2d;
28 %% Resample AVIRIS image.
29 desiredLambdasNm = 400:(2400-400)/(224-1):2400;
30 M = hyperResample(M, lambdasNm, desiredLambdasNm);
31
32 %% Remove low SNR bands.
33 goodBands = [10:100 116:150 180:216]; % for AVIRIS with 224
       channels
34 %goodbands_KSC =[10:100 116:150];
35
36 %KSC_2d = KSC_2d(goodbands_KSC,:);
37 %p = length(goodbands_KSC);
38 M = M(goodBands, :);
39 p = length(goodBands);
40
41 %% Demonstrate difference spectral similarity measurements
42 M = hyperConvert3d(M, h, w, p);
43 target = squeeze(M(11, 77, :));
44 figure; plot(desiredLambdasNm(goodBands), target); grid on;
45       title('Target Signature; Pixel (32, 257)');
46
47 M = hyperConvert2d(M);
48
49 %% RX Anomly Detector
50 %r = hyperRxDetector(M);
51 %r = hyperRxDetectorCor(M);
52 K=23;
53 resultsDir =['E:\One Drive\OneDrive for Business\NTNU\Master\
       Anomaly detection results\MATLAB\LRX\real image data Cuprite
       scene\' ,datestr(now, 'dd-mmm-yyyy')];
54 %r = hyperLRxDetectorCorr(M,K);
55 %g = ground_truth(h,614, M, M_endmembers);
56 %figure; imagesc(g);colorbar;
57 treshold = 500;
58
59
60 for treshold= 500: 250 :2000
61 [r,anomalies_detected, location_of_anomalies,
       last_local_anomalies_set]=hyperLRX_anomaly_set_remover(M,K,
       treshold);
62 [r,anomalies_detected, location_of_anomalies,
       last_local_anomalies_set]=hyperLRX_anomaly_set_remover(KSC_2d
       ,K,treshold);
63
64
65
66 figure; imagesc(r); title(['ALRX Detector Results.K=23, tresh ='
       num2str(treshold) '.'] ); axis image;
67       colorbar;
68  hyperSaveFigure(gcf, sprintf(['%s\\ALRX Detector Results.K=23,
```

```
      tresh' num2str(treshold) '.png' ], resultsDir));%
69
70 %figure; imagesc(r); title(['LRX removing anomalies,tresh =2000,
       K=25 .'] ); axis image;
71 %      colorbar;
72 %  figure; imagesc(r); title(['LRX Cuprite image data sc02 K='
       num2str(K) '.'] ); axis image;
73 %      colorbar;
74 % hyperSaveFigure(gcf, sprintf(['%s\\LRX_K=25_treshold_500_KSC'
       num2str(treshold) '.png' ], resultsDir));%
75 %
76   anomaly_map = hyperConvert3d(anomaly_map.', h, w, 1);
77 figure; imagesc(anomaly_map); title(['Anomaly map ACAD, using
       LUTs, treshold = ' num2str(treshold) '.'] ); axis image;
78      colorbar;
79  hyperSaveFigure(gcf, sprintf(['%s\\Anomaly Map ACAD using LUTs.
        treshold= ' num2str(treshold) '.png' ], resultsDir));%
80
81
82
83 end
84
85 %% Constrained Energy Minimization (CEM)
86 r = hyperCem(M, target);
87 r = hyperConvert3d(r, h, w, 1);
88 figure; imagesc(abs(r)); title('CEM Detector Results'); axis
       image;
89      colorbar;
90 hyperSaveFigure(gcf, sprintf('%s\\cem detector.png', resultsDir)
       );
91
92 %% Adaptive Cosine Estimator (ACE)
93 r = hyperAce(M, target);
94 r = hyperConvert3d(r, h, w, 1);
95 figure; imagesc(r); title('ACE Detector Results'); axis image;
96      colorbar;
97 hyperSaveFigure(gcf, sprintf('%s\\ace detector.png', resultsDir)
       );
98
99 %% Signed Adaptive Cosine Estimator (S-ACE)
100 r = hyperSignedAce(M, target);
101 r = hyperConvert3d(r, h, w, 1);
102 figure; imagesc(r); title('Signed ACE Detector Results'); axis
       image;
103      colorbar;
104 hyperSaveFigure(gcf, sprintf('%s\\signed ace detector.png',
       resultsDir));
105
106 %% Matched Filter
107 r = hyperMatchedFilter(M, target);
```

```
108 r = hyperConvert3d(r, h, w, 1);
109 figure; imagesc(r); title('MF Detector Results'); axis image;
110     colorbar;
111 hyperSaveFigure(gcf, sprintf('%s\\mf detector.png', resultsDir))
        ;
112
113 %% Generalized Likehood Ratio Test (GLRT) detector
114 r = hyperGlrt(M, target);
115 r = hyperConvert3d(r, h, w, 1);
116 figure; imagesc(r); title('GLRT Detector Results'); axis image;
117     colorbar;
118 hyperSaveFigure(gcf, sprintf('%s\\cem detector.png', resultsDir)
        );
119
120
121 %% Estimate background endmembers
122 U = hyperAtgp(M, 5);
123
124 %% Hybrid Unstructured Detector (HUD)
125 r = hyperHud(M, U, target);
126 r = hyperConvert3d(r, h, w, 1);
127 figure; imagesc(abs(r)); title('HUD Detector Results'); axis
        image;
128     colorbar;
129 hyperSaveFigure(gcf, sprintf('%s\\hud detector.png', resultsDir)
        );
130
131 %% Adaptive Matched Subspace Detector (AMSD)
132 r = hyperAmsd(M, U, target);
133 r = hyperConvert3d(r, h, w, 1);
134 figure; imagesc(abs(r)); title('AMSD Detector Results'); axis
        image;
135     colorbar;
136 hyperSaveFigure(gcf, sprintf('%s\\amsd detector.png', resultsDir
        ));
137 figure; mesh(r); title('AMSD Detector Results');
138
139 %% Orthogonal Subspace Projection (OSP)
140 r = hyperOsp(M, U, target);
141 r = hyperConvert3d(r, h, w, 1);
142 figure; imagesc(abs(r)); title('OSP Detector Results'); axis
        image;
143     colorbar;
144 hyperSaveFigure(gcf, sprintf('%s\\osp detector.png', resultsDir)
        );
```

A.2.2 Generating synthetic images

Listing A.7: Synthetic image 30 30

```
clc; clear;close all;
%generating random image based on cuprite scene data
h=30;
w= 30;
%load('E:\One Drive\OneDrive for Business\NTNU\Master\
    ground_truthing_aviris_cuprite\cuprite\
    groundTruth_Cuprite_end12\groundTruth_Cuprite_nEnd12.mat','-
    mat');
load('groundTruth_Cuprite_nEnd12.mat','-mat');
M_endmembers=M;
goodBands = [10:100 116:150 180:216]; % for AVIRIS with 224
    channels
M_endmembers=M(goodBands,:);
[n_bands,k] = size(M_endmembers);
image_30_30 = zeros(30,30,n_bands);
reference_anomaly_map = zeros(30,30);
n_true_anomalies =4;
% Setting background
for i=1:h
    for j=1:w
        dice = randi(6);
        if dice>4
            image_30_30(i,j,:)= M_endmembers(:,1); %setting
                background to alunite
        elseif dice>2
            image_30_30(i,j,:)= M_endmembers(:,6); %setting
                background to Kalonite
        else
            image_30_30(i,j,:)= M_endmembers(:,10);% setting
                background to pyrope
        end
%           rN = rand;
%           image_30_30(i,j,:)= rN * M_endmembers(:,1) + 0.25*
    M_endmembers(:,3)+0.25* M_endmembers(:,6) +(1-rN)*
    M_endmembers(:,8);

%         rN= rand;
%         image(i,j,:) = rN*M_endmembers(:,1) +0.2*M_endmembers
    (:,3)+0.2*M_endmembers(:,4)+0.2*M_endmembers(:,7)+rN*
    M_endmembers(:,12);

    end
end


%create kernels with anomalies of size 2x2 with bottom left
    pixel in 15,15
%column locations

KERNEL_SIZE_TWO_LOCATION = 15;
```

```
39
40 image_30_30(KERNEL_SIZE_TWO_LOCATION,KERNEL_SIZE_TWO_LOCATION,:)
       = M_endmembers(:,3);
41 reference_anomaly_map(KERNEL_SIZE_TWO_LOCATION+1,
       KERNEL_SIZE_TWO_LOCATION)=1;
42 reference_anomaly_map(KERNEL_SIZE_TWO_LOCATION,
       KERNEL_SIZE_TWO_LOCATION)=1;
43 reference_anomaly_map(KERNEL_SIZE_TWO_LOCATION+1,
       KERNEL_SIZE_TWO_LOCATION+1)=1;
44 reference_anomaly_map(KERNEL_SIZE_TWO_LOCATION,
       KERNEL_SIZE_TWO_LOCATION+1)=1;
45
46 image_30_30(KERNEL_SIZE_TWO_LOCATION+1,KERNEL_SIZE_TWO_LOCATION
       ,:)= M_endmembers(:,3);
47 image_30_30(KERNEL_SIZE_TWO_LOCATION,KERNEL_SIZE_TWO_LOCATION
       +1,:)= M_endmembers(:,3);
48 image_30_30(KERNEL_SIZE_TWO_LOCATION+1,KERNEL_SIZE_TWO_LOCATION
       +1,:)= M_endmembers(:,3);
49
50 imnoise(image_30_30,'gaussian',1);
51 matrix=hyperConvert2d(image_30_30);
52 %[d_acad, anomaly_map,threshold_check_values] = hyperACAD(matrix
       ,100);
53 % K is size of kernel
54 K=5;
55 treshold = 0.9;
56 %[r_alrx,anomaly_map,not_used ,not_use] =
       hyperLRX_anomaly_set_remover(matrix,K,treshold);
57 [r_alrx,anomaly_map,not_used ] =hyperACAD(matrix,treshold);
58
59 %d_acad_2d = hyperConvert3d(d_acad.', 30, 30, 1);
60 r_alrx_2d = hyperConvert3d(r_alrx.', 30, 30, 1);
61 anomaly_map_2d = hyperConvert3d(anomaly_map.', 30, 30, 1);
62
63 %figure;imagesc(r_alrx_2d);title(['ALRX AD detector, K= '
       num2str(K) ]); axis image; colorbar;
64 figure;imagesc(r_alrx_2d);title(['ACAD result, treshold' num2str
       (treshold) ]); axis image; colorbar;
65
66 figure;imagesc(anomaly_map_2d);title(['ACAD anomaly map,
       treshold= ' num2str(treshold) ]); axis image; colorbar;
67
68 %% Evaluate the performance of the AD by setting objective
       measures
69 % find max value outputted from the AD
70 %max_ad_score = max(r_rlx);
71 treshold_percentage = 0.75;
72 predicted_anomalies =0;
73 % for i=1:w*h
74 %     if r_rlx(i)>=treshold_percentage *max_ad_score
```

```
%           predicted_anomalies =predicted_anomalies+1;
%       end
% end
n_actual_anomalies=n_true_anomalies;
n_true_anomalies =0;
 for i=1:w
      for j=1:h
     if anomaly_map_2d(i,j)==1
          predicted_anomalies =predicted_anomalies+1;
          if reference_anomaly_map(i,j)==1
           n_true_anomalies=n_true_anomalies+1;
          end
     end
     end
end

false_anomalies = predicted_anomalies-n_true_anomalies;
if predicted_anomalies<n_actual_anomalies
     correctly_predicted_anomalies =n_true_anomalies/
         n_actual_anomalies;
else
correctly_predicted_anomalies =n_true_anomalies/
    predicted_anomalies;
end


%figure;imagesc(d_acad_2d); axis image; colorbar;

%figure;imagesc(anomaly_map_2d); axis image; colorbar;

%figure;imagesc(reference_anomaly_map);axis image; colorbar;
```

Listing A.8: Synthetic image 100x614

```
%for Cuprite scene
clc; close all; clear;
h=100;
w= 614;
load('groundTruth_Cuprite_nEnd12.mat','-mat');
M_endmembers=M;
goodBands = [10:100 116:150 180:216]; % for AVIRIS with 224
    channels
M_endmembers=M(goodBands,:);
[n_bands,k] = size(M_endmembers);
image = zeros(h,w,n_bands);
reference_anomaly_map = zeros(h,w);
% Setting background
for i=1:h
     for j=1:w
          dice = randi(6);
```

```
16          if dice>4
17              image(i,j,:)= M_endmembers(:,1); %setting background
                    to alunite
18          elseif dice>2
19              image(i,j,:)= M_endmembers(:,6); %setting background
                    to Kalonite
20          else
21              image(i,j,:)= M_endmembers(:,10);% setting
                    background to pyrope
22          end
23 %            rN=rand;
24 %            image(i,j,:) = rN*M_endmembers(:,1) +0.2*M_endmembers
       (:,3)+0.2*M_endmembers(:,4)+0.2*M_endmembers(:,7)+(1-rN)*
       M_endmembers(:,12);
25        end
26 end
27
28 %imnoise(image,'gaussian',1);
29
30
31 %setting 50 random pixels to be an anomaly
32 % for i=1: 50
33 %      h_index=randi(h);
34 %      w_index= randi(w);
35 %      signature_index = randi([2 12]);
36 %      image(:,h_index,w_index) = M_endmembers(:,signature_index)
       ;
37 %      anomaly_map(h_index,w_index)=1;
38 % end
39
40 %create kernels with anomalies of size 1, 5, 10,15, 20, 25 in
       columns 5, 20,50,100, 400,
41 %600, in row 35 and 70
42 %column locations
43 KERNEL_SIZE_ONE_LOCATION =50;
44 KERNEL_SIZE_TWO_LOCATION = 100;
45 KERNEL_SIZE_FIVE_LOCATION =150;
46 KERNEL_SIZE_TEN_LOCATION =250;
47 KERNEL_SIZE_FIFTEEN_LOCATION =350;
48 KERNEL_SIZE_TWENTY_LOCATION =450;
49 KERNEL_SIZE_TWENTYFIVE_LOCATION =550;
50 for i=1:h
51      if(mod(i,35)==0)
52          image(i,KERNEL_SIZE_ONE_LOCATION,:) = M_endmembers(:,3);
53          reference_anomaly_map(i,KERNEL_SIZE_ONE_LOCATION)=1;
54
55          image(i,KERNEL_SIZE_TWO_LOCATION,:) = M_endmembers(:,3);
56          reference_anomaly_map(i,KERNEL_SIZE_TWO_LOCATION)=1;
57
58          image(i,KERNEL_SIZE_TWO_LOCATION +1,:) = M_endmembers
```

```
            (:,3);
59          reference_anomaly_map(i,KERNEL_SIZE_TWO_LOCATION +1)=1;
60
61          image(i+1,KERNEL_SIZE_TWO_LOCATION,:) = M_endmembers
                (:,3);
62          reference_anomaly_map(i+1,KERNEL_SIZE_TWO_LOCATION)=1;
63
64          image(i+1,KERNEL_SIZE_TWO_LOCATION +1,:) = M_endmembers
                (:,3);
65          reference_anomaly_map(i+1,KERNEL_SIZE_TWO_LOCATION +1)
                =1;
66          for(j=5:5:25)
67              for row =i-floor(j/2):i+floor(j/2)
68                  switch j
69                      case 5
70                          colcenter =KERNEL_SIZE_FIVE_LOCATION;
71                      case 10
72                          colcenter =KERNEL_SIZE_TEN_LOCATION;
73                      case 15
74                          colcenter =
                                KERNEL_SIZE_FIFTEEN_LOCATION;
75                      case 20
76                          colcenter =KERNEL_SIZE_TWENTY_LOCATION
                                ;
77                      case 25
78                          colcenter =
                                KERNEL_SIZE_TWENTYFIVE_LOCATION;
79                  end
80                  for col =colcenter-floor(j/2):colcenter+floor(j
                        /2)
81                      image(row,col,:) = M_endmembers(:,3);
82                      reference_anomaly_map(row,col)=1;
83                  end
84              end
85          end
86      end
87  end
88  imnoise(image,'gaussian',1);
89
90
91
92   matrix=hyperConvert2d(image);
93
94
95  % r_rx = hyperRxDetector(matrix);
96  % r_rx_2d = hyperConvert3d(r_rx.', h, w, 1);
97  % figure; imagesc(reference_anomaly_map); title(['Expected
        anomaly map'] ); axis image; colorbar;
98  % figure; imagesc(r_rx_2d); title(['RX AD results '] ); axis
        image; colorbar;
```

```
99  %
100 % max_value_rx= max(r_rx);
101 % % set 75% of max_value as an anomaly
102 % treshold_rx = max_value_rx*0.75;
103 % anomaly_map_rx=zeros(h,w);
104 % for i=1:h
105 %     for j=1:w
106 %         if r_rx_2d(i,j) >=treshold_rx
107 %             anomaly_map_rx(i,j)=1;
108 %         end
109 %     end
110 % end
111 % figure; imagesc(anomaly_map_rx); title(['RX anomaly map'] );
      axis image; colorbar;
112 % %check difference RX-anomaly_map and reference anomaly map
113 % difference_from_reference_rx = zeros(h,w);
114 % false_anomalies_hsueh_rx=nnz(anomaly_map_rx) - nnz(
      reference_anomaly_map);
115 % if(false_anomalies_hsueh_rx<0)
116 %     false_anomalies_hsueh_rx=0;
117 % end
118 %
119 % for i=1:h
120 %     for j=1:w
121 %         difference_from_reference_rx(i,j)= (
      reference_anomaly_map(i,j)- anomaly_map_rx(i,j));
122 %     end
123 % end
124 % figure; imagesc(difference_from_reference_rx); title(['false
      or undetected anomalies RX'] ); axis image; colorbar;
125 % %hyperSaveFigure(gcf, sprintf(['%s\\Undetected anomalies RX'
      '.png' ], resultsDir));%
126 %
127 % nnz_rx = nnz(difference_from_reference_rx);
128 % percent_predicted_anomalies_hsueh_rx = (nnz(anomaly_map_rx) -
      false_anomalies_hsueh_rx)/nnz(reference_anomaly_map);
129
130
131 % K=25;
132 % r_rlx =hyperLRxDetectorCorr(matrix,K);
133 % r_rlx_2d = hyperConvert3d(r_rlx.', h, w, 1);
134 %
135 % %anomaly_map_2d = hyperConvert3d(anomaly_map.', 30, 30, 1);
136 % figure;imagesc(r_rlx_2d);title(['LRX AD detector, K= ' num2str
      (K) ]); axis image; colorbar;
137 %
138 % treshold=100;
139
140
141
```

```
%%ACAD
figure;imagesc(reference_anomaly_map);axis image; colorbar;

false_anomalies = zeros(1,10);
true_anomalies = zeros(1,10);
correctly_predicted_anomalies = zeros(1,10);
counter_i=1;
for treshold =0.1:0.1:1
[d_acad, anomaly_map,threshold_check_values] = hyperACAD(matrix,
    treshold);
d_acad_2d = hyperConvert3d(d_acad.', h, w, 1);
anomaly_map_2d = hyperConvert3d(anomaly_map.', h, w, 1);
figure;imagesc(d_acad_2d); title(['ACAD result, treshold = '
    num2str(treshold) ]); axis image; colorbar;
figure;imagesc(anomaly_map_2d);title(['ACAD anomaly result,
    treshold = ' num2str(treshold) ]); axis image; colorbar;

for i=1:h
    for j =1: w
        if(anomaly_map_2d(i,j)==1 && reference_anomaly_map(i,j
            )==1)
            true_anomalies(counter_i) =true_anomalies(counter_i
                )+1;
        elseif anomaly_map_2d(i,j)==1
            false_anomalies(counter_i) =false_anomalies(
                counter_i)+1;
        end
    end
%       end
 correctly_predicted_anomalies(counter_i) = true_anomalies(
    counter_i)/nnz(reference_anomaly_map);
end
counter_i=counter_i+1;
end
```

Listing A.9: Hsueh mimicked image

```
%%
clear; clc;close all;
load('groundTruth_Cuprite_nEnd12.mat','-mat');
w = 200;
h = 200;

resultsDir= ['M:\Documents\Forked_MATLAB_hyperspectral_toolbox\
    HyperSpectralToolbox\figures\Hsueh',datestr(now, 'dd-mmm-
    yyyy')];

%resultsDir =regexprep(resultsDir,':d*','')
%resultsDir ='E:\One Drive\OneDrive for Business\NTNU\Master\
    Anomaly detection results\MATLAB\synthetic_images\lol';
```

```
11 [status, msg, msgID] = mkdir(resultsDir);
12
13 M_endmembers=M;
14 goodBands = [10:100 116:150 180:216]; % for AVIRIS with 224
       channels
15 M_endmembers=M(goodBands,:);
16 [n_bands,k] = size(M_endmembers);
17 image = zeros(h,w,n_bands);
18 reference_anomaly_map = zeros(h,w);
19 BACKGROUND=0.2*M_endmembers(:,1) +0.2*M_endmembers(:,3)+0.2*
       M_endmembers(:,5)+0.2*M_endmembers(:,7)+0.2*M_endmembers
       (:,12);
20 SNR = 20;
21 % Setting background
22 for i=1:h
23     for j=1:w
24         image(i,j,:) = BACKGROUND;
25     end
26 end
27
28 %column locations
29 KERNEL_SIZE_TWO_LOCATION =40; % column one
30 KERNEL_SIZE_TWO_2_LOCATION = 70; %column two, mixed pixels
31 KERNEL_SIZE_TWO_MIXED_LOCATION =100; % column three, mixed pixel
32 KERNEL_SIZE_ONE_BKG_MIXED_LOCATION =130; % column four, mixed
       pixel and background, 50/50
33 KERNEL_SIZE_ONE_BKG_75_MIX_LOCATION =160; %column five, mixed
       pixel and background, 25/75
34 M_anomaly_pure = M_endmembers(:,10);
35 M_A = M_endmembers(:,1);
36 M_B = M_endmembers(:,3);
37 M_K = M_endmembers(:,5);
38 M_M = M_endmembers(:,7);
39 M_C = M_endmembers(:,12);
40
41 for i=KERNEL_SIZE_TWO_LOCATION:30:
       KERNEL_SIZE_ONE_BKG_75_MIX_LOCATION
42     %first column
43     image(i,KERNEL_SIZE_TWO_LOCATION,:) = M_anomaly_pure;
44     reference_anomaly_map(i,KERNEL_SIZE_TWO_LOCATION)=1;
45
46     image(i,KERNEL_SIZE_TWO_LOCATION +1,:) = M_anomaly_pure;
47     reference_anomaly_map(i,KERNEL_SIZE_TWO_LOCATION +1)=1;
48
49     image(i+1,KERNEL_SIZE_TWO_LOCATION,:) = M_anomaly_pure;
50     reference_anomaly_map(i+1,KERNEL_SIZE_TWO_LOCATION)=1;
51
52     image(i+1,KERNEL_SIZE_TWO_LOCATION +1,:) = M_anomaly_pure;
53     reference_anomaly_map(i+1,KERNEL_SIZE_TWO_LOCATION +1)=1;
54
```

```
55      % second column
56       image(i,KERNEL_SIZE_TWO_2_LOCATION,:) = M_anomaly_pure;
57      reference_anomaly_map(i,KERNEL_SIZE_TWO_2_LOCATION)=1;
58
59      image(i,KERNEL_SIZE_TWO_2_LOCATION +1,:) = M_anomaly_pure;
60      reference_anomaly_map(i,KERNEL_SIZE_TWO_2_LOCATION +1)=1;
61
62      image(i+1,KERNEL_SIZE_TWO_2_LOCATION,:) = M_anomaly_pure;
63      reference_anomaly_map(i+1,KERNEL_SIZE_TWO_2_LOCATION)=1;
64
65      image(i+1,KERNEL_SIZE_TWO_2_LOCATION +1,:) = M_anomaly_pure;
66      reference_anomaly_map(i+1,KERNEL_SIZE_TWO_2_LOCATION +1)=1;
67
68      for(j=KERNEL_SIZE_TWO_MIXED_LOCATION:30:
            KERNEL_SIZE_ONE_BKG_75_MIX_LOCATION)
69           switch j
70                  case KERNEL_SIZE_TWO_MIXED_LOCATION
71                      switch i
72                          case 40
73                              image(i+1,j,:)= 0.5*M_A + 0.5 *M_B;
74                              image(i+1,j+1,:)=0.5*M_A + 0.5*M_C;
75                              image(i,j,:) = 0.5*M_A + 0.5*M_K;
76                              image(i,j+1,:)= 0.5*M_A + 0.5*M_M;
77                          case 70
78                              image(i+1,j,:)= 0.5*M_A + 0.5 *M_B;
79                              image(i+1,j+1,:)=0.5*M_A + 0.5*M_C;
80                              image(i,j,:) = 0.5*M_B + 0.5*M_K;
81                              image(i,j+1,:)= 0.5*M_B + 0.5*M_M;
82                          case 100
83                              image(i+1,j,:)= 0.5*M_A + 0.5 *M_C;
84                              image(i+1,j+1,:)=0.5*M_B + 0.5*M_C;
85                              image(i,j,:) = 0.5*M_C + 0.5*M_K;
86                              image(i,j+1,:)= 0.5*M_C + 0.5*M_M;
87                          case 130
88                              image(i+1,j,:)= 0.5*M_A + 0.5 *M_K;
89                              image(i+1,j+1,:)=0.5*M_B +0.5*M_K;
90                              image(i,j,:) = 0.5*M_C + 0.5*M_K;
91                              image(i,j+1,:)= 0.5*M_K + 0.5*M_M;
92                         case 160
93                              image(i+1,j,:)= 0.5*M_A + 0.5 *M_M;
94                              image(i+1,j+1,:)=0.5*M_B + 0.5*M_M;
95                              image(i,j,:) = 0.5*M_C + 0.5*M_M;
96                              image(i,j+1,:)= 0.5*M_K + 0.5*M_M;
97                      end
98                      reference_anomaly_map(i,j)=1;
99                      reference_anomaly_map(i+1,j)=1;
100                     reference_anomaly_map(i+1,j+1)=1;
101                     reference_anomaly_map(i,j+1)=1;
102                case  KERNEL_SIZE_ONE_BKG_MIXED_LOCATION
103                    switch i
```

```
104                         case 40
105                             image(i,j,:) = 0.5*M_A + 0.5*BACKGROUND;
106                              reference_anomaly_map(i,j)=1;
107                         case 70
108                             image(i,j,:) = 0.5*M_B + 0.5*BACKGROUND;
109                              reference_anomaly_map(i,j)=1;
110                         case 100
111                             image(i,j,:) = 0.5*M_C + 0.5*BACKGROUND;
112                              reference_anomaly_map(i,j)=1;
113                         case 130
114                             image(i,j,:) = 0.5*M_K + 0.5*BACKGROUND;
115                              reference_anomaly_map(i,j)=1;
116                         case 160
117                             image(i,j,:) = 0.5*M_M + 0.5*BACKGROUND;
118                              reference_anomaly_map(i,j)=1;
119                     end
120                 case  KERNEL_SIZE_ONE_BKG_75_MIX_LOCATION
121                 switch i
122                     case 40
123                         image(i,j,:) = 0.25*M_A + 0.75*BACKGROUND;
124                          reference_anomaly_map(i,j)=1;
125                     case 70
126                         image(i,j,:) = 0.25*M_B + 0.75*BACKGROUND;
127                          reference_anomaly_map(i,j)=1;
128                     case 100
129                         image(i,j,:) = 0.25*M_C + 0.75*BACKGROUND;
130                          reference_anomaly_map(i,j)=1;
131                     case 130
132                         image(i,j,:) = 0.25*M_K + 0.75*BACKGROUND;
133                          reference_anomaly_map(i,j)=1;
134                     case 160
135                         image(i,j,:) = 0.25*M_M + 0.75*BACKGROUND;
136                          reference_anomaly_map(i,j)=1;
137                 end
138             end
139         end
140     end
141     for(i=1:n_bands)
142         image(:,:,i) = awgn(image(:,:,i),SNR);
143     end
144     figure;imagesc(image(:,:,160)); title('Band 160 of Hsueh-
        mimicked image, with gaussian noise'); axis image;
145
146     %% RX testing
147     matrix = hyperConvert2d(image);
148     r_rx=hyperRxDetector(matrix);
149     r_rx = hyperConvert3d(r_rx.', h, w, 1);
150     figure; imagesc(reference_anomaly_map); title(['Expected anomaly
            map'] ); axis image; colorbar;
151     hyperSaveFigure(gcf, sprintf(['%s\\hsueh_expected_anomaly_map' '
```

```
    .png' ], resultsDir));%
figure; imagesc(r_rx); title(['RX AD '] ); axis image; colorbar;
hyperSaveFigure(gcf, sprintf(['%s\\RX AD' '.png' ], resultsDir))
    ;%
max_value_rx= max(r_rx);
max_value_rx = max(max_value_rx);
% set 90% of max_value as an anomaly
treshold_rx = max_value_rx*0.90;
anomaly_map_rx=zeros(h,w);
correctly_predicted_anomalies_rx=0;
for i=1:h
    for j=1:w
        if r_rx(i,j) >=treshold_rx
            anomaly_map_rx(i,j)=1;
            if reference_anomaly_map(i,j)==1
                correctly_predicted_anomalies_rx =
                    correctly_predicted_anomalies_rx+1;
            end
        end
    end
end
figure; imagesc(anomaly_map_rx); title(['RX anomaly map'] );
    axis image; colorbar;
hyperSaveFigure(gcf, sprintf(['%s\\RX anomaly_map' '.png' ],
    resultsDir));%
%check difference RX-anomaly_map and reference anomaly map
difference_from_reference_rx = zeros(h,w);
false_anomalies_hsueh_rx=nnz(anomaly_map_rx) - nnz(
    reference_anomaly_map);
if(false_anomalies_hsueh_rx<0)
    false_anomalies_hsueh_rx=0;
end

for i=1:h
    for j=1:w
        difference_from_reference_rx(i,j)= (
            reference_anomaly_map(i,j)- anomaly_map_rx(i,j));
    end
end
figure; imagesc(difference_from_reference_rx); title(['false or
    undetected anomalies RX'] ); axis image; colorbar;
hyperSaveFigure(gcf, sprintf(['%s\\Undetected anomalies RX' '.
    png' ], resultsDir));%

nnz_rx = nnz(difference_from_reference_rx);
percent_predicted_anomalies_hsueh_rx = (
    correctly_predicted_anomalies_rx)/nnz((reference_anomaly_map)
    );


```

```
191 %% LRX without anomaly removal
192 difference_from_reference_lrx = zeros(h,w);
193 counter_i=1;
194 anomaly_map_alrx=zeros(1,h*w);
195 for K=5:5:30 % 35 bugged
196  difference_from_reference_lrx = zeros(h,w);
197  r_lrx=hyperLRxDetectorCorr(matrix,K);
198  r_lrx = hyperConvert3d(r_lrx.', h, w, 1);
199  figure; imagesc(r_lrx); title(['LRX K=' num2str(K)] ); axis
       image; colorbar;
200  hyperSaveFigure(gcf, sprintf(['%s\\LRX K=' num2str(K) '.png' ],
        resultsDir));%
201  max_value_lrx= max(r_lrx);
202  max_value_lrx = max(max_value_lrx);
203 % set 75% of max_value as an anomaly
204 treshold_lrx = max_value_lrx*0.75;
205 anomaly_map_lrx=zeros(h,w);
206 correctly_predicted_anomalies_lrx =0;
207 for i=1:h
208     for j=1:w
209         if r_lrx(i,j) >=treshold_lrx
210             anomaly_map_lrx(i,j)=1;
211             if reference_anomaly_map(i,j)==1
212                 correctly_predicted_anomalies_lrx =
                        correctly_predicted_anomalies_lrx+1;
213             end
214         end
215     end
216 end
217
218  for i=1:h
219     for j=1:w
220         difference_from_reference_lrx(i,j)= (
                reference_anomaly_map(i,j)- anomaly_map_lrx(i,j));
221     end
222  end
223  figure; imagesc(difference_from_reference_lrx); title(['False
       or undetected anomalies LRX, K=' num2str(K)] ); axis image;
       colorbar;
224  hyperSaveFigure(gcf, sprintf(['%s\\false anomalies LRX K='
       num2str(K) '.png' ], resultsDir));%
225 false_anomalies_hsueh_lrx(counter_i)=nnz(anomaly_map_lrx) - nnz(
    reference_anomaly_map);
226 if(false_anomalies_hsueh_lrx(counter_i)<0)
227     false_anomalies_hsueh_lrx(counter_i)=0;
228 end
229 percent_predicted_anomalies_hsueh_lrx(counter_i) =
    correctly_predicted_anomalies_lrx/nnz(reference_anomaly_map);
230 %percent_predicted_anomalies_hsueh_lrx(counter_i) = (nnz(
    anomaly_map_lrx) - false_anomalies_hsueh_lrx)/nnz(
```

```
    reference_anomaly_map);
231  nnz_lrx(counter_i) = nnz(difference_from_reference_lrx);
232  counter_i= counter_i +1;
233 end
234 %%
235 %LRX with anomaly removal
236 difference_from_reference_lrx_ad_remov = zeros(h,w);
237 counter_i=1;
238 correctly_predicted_anomalies_alrx=0;
239 treshold =250;
240 for K=5:5:35
241  difference_from_reference_lrx_ad_remov = zeros(h,w);
242  [r_lrx_ad_remov,anomaly_map_alrx,location_of_anomalies,lsllsl]=
        hyperLRX_anomaly_set_remover(matrix,K,treshold);
243  r_lrx_ad_remov = hyperConvert3d(r_lrx_ad_remov.', h, w, 1);
244  figure; imagesc(r_lrx_ad_remov); title(['ALRX AD K=' num2str(K)
        ]); axis image; colorbar;
245  hyperSaveFigure(gcf, sprintf(['%s\\ALRX AD K=' num2str(K) '.png
        '], resultsDir));%
246  anomaly_map_alrx_2d = hyperConvert3d(anomaly_map_alrx.',h,w,1);
247       for i=1:h
248           for j=1:w
249               if anomaly_map_alrx_2d(i,j) ==1 &&
                      reference_anomaly_map(i,j) ==1
250                   correctly_predicted_anomalies_alrx =
                          correctly_predicted_anomalies_alrx +1;
251               end
252           end
253       end
254  false_anomalies_hsueh_alrx(counter_i)=nnz(anomaly_map_alrx) -
        correctly_predicted_anomalies_alrx;
255 % if(false_anomalies_hsueh_alrx(counter_i)<0)
256 %     false_anomalies_hsueh_alrx=0;
257 % end
258 %percent_predicted_anomalies_hsueh_alrx(counter_i) = (nnz(
        anomaly_map_alrx) - false_anomalies_hsueh_alrx)/nnz(
        reference_anomaly_map);
259 percent_predicted_anomalies_hsueh_alrx(counter_i) =
        correctly_predicted_anomalies_alrx/nnz(reference_anomaly_map)
        ;
260
261 figure; imagesc(difference_from_reference_rx); title(['False or
        undetected anomalies ALRX AD, K=' num2str(K)]); axis image;
        colorbar;
262  hyperSaveFigure(gcf, sprintf(['%s\\false anomalies ALRX AD K='
        num2str(K) '.png'], resultsDir));%
263  nnz_lrx(counter_i) = nnz(difference_from_reference_lrx_ad_remov
        );
264  counter_i= counter_i +1;
265 end
```

```
266
267
268 %% ACAD
269 matrix = hyperConvert2d(image);
270 treshold= 0.9;
271 [r_acad,anomaly_map,not_used]=hyperACAD(matrix,treshold);
272 r_acad = hyperConvert3d(r_acad.', h, w, 1);
273 figure; imagesc(reference_anomaly_map); title(['Expected anomaly
      map']); axis image; colorbar;
274 figure; imagesc(r_acad); title(['ACAD ']); axis image; colorbar
      ;
275 figure; imagesc(anomaly_map); title(['Anomaly map ']); axis
     image; colorbar;
276
277 %% format data
278 r_acad_formatted = zeros(w,h);
279 for i =1: w
280     for j=1:h
281         if r_acad(i,j) <0
282             r_acad_formatted(i,j)=0;
283         elseif r_acad(i,j) >0 && ~isinf(r_acad(i,j))
284             r_acad_formatted(i,j) = r_acad(i,j);
285         else
286             r_acad_formatted(i,j)=0;
287         end
288     end
289 end
```

Appendix B

VHDL Code description

Most of the entities are written in the code-writing technique called the two-process method, introduced by Jiri Gaisler. This technique is described on the following web-page:
"https://www.gaisler.com/doc/vhdl2proc.pdf".

The two-process method divides the code into two processes; one asynchronous process and one synchronous process. The algorithm to be executed by the entity is located within the asynchronous block. Results of the asynchronous block get registered into the synchronous process. The asynchronous process uses variables to a wide extent. Record types are also widely used. A two-process entity can be seen in Figure B.1.

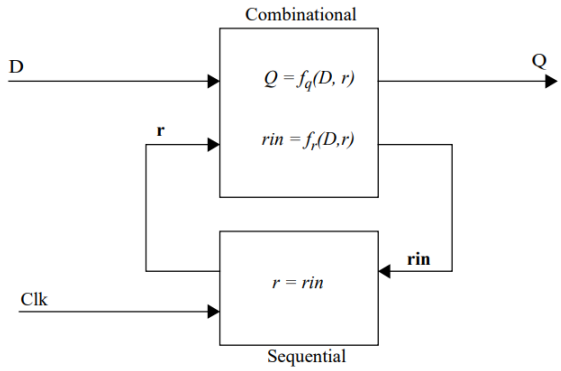

Figure 20: Generic two-process circuit

Figure B.1: Two process method.

Appendix C

VHDL code

C.1 ACAD correlation

Listing C.1: ACAD correlation

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.Common_types_and_functions.all;
7
8  -- Correlation module with AXI lite stream interface
9  entity acad_correlation is
10   port(din                    : in     std_logic_vector(P_BANDS*
         PIXEL_DATA_WIDTH-1 downto 0);  --
11                                          --Horizontal
12                                          --input vector
13         valid                 : in     std_logic;
14         clk                   : in     std_logic;
15         clk_en                : in     std_logic;
16         reset_n               : in     std_logic;
17         dout                  : out std_logic_vector(P_BANDS*
               PIXEL_DATA_WIDTH*2*2 -1 downto
18                                       0);  -- writing two 32-bit
                                             elements per cycle
19         valid_out             : out    std_logic;
20         writes_done_on_column : out    std_logic_vector(log2(
               P_BANDS/2) downto 0)
21         );
22 end acad_correlation;
23
24 architecture Behavioral of acad_correlation is
25 -- using 18kbit BRAM, one for odd indexes, one for even per row
```

```
    of the
-- correlation matrix. This results in P_BANDS 36kbit BRAMs
    actually being synthesized.
  constant NUMBER_OF_WRITES_PER_CYCLE : integer range 0 to 2
              := 2;  --
                                                                --
                                                                --
                                                                --
                                                                --
                                                                --
  constant NUMBER_OF_WRITES_PER_COLUMN   : integer range 0 to
      P_BANDS/2 := P_BANDS/2;

  signal r_write_address       : integer range 0 to B_RAM_SIZE-1
      := 0;
  signal write_done_on_column : integer range 0 to P_BANDS/2
      := 0;
  signal flag_has_read_first   : std_logic :=
    '0';  --first element in the read-write pipeline
  signal flag_has_read_second : std_logic :=
    '0';  --second element in the read-write pipeline
  signal write_enable      : std_logic := '0';
  signal read_enable       : std_logic := '1';
  signal read_address      : integer range 0 to B_RAM_SIZE-1;
  signal write_address     : integer range 0 to B_RAM_SIZE-1;
  signal flag_first_pixel : std_logic := '1';
-- indicates that the current pixel working on is the first
    pixel
  signal r_dout_prev : std_logic_vector(P_BANDS*PIXEL_DATA_WIDTH
      *2*2-1 downto
                                        0);  -- Previous value
                                             -- outputted from
                                                the BRAMs.
  signal r_read_address                  : integer range 0 to
      B_RAM_SIZE-1                  := 0;
  constant EVEN_ROW_TOP_INDEX_INPUT       : integer range 0 to
      P_BANDS*PIXEL_DATA_WIDTH-1   := P_BANDS*PIXEL_DATA_WIDTH-1;
  constant EVEN_ROW_TOP_INDEX_CORRELATION : integer range 0 to
      P_BANDS*PIXEL_DATA_WIDTH*2-1 := P_BANDS*PIXEL_DATA_WIDTH
      *2-1;
  signal dout_BRAMS                       : std_logic_vector(
      P_BANDS*PIXEL_DATA_WIDTH*2*2-1 downto 0):= (others=>'0');
```

```
56 begin
57
58
59   GEN_BRAM_18_updates : for i in 0 to P_BANDS-1 generate
60     -- Generating N_BRAMS = P_BANDS BRAM 36 kbits.
61     signal data_in_even_i, data_in_odd_i, data_out_even_i,
           data_out_odd_i : std_logic_vector(B_RAM_BIT_WIDTH -1
           downto 0);
62 --value read from BRAM (odd index) before writing to address
63   begin
64     -- Block ram row for even addresses and row indexes of the
           correlation matrix
65     block_ram_even : entity work.block_ram
66       generic map (
67         B_RAM_SIZE      => B_RAM_SIZE,
68         B_RAM_BIT_WIDTH => B_RAM_BIT_WIDTH)
69       port map (
70         clk           => clk,
71         aresetn       => reset_n,
72         data_in       => data_in_even_i,
73         write_enable  => write_enable,
74         read_enable   => read_enable,
75         read_address  => read_address,
76         write_address => write_address,
77         data_out      => data_out_even_i);
78     -- Block ram row for odd addresses and row indexes of the
           correlation matrix
79     block_ram_odd : entity work.block_ram
80       generic map (
81         B_RAM_SIZE      => B_RAM_SIZE,
82         B_RAM_BIT_WIDTH => B_RAM_BIT_WIDTH)
83       port map (
84         clk           => clk,
85         aresetn       => reset_n,
86         data_in       => data_in_odd_i,
87         write_enable  => write_enable,
88         read_enable   => read_enable,
89         read_address  => read_address,
90         write_address => write_address,
91         data_out      => data_out_odd_i);
92
93 -- generate P_BAND write PROCESSES writes on clock cycle after
94     process(clk, clk_en, din, valid, reset_n, r_write_address,
           read_address, write_address, data_out_odd_i,
           data_out_even_i, write_done_on_column, flag_first_pixel,
           write_enable)
95       -- a_factor_0x is the even indexed row factor
96       -- b_factor_0x is the odd indexed row factor
97       variable a_factor_01_i                  : std_logic_vector(
             PIXEL_DATA_WIDTH-1 downto 0);
```

```
98        variable a_factor_02_i                    : std_logic_vector(
              PIXEL_DATA_WIDTH-1 downto 0);
99        variable b_factor_01_i                    : std_logic_vector(
              PIXEL_DATA_WIDTH-1 downto 0);
100       variable b_factor_02_i                    : std_logic_vector(
              PIXEL_DATA_WIDTH-1 downto 0);
101       variable v_input_even_i, v_input_odd_i : std_logic_vector(
              B_RAM_BIT_WIDTH-1 downto 0); --
102       variable v_data_out_prev_even_i           : std_logic_vector(
              PIXEL_DATA_WIDTH*2-1 downto 0);
103       variable v_data_out_prev_odd_i            : std_logic_vector(
              PIXEL_DATA_WIDTH*2-1 downto 0);
104     begin
105
106       if rising_edge(clk) and clk_en = '1' then
107         if reset_n = '0' or valid = '0' then
108           a_factor_01_i := (others => '0');
109           a_factor_02_i := (others => '0');
110           b_factor_01_i := (others => '0');
111           b_factor_02_i := (others => '0');
112
113         elsif valid = '1' and write_done_on_column <=
              NUMBER_OF_WRITES_PER_COLUMN-1 and write_enable = '1'
              then --and to_integer(unsigned(write_done_on_column)
              ) > 0 then
114           if flag_first_pixel = '0' then
115             --input din is horizontal vector. A/B_factor 01 is
                    the transposed
116             --vertical element factor of the product din.' * din
                    . A/B_factor_02 is
117             --the horizontal element.
118             a_factor_01_i := din(PIXEL_DATA_WIDTH -1 +
                    PIXEL_DATA_WIDTH*write_done_on_column*
                    NUMBER_OF_WRITES_PER_CYCLE downto
                    PIXEL_DATA_WIDTH*write_done_on_column*
                    NUMBER_OF_WRITES_PER_CYCLE);
119             b_factor_01_i := din(PIXEL_DATA_WIDTH*2-1 +
                    PIXEL_DATA_WIDTH*write_done_on_column*
                    NUMBER_OF_WRITES_PER_CYCLE downto
                    PIXEL_DATA_WIDTH + PIXEL_DATA_WIDTH*
                    write_done_on_column*NUMBER_OF_WRITES_PER_CYCLE);
120             -- "Horizontal" element
121             a_factor_02_i := din(P_BANDS*PIXEL_DATA_WIDTH -(
                    P_BANDS-i)*PIXEL_DATA_WIDTH + PIXEL_DATA_WIDTH-1
                    downto P_BANDS*PIXEL_DATA_WIDTH-((P_BANDS - i)*
                    PIXEL_DATA_WIDTH));
122             b_factor_02_i := a_factor_02_i;
123
124             v_input_even_i := std_logic_vector(to_signed(
                    to_integer(signed(a_factor_01_i))*to_integer(
```

```
                signed(a_factor_02_i)), v_input_even_i'length));
125             v_input_odd_i  := std_logic_vector(to_signed(
                    to_integer(signed(b_factor_01_i))*to_integer(
                    signed(b_factor_02_i)), v_input_odd_i'length));
126
127             v_data_out_prev_even_i := r_dout_prev(P_BANDS*
                    PIXEL_DATA_WIDTH*2-(P_BANDS-i)*PIXEL_DATA_WIDTH*2
                     + PIXEL_DATA_WIDTH*2-1 downto P_BANDS*
                    PIXEL_DATA_WIDTH*2-((P_BANDS-i)*PIXEL_DATA_WIDTH
                    *2));
128             v_data_out_prev_odd_i  := r_dout_prev(P_BANDS*
                    PIXEL_DATA_WIDTH*NUMBER_OF_WRITES_PER_CYCLE*2-(
                    P_BANDS-i)*PIXEL_DATA_WIDTH*2+PIXEL_DATA_WIDTH
                    *2-1 downto P_BANDS*PIXEL_DATA_WIDTH*
                    NUMBER_OF_WRITES_PER_CYCLE*2-(P_BANDS-i)*
                    PIXEL_DATA_WIDTH*2);
129
130             data_in_even_i <= std_logic_vector(to_signed(
                    to_integer(signed(v_input_even_i))+ to_integer(
                    signed(v_data_out_prev_even_i)), data_in_even_i'
                    length));
131             data_in_odd_i  <= std_logic_vector(to_signed(
                    to_integer(signed(v_input_odd_i)) + to_integer(
                    signed(v_data_out_prev_odd_i)), data_in_odd_i'
                    length));
132
133           elsif flag_first_pixel = '1' then
134             -- special case for the first pixel written, where
135             -- the data contained in the BRAM is not
136             -- initialized to something known.
137             --input din is horizontal vector. A/B_factor 01 is
                    the transposed
138             --vertical element factor of the product din.' * din
                    . A/B_factor_02 is
139             --the horizontal element.
140             a_factor_01_i := din(PIXEL_DATA_WIDTH -1 +
                    PIXEL_DATA_WIDTH*write_done_on_column*
                    NUMBER_OF_WRITES_PER_CYCLE downto
                    PIXEL_DATA_WIDTH*write_done_on_column*
                    NUMBER_OF_WRITES_PER_CYCLE);
141             b_factor_01_i := din(PIXEL_DATA_WIDTH*2-1 +
                    PIXEL_DATA_WIDTH*write_done_on_column *
                    NUMBER_OF_WRITES_PER_CYCLE downto
                    PIXEL_DATA_WIDTH + PIXEL_DATA_WIDTH*
                    write_done_on_column*NUMBER_OF_WRITES_PER_CYCLE);
142             -- "Horizontal" element
143             a_factor_02_i := din(P_BANDS*PIXEL_DATA_WIDTH -(
                    P_BANDS-i)*PIXEL_DATA_WIDTH + PIXEL_DATA_WIDTH-1
                    downto P_BANDS*PIXEL_DATA_WIDTH-((P_BANDS - i)*
                    PIXEL_DATA_WIDTH));
```

```
144                 b_factor_02_i := a_factor_02_i;
145
146                 v_input_even_i := std_logic_vector(to_signed(
                        to_integer(signed(a_factor_01_i))*to_integer(
                        signed(a_factor_02_i)), v_input_even_i'length));
147                 v_input_odd_i  := std_logic_vector(to_signed(
                        to_integer(signed(b_factor_01_i))*to_integer(
                        signed(b_factor_02_i)), v_input_odd_i'length));
148                 data_in_even_i <= v_input_even_i;
149                 data_in_odd_i  <= v_input_odd_i;
150               end if;
151
152           end if;
153         end if;
154
155       end process;
156       -- Even row of output
157       dout_BRAMS(P_BANDS*PIXEL_DATA_WIDTH*2-(P_BANDS-i)*
              PIXEL_DATA_WIDTH*2 +PIXEL_DATA_WIDTH*2-1 downto P_BANDS*
              PIXEL_DATA_WIDTH*2 -(P_BANDS-i)*PIXEL_DATA_WIDTH*2)

              <= data_out_even_i;
158       dout(P_BANDS*PIXEL_DATA_WIDTH*2-(P_BANDS-i)*PIXEL_DATA_WIDTH
              *2 +PIXEL_DATA_WIDTH*2-1 downto P_BANDS*PIXEL_DATA_WIDTH
              *2 -(P_BANDS-i)*PIXEL_DATA_WIDTH*2)

              <= data_in_even_i;
159       -- Odd row of output
160       dout_BRAMS(P_BANDS*PIXEL_DATA_WIDTH*2-(P_BANDS-i)*
              PIXEL_DATA_WIDTH*2+ PIXEL_DATA_WIDTH*2-1+
              EVEN_ROW_TOP_INDEX_CORRELATION+1 downto P_BANDS*
              PIXEL_DATA_WIDTH *2 - (P_BANDS-i)*PIXEL_DATA_WIDTH*2 +
              EVEN_ROW_TOP_INDEX_CORRELATION+1) <= data_out_odd_i;
161       dout(P_BANDS*PIXEL_DATA_WIDTH*2-(P_BANDS-i)*PIXEL_DATA_WIDTH
              *2+ PIXEL_DATA_WIDTH*2-1+EVEN_ROW_TOP_INDEX_CORRELATION+1
               downto P_BANDS*PIXEL_DATA_WIDTH *2 - (P_BANDS-i)*
              PIXEL_DATA_WIDTH*2 +EVEN_ROW_TOP_INDEX_CORRELATION+1)
                    <= data_in_odd_i;
162
163     end generate;
164
165     -- Register in old values of dout
166     process (clk, clk_en, dout)
167     begin
168       if rising_edge(clk) and clk_en = '1' then
169         --r_dout_prev <= dout;
170         r_dout_prev <= dout_BRAMS;
171       end if;
172     end process;
173
```

```
174
175
176 -- process to drive address and control
177   process(clk, clk_en, r_write_address, write_done_on_column,
          reset_n, valid, flag_has_read_second, flag_has_read_first)
178
179   begin
180     if rising_edge(clk) and clk_en = '1' then
181       if reset_n = '0' then              --or valid = '0' then
182         r_write_address      <= 0;
183         read_address         <= 0;
184         write_enable         <= '0';
185         read_enable          <= '1';
186         write_done_on_column <= 0;
187         flag_has_read_first  <= '0';
188         flag_first_pixel     <= '1';
189         valid_out <= '0';
190       elsif valid = '0' then
191         write_enable         <= '0';
192         flag_has_read_first  <= '0';
193         flag_has_read_second <= '0';
194         valid_out <= '0';
195       elsif valid = '1' and write_done_on_column <=
            NUMBER_OF_WRITES_PER_COLUMN-1 and flag_first_pixel =
            '1' then
196         if flag_has_read_first = '0' then
197           -- Need to read first element of the pixel before
                starting any writes
198           flag_has_read_first <= '1';
199           read_address        <= r_write_address;
200           write_address       <= r_write_address;
201           read_enable         <= '1';
202           write_enable        <= '1';
203           valid_out <= '0';
204         elsif flag_has_read_first = '1' and write_enable = '1'
              then
205           r_write_address      <= r_write_address +1;
206           write_address        <= r_write_address;
207           read_address         <= r_write_address+1;
208           write_enable         <= '1';
209           write_done_on_column <= write_done_on_column + 1;
210           valid_out <= '1';
211         end if;
212       -- Going to buffer two read elements.
213       elsif valid = '1' and write_done_on_column <=
            NUMBER_OF_WRITES_PER_COLUMN-1 and flag_first_pixel =
            '0' then
214         if flag_has_read_first = '0' and flag_has_read_second =
              '0' then
215           -- Need to read first element of the pixel before
```

```
                starting any writes
216             flag_has_read_first <= '1';
217             read_address        <= r_write_address;
218             write_address       <= r_write_address;
219             read_enable         <= '1';
220             write_enable        <= '0';
221             valid_out <= '0';
222           elsif flag_has_read_first = '1' and write_enable = '0'
                  and flag_has_read_second = '0' then
223             read_address          <= r_write_address +1;
224             read_enable           <= '1';
225             flag_has_read_second <= '1';
226             r_read_address        <= r_read_address +1;
227             valid_out <= '0';
228           end if;
229           if flag_has_read_second = '1' and write_enable = '0'
                  then
230             write_address  <= r_write_address;
231             read_address   <= r_write_address+2;
232             write_enable   <= '1';
233             r_read_address <= r_read_address +1;
234             valid_out <= '0';
235           elsif flag_has_read_second = '1' and write_enable = '1'
                  then
236             r_write_address       <= r_write_address +1;
237             write_address         <= r_write_address;
238             read_address          <= r_read_address;
239             r_read_address        <= r_read_address +1;
240             write_done_on_column <= write_done_on_column + 1;
241             valid_out <= '1';
242           end if;
243         elsif valid = '1' and write_done_on_column >
              NUMBER_OF_WRITES_PER_COLUMN-1 then
244           -- New pixel coming on data_in input
245           -- Assuming consequent pixels are hold valid, starting
                  working on
246           -- next pixel next cycle;
247           valid_out <= '1';
248           r_write_address       <= 0;
249           r_read_address        <= 0;
250           read_address          <= 0;
251           write_enable          <= '0';
252           write_done_on_column <= 0;
253           flag_has_read_first  <= '0';
254           flag_has_read_second <= '0';
255           -- Now one pixel has been finished processed, the
                  contents of the
256           -- BRAM is at least known
257           flag_first_pixel      <= '0';
258         end if;
```

```
259
260       end if;
261     end process;
262
263
264     writes_done_on_column <= std_logic_vector(to_unsigned(
            write_done_on_column, writes_done_on_column'length));
265
266   end Behavioral;
```

C.2 Elimination core

The entity referred to as **Elimination core** in the thesis, is named **backward elim core** in the VHDL code.

Listing C.2: Elimination core

```
1
2   library IEEE;
3   use IEEE.std_logic_1164.all;
4   use ieee.numeric_std.all;
5   --use IEEE.fixed_pkg.all;
6
7   library work;
8   use work.Common_types_and_functions.all;
9
10  -- This core is utilized by both backward and forward
        elimination
11  entity backward_elim_core is
12    port(clk                  : in  std_logic;
13         reset_n              : in  std_logic;
14         clk_en               : in  std_logic;
15         input_backward_elim  : in  input_elimination_reg_type;
16         output_backward_elim : out
               output_backward_elimination_reg_type);
17  end backward_elim_core;
18
19  architecture Behavioral of backward_elim_core is
20
21    signal r, r_in             : input_elimination_reg_type;
22    constant ONE               : signed(PIXEL_DATA_WIDTH*2-1
          downto 0) := (0 => '1', others => '0');
23    constant PRECISION_SHIFT   : integer range 0 to 3
                          := 3;  -- Used to specify numbers of
24                                   -- shift of r_j_i
25    signal divisor_is_negative : std_logic;
26    -- If the divisor is negative, we need to take two's
          complement of the divisor
27    signal divisor             : std_logic_vector(PIXEL_DATA_WIDTH
          *2 -1 downto 0);
```

```
28    signal divisor_valid        : std_logic
                                    := '0';
29    signal remainder_valid      : std_logic
                                    := '0';
30    type remainders_array is array(0 to PIXEL_DATA_WIDTH*2-2) of
          std_logic_vector(PIXEL_DATA_WIDTH*2-1 downto 0);
31    signal remainders           : remainders_array;
32    signal msb_index            : integer range 0 to 31;  -- msb of
          the divisor(unsigned)
33    signal msb_valid            : std_logic
                                    := '0';
34    -- to be used in two's complement.
35    signal divisor_lut          : unsigned(DIV_PRECISION-1 downto
          0);
36    signal divisor_inv          : unsigned(DIV_PRECISION-1 downto
          0);
37  begin
38
39    division_lut_2 : entity work.division_lut
40      port map (
41        y     => divisor_lut,
42        y_inv => divisor_inv);
43
44    input_to_divisor_lut : process(msb_valid, msb_index)
45    begin
46      if msb_valid = '1' and msb_index<=DIV_PRECISION then
47        divisor_lut <= to_unsigned(to_integer(unsigned(divisor)),
              DIV_PRECISION);
48      else
49        divisor_lut <= to_unsigned(0, DIV_PRECISION);
50      end if;
51    end process;
52
53
54    check_if_divisor_is_negative : process(input_backward_elim.
          state_reg.state, input_backward_elim.row_i,
          input_backward_elim.valid_data, reset_n)
55    begin
56      if reset_n = '0' or not(input_backward_elim.state_reg.state
            = STATE_LAST_DIVISION) then
57        divisor_valid       <= '0';
58        divisor_is_negative <= '0';
59        divisor             <= std_logic_vector(to_signed(1,
              PIXEL_DATA_WIDTH*2));
60      elsif(input_backward_elim.row_i(input_backward_elim.index_i)
            (PIXEL_DATA_WIDTH*2-1) = '1' and input_backward_elim.
            valid_data = '1') then
61        -- row[i][i] is negative
62        -- using the absolute value
63        divisor_is_negative <= '1';
```

```
64        divisor             <= std_logic_vector(abs(signed(
              input_backward_elim.row_i(input_backward_elim.index_i))
              ));
65        divisor_valid       <= '1';
66      elsif input_backward_elim.valid_data = '1' then
67        divisor_is_negative <= '0';
68        divisor             <= std_logic_vector(
              input_backward_elim.row_i(input_backward_elim.index_i))
              ;
69        divisor_valid       <= '1';
70      else
71        divisor_valid       <= '0';
72        divisor_is_negative <= '0';
73        divisor             <= std_logic_vector(to_signed(1,
              PIXEL_DATA_WIDTH*2));
74      end if;
75    end process;
76
77
78  -- generate PIXEL_DATA_WIDTH*2-1 number of shifters that shifts
79  -- A[i][i] n places in order to see how many shifts yield the
      best
80  -- approximation to the division. Don't need to shift the
81  -- 31 bit as this is the sign bit.
82    generate_shifters : for i in 1 to PIXEL_DATA_WIDTH*2-1
        generate
83      signal remainder_after_approximation_i :
          remainder_after_approximation_record;
84    begin
85      process(divisor, divisor_valid, reset_n, input_backward_elim
          .state_reg)
86      begin
87        if reset_n = '0' or not(input_backward_elim.state_reg.
            state = STATE_LAST_DIVISION) then
88          remainder_after_approximation_i.remainder        <=
              std_logic_vector(shift_right(signed(divisor), i));
89          remainder_after_approximation_i.number_of_shifts <= i;
90          remainder_after_approximation_i.remainder_valid  <= '0';
91        elsif divisor_valid = '1' then
92          remainder_after_approximation_i.remainder        <=
              std_logic_vector(shift_right(signed(divisor), i));
93          remainder_after_approximation_i.number_of_shifts <= i;
94          remainder_after_approximation_i.remainder_valid  <= '1';
95        else
96          remainder_after_approximation_i.remainder        <=
              std_logic_vector(shift_right(signed(divisor), i));
97          remainder_after_approximation_i.number_of_shifts <= i;
98          remainder_after_approximation_i.remainder_valid  <= '0';
99        end if;
100     end process;
```

```
101       remainders(i-1) <= remainder_after_approximation_i.remainder
              ;
102       remainder_valid <= remainder_after_approximation_i.
              remainder_valid;
103     end generate;
104
105
106     find_msb : process(divisor_valid, input_backward_elim, reset_n
            , divisor)
107     begin
108       if divisor_valid = '1' and reset_n = '1' then
109         --For PIXEL_DATA_WIDTH = 16.
110         if divisor(30) = '1' then
111           msb_index <= 30;
112           msb_valid <= '1';
113         elsif divisor(29) = '1' then
114           msb_index <= 29;
115           msb_valid <= '1';
116         elsif divisor(28) = '1' then
117           msb_index <= 28;
118           msb_valid <= '1';
119         elsif divisor(27) = '1' then
120           msb_index <= 27;
121           msb_valid <= '1';
122         elsif divisor(26) = '1'then
123           msb_index <= 26;
124           msb_valid <= '1';
125         elsif divisor(25) = '1' then
126           msb_index <= 25;
127           msb_valid <= '1';
128         elsif divisor(24) = '1' then
129           msb_index <= 24;
130           msb_valid <= '1';
131         elsif divisor(23) = '1' then
132           msb_index <= 23;
133           msb_valid <= '1';
134         elsif divisor(22) = '1' then
135           msb_index <= 22;
136           msb_valid <= '1';
137         elsif divisor(21) = '1' then
138           msb_index <= 21;
139           msb_valid <= '1';
140         elsif divisor(20) = '1' then
141           msb_index <= 20;
142           msb_valid <= '1';
143         elsif divisor(19) = '1' then
144           msb_index <= 19;
145           msb_valid <= '1';
146         elsif divisor(18) = '1' then
147           msb_index <= 18;
```

```
148           msb_valid <= '1';
149         elsif divisor(17) = '1'then
150           msb_index <= 17;
151           msb_valid <= '1';
152         elsif divisor(16) = '1' then
153           msb_index <= 16;
154           msb_valid <= '1';
155         elsif divisor(15) = '1' then
156           msb_index <= 15;
157           msb_valid <= '1';
158         elsif divisor(14) = '1' then
159           msb_index <= 14;
160           msb_valid <= '1';
161         elsif divisor(13) = '1' then
162           msb_index <= 13;
163           msb_valid <= '1';
164         elsif divisor(12) = '1' then
165           msb_index <= 12;
166           msb_valid <= '1';
167         elsif divisor(11) = '1' then
168           msb_index <= 11;
169           msb_valid <= '1';
170         elsif divisor(10) = '1'then
171           msb_index <= 10;
172           msb_valid <= '1';
173         elsif divisor(9) = '1' then
174           msb_index <= 9;
175           msb_valid <= '1';
176         elsif divisor(8) = '1' then
177           msb_index <= 8;
178           msb_valid <= '1';
179         elsif divisor(7) = '1' then
180           msb_index <= 7;
181           msb_valid <= '1';
182         elsif divisor(6) = '1' then
183           msb_index <= 6;
184           msb_valid <= '1';
185         elsif divisor(5) = '1' then
186           msb_index <= 5;
187           msb_valid <= '1';
188         elsif divisor(4) = '1' then
189           msb_index <= 4;
190           msb_valid <= '1';
191         elsif divisor(3) = '1' then
192           msb_index <= 3;
193           msb_valid <= '1';
194         elsif divisor(2) = '1' then
195           msb_index <= 2;
196           msb_valid <= '1';
197         elsif divisor(1) = '1' then
```

```
198           msb_index <= 1;
199           msb_valid <= '1';
200         elsif divisor(0) = '1' then
201           msb_index <= 0;
202           msb_valid <= '1';
203         else
204           msb_index <= 0;
205           msb_valid <= '0';
206         end if;
207       else
208         msb_index <= 0;
209         msb_valid <= '0';
210       end if;
211     end process;
212
213     comb_process : process(input_backward_elim, r, reset_n,
          divisor_is_negative, divisor, remainder_valid, remainders,
          msb_valid, divisor, divisor_inv, msb_index)
214       variable v                          :
            input_elimination_reg_type;
215       variable r_j_i                      : signed(
            PIXEL_DATA_WIDTH*2 + PRECISION_SHIFT-1 downto 0);
216       variable r_i_i                      : integer;
217       variable temp                       : integer;
218       --variable r_j_i_divided              : signed (
            PIXEL_DATA_WIDTH*2+PRECISION_SHIFT-1 downto 0);
219       variable inner_product              : signed(
            PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2+PRECISION_SHIFT-1
             downto 0);
220       variable shifted_down_inner_product : signed(
            PIXEL_DATA_WIDTH*2-1 downto 0);
221       --variable r_i_i_halv                 : integer;
222       variable r_i_i_halv                 : signed(
            PIXEL_DATA_WIDTH*2 +PRECISION_SHIFT-1 downto 0);
223       variable divisor_inv_from_lut       : integer range 0 to 2**
            DIV_PRECISION := 0;
224     ---
225     begin
226       v := r;
227
228       if((input_backward_elim.state_reg.state =
            STATE_BACKWARD_ELIMINATION or input_backward_elim.
            state_reg.state = STATE_FORWARD_ELIMINATION) and
            input_backward_elim.valid_data = '1' and remainder_valid
            = '1' and msb_valid = '1') then
229         -- Load data set index_j
230         v.row_j       := input_backward_elim.row_j;
231         v.row_i       := input_backward_elim.row_i;
232         v.inv_row_j   := input_backward_elim.inv_row_j;
233         v.inv_row_i   := input_backward_elim.inv_row_i;
```

```
234         v.index_i      := input_backward_elim.index_i;
235         v.index_j      := input_backward_elim.index_j;
236         v.best_approx  := INITIAL_BEST_APPROX;
237         v.msb_index    := msb_index;
238
239         r_i_i       := to_integer(input_backward_elim.row_i(
                input_backward_elim.index_i));
240         r_i_i_halv := shift_left((shift_right(to_signed(r_i_i,
                r_i_i_halv'length), 1)), PRECISION_SHIFT);
241 --dividing by two, then shifting up again with precision shift.
242
243         r_j_i := shift_left(resize(input_backward_elim.row_j(
                input_backward_elim.index_i), r_j_i'length),
                PRECISION_SHIFT);
244         -- For more precise integer division(in Vivado the
                rounding is always downwards)
245 --        r_j_i := r_j_i+r_i_i_halv;
246
247         if v.msb_index <= DIV_PRECISION then
248           divisor_inv_from_lut := to_integer(divisor_inv);
249         else
250           --Using shifting approach
251           divisor_inv_from_lut := to_integer(divisor_inv);
252
253           -- The best approximation may be either the msb-shifted
                division, or the
254           -- msb+1 shifted division.
255           v.best_approx.remainder         := remainders(v.msb_index
                );
256           v.best_approx.number_of_shifts := v.msb_index;
257           -- The best approximation to the divisor may be larger
                than the divisor.
258           if to_integer(signed(divisor))- to_integer(shift_left(
                to_signed(1, PIXEL_DATA_WIDTH*2), v.best_approx.
                number_of_shifts)) > to_integer(shift_left(to_signed
                (1, PIXEL_DATA_WIDTH*2), v.best_approx.
                number_of_shifts+1))- to_integer(signed(divisor))
                then
259             -- This is a better approximation
260             v.best_approx.remainder         := std_logic_vector(
                    to_signed(to_integer(shift_left(to_signed(1,
                    PIXEL_DATA_WIDTH*2), v.best_approx.number_of_shifts
                    +1))-to_integer(signed(divisor)), PIXEL_DATA_WIDTH
                    *2));
261             v.best_approx.number_of_shifts := v.best_approx.
                    number_of_shifts+1;
262           end if;
263         end if;
264
265         for i in 0 to P_BANDS-1 loop
```

```
266         --inner_product := to_signed(to_integer(
                input_backward_elim.row_i(i))*to_integer(
                r_j_i_divided), inner_product'length);
267         if v.msb_index <= DIV_PRECISION then
268           -- Using lut-table
269           inner_product               := resize(
                  input_backward_elim.row_i(i)* r_j_i*
                  divisor_inv_from_lut, inner_product'length);
270           shifted_down_inner_product := resize(shift_right(
                  inner_product, PRECISION_SHIFT+DIV_PRECISION),
                  shifted_down_inner_product'length);
271           -- To matrix A
272           v.row_j(i)                  := to_signed(to_integer(
                  signed(input_backward_elim.row_j(i)))-to_integer(
                  shifted_down_inner_product), PIXEL_DATA_WIDTH*2);
273
274           inner_product               := resize(
                  input_backward_elim.inv_row_i(i)* r_j_i*
                  divisor_inv_from_lut, inner_product'length);
275           shifted_down_inner_product := resize(shift_right(
                  inner_product, PRECISION_SHIFT+DIV_PRECISION),
                  shifted_down_inner_product'length);
276           -- To matrix A_inv
277           v.inv_row_j(i)              := to_signed(to_integer(
                  input_backward_elim.inv_row_j(i))-to_integer(
                  shifted_down_inner_product), PIXEL_DATA_WIDTH*2);
278
279         else
280           -- Using shifting approach to division
281           inner_product              := shift_right(
                  input_backward_elim.row_i(i)*r_j_i, v.best_approx.
                  number_of_shifts);
282           shifted_down_inner_product := resize(shift_right(
                  inner_product, PRECISION_SHIFT),
                  shifted_down_inner_product'length);
283           -- To matrix A
284           v.row_j(i)                  := to_signed(to_integer(
                  signed(input_backward_elim.row_j(i)))-to_integer(
                  shifted_down_inner_product), PIXEL_DATA_WIDTH*2);
285           inner_product               := shift_right(
                  input_backward_elim.inv_row_i(i)*r_j_i, v.
                  best_approx.number_of_shifts);
286           shifted_down_inner_product := resize(shift_right(
                  inner_product, PRECISION_SHIFT),
                  shifted_down_inner_product'length);
287           -- To matrix A_inv
288           v.inv_row_j(i)              := to_signed(to_integer(
                  input_backward_elim.inv_row_j(i))-to_integer(
                  shifted_down_inner_product), PIXEL_DATA_WIDTH*2);
289
```

```
290         end if;
291       end loop;
292       -- Control signals --
293       v.write_address_odd                := input_backward_elim.
              write_address_odd;
294       v.write_address_even               := input_backward_elim.
              write_address_even;
295       v.flag_write_to_odd_row            := input_backward_elim.
              flag_write_to_odd_row;
296       v.flag_write_to_even_row           := input_backward_elim.
              flag_write_to_even_row;
297       v.state_reg                        := input_backward_elim.
              state_reg;
298       v.valid_data                       := input_backward_elim.
              valid_data;
299       v.forward_elimination_write_state := input_backward_elim.
              forward_elimination_write_state;
300       v.valid_data                       := '1';
301     end if;
302     if(reset_n = '0') then
303       v.index_i    := P_BANDS-1;
304       v.index_j    := P_BANDS-2;
305       v.valid_data := '0';
306     end if;
307     r_in                                                <= v;
308     -- data
309     output_backward_elim.new_row_j                      <= r.
          row_j;
310     output_backward_elim.new_inv_row_j                  <= r.
          inv_row_j;
311     -- control
312     output_backward_elim.state_reg                      <= r.
          state_reg;
313     output_backward_elim.valid_data                     <= r.
          valid_data;
314     output_backward_elim.write_address_even             <= r.
          write_address_even;
315     output_backward_elim.write_address_odd              <= r.
          write_address_odd;
316     output_backward_elim.flag_write_to_even_row         <= r.
          flag_write_to_even_row;
317     output_backward_elim.flag_write_to_odd_row          <= r.
          flag_write_to_odd_row;
318     output_backward_elim.forward_elimination_write_state <= r.
          forward_elimination_write_state;
319   end process;
320
321
322   sequential_process : process(clk)
323   begin
```

```
324     if rising_edge(clk) then
325       if clk_en = '1' then
326         r <= r_in;
327       end if;
328     end if;
329   end process;
330
331 end Behavioral;
```

C.3 BRAM SDP 18kbit

Listing C.3: BRAM

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  entity block_ram is
4    generic(
5      B_RAM_SIZE      : integer := 100;
6      B_RAM_BIT_WIDTH : integer := 32
7      );
8    port (
9      clk           : in  std_logic;
10     aresetn       : in  std_logic;
11     data_in       : in  std_logic_vector(B_RAM_BIT_WIDTH-1
           downto 0);
12     write_enable  : in  std_logic;
13     read_enable   : in  std_logic;
14     read_address  : in  integer range 0 to B_RAM_SIZE-1;
15     write_address : in  integer range 0 to B_RAM_SIZE -1;  --
           added
16     data_out      : out std_logic_vector(B_RAM_BIT_WIDTH-1
           downto 0)
17     );
18 end block_ram;
19 architecture Behavioral of block_ram is
20   type bus_array is array(0 to B_RAM_SIZE-1) of std_logic_vector
       (B_RAM_BIT_WIDTH-1 downto 0);
21   signal b_ram_data : bus_array;
22 begin
23   process(clk)
24   begin
25     if(rising_edge(clk)) then
26       if(write_enable = '1') then
27         b_ram_data(write_address) <= data_in;
28       end if;
29     end if;
30   end process;
31   process(clk)
```

```
32    begin
33      if(rising_edge(clk)) then
34        if(read_enable = '1') then
35          data_out <= b_ram_data(read_address);
36        end if;
37      end if;
38
39    end process;
40  end Behavioral;
```

C.4 Package Common types and functions

Listing C.4: Common types and functions

```
1  ----------------------------------------------------------------------------------
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date: 01/29/2018 12:31:59 PM
6  -- Design Name:
7  -- Module Name: common_types_and_functions - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool Versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------------------------------------------------
20
21
22 library IEEE;
23 use IEEE.STD_LOGIC_1164.all;
24 use ieee.numeric_std.all;
25
26 library work;
27
28
29 package Common_types_and_functions is
30   -- N_PIXELS is the number of pixels in the hyperspectral image
31   constant N_PIXELS                          : integer range 0
       to 628864 := 628864;  -- 578 pixels per row * 1088 rows
```

```
32    -- P_BANDS  is the number of spectral bands
33    constant P_BANDS                                : integer range 0
          to 100     := 12;
34    --constant P_BANDS : integer := 100;
35    -- K is size of the kernel used in LRX.
36    constant K                                      : integer;
37    -- PIXEL_DATA_WIDTH is the width of the raw input data from
          the HSI
38    constant PIXEL_DATA_WIDTH                       : integer range 0
          to 16      := 16;
39    constant NUMBER_OF_WRITES_PER_CYCLE             : integer range 0
          to 2       := 2;
40    constant BRAM_TDP_ADDRESS_WIDTH                 : integer range 0
          to 10      := 10;
41    -- component generics
42    constant B_RAM_SIZE                             : integer
                              := 100;
43  -- Need to be 33 bit due to updating(adding) of two 32 bit
      variables. Is 33 bit necessary? Precision  question.
44    constant B_RAM_BIT_WIDTH                        : integer
                              := 32;
45    -- Time from issuing write in top-level inverse to data is
          possible to read
46    -- from BRAM:
47    constant B_RAM_WAIT_CLK_CYCLES                  : integer range 0
          to 3       := 3;
48    constant ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA : integer range 0
          to 6       := 4;
49  --per clock cycle
50    type matrix is array (natural range <>, natural range <>) of
          std_logic_vector(15 downto 0);
51    -- for correlation results
52    type matrix_32 is array (natural range <>, natural range <>)
          of std_logic_vector(31 downto 0);
53
54    type row_array is array (0 to P_BANDS-1) of signed(
          PIXEL_DATA_WIDTH*2 -1 downto 0);
55
56    -- drive signals
57    constant STATE_IDLE_DRIVE                       :
          std_logic_vector(2 downto 0) := "000";
58    constant STATE_FORWARD_ELIM_TRIANGULAR_FINISHED :
          std_logic_vector(2 downto 0) := "001";
59    constant STATE_FORWARD_ELIMINATION_FINISHED     :
          std_logic_vector(2 downto 0) := "010";
60    constant STATE_BACKWARD_ELIMINATION_FINISHED    :
          std_logic_vector(2 downto 0) := "011";
61    constant STATE_LAST_DIVISION_FINISHED           :
          std_logic_vector(2 downto 0) := "111";
62
```

```
63    -- start signals for fsm
64    constant IDLING                              : std_logic_vector(1
          downto 0) := "00";
65    constant START_FORWARD_ELIMINATION           : std_logic_vector(1
          downto 0) := "01";
66    constant START_BACKWARD_ELIMINATION          : std_logic_vector(1
          downto 0) := "10";
67    constant START_IDENTITY_MATRIX_BUILDING      : std_logic_vector(1
          downto 0) := "11";
68
69    -- Forward-elimination specific signals
70
71    constant START_FORWARD_ELIM_TRIANGULAR : std_logic_vector (1
          downto 0) := "10";
72    constant START_FORWARD_ELIM_CORE       : std_logic_vector(1
          downto 0)  := "11";
73    constant STATE_FORWARD_TRIANGULAR      : std_logic_vector(1
          downto 0)  := "10";
74    constant STATE_FORWARD_ELIM            : std_logic_vector(1
          downto 0)  := "11";
75
76    -- Different state machines used in the design
77    type state_type is (STATE_IDLE, STATE_STORE_CORRELATION_MATRIX
          , STATE_FORWARD_ELIMINATION, STATE_BACKWARD_ELIMINATION,
          STATE_LAST_DIVISION, STATE_OUTPUT_INVERSE_MATRIX);
78
79    type elimination_write_state is (STATE_IDLE, FIRST_ELIMINATION
          , ODD_j_WRITE, EVEN_j_WRITE, EVEN_i_START, ODD_i_START);
80    type forward_elimination_write_state_type is (STATE_IDLE,
          CHECK_DIAGONAL_ELEMENT_IS_ZERO, SWAP_ROWS, EVEN_j_WRITE,
          ODD_j_WRITE);
81    type last_division_write_state_type is (STATE_IDLE,
          EVEN_i_WRITE, ODD_i_WRITE);
82
83    type remainder_after_approximation_record is record
84      remainder         : std_logic_vector(PIXEL_DATA_WIDTH*2-1
            downto 0);  -- For PIXEL_DATA_WIDTH of 16
85      number_of_shifts  : integer range 0 to 31;
86      remainder_valid   : std_logic;
87    end record;
88
89    type reg_state_type is record
90      state : state_type;
91    --drive                          : std_logic_vector(2 downto
          0);
92    --fsm_start_signal               : std_logic_vector(1 downto
          0);
93    --inner_loop_iter_finished       : std_logic;
94    --inner_loop_last_iter_finished  : std_logic;
95    --start_inner_loop               : std_logic;
```

```
96    --forward_elim_ctrl_signal        : std_logic_vector(1 downto
          0);
97    --forward_elim_state_signal       : std_logic_vector(1 downto
          0);
98    --flag_forward_core_started       : std_logic;
99    --flag_forward_triangular_started : std_logic;
100   end record;
101
102
103   type input_elimination_reg_type is record
104     row_j                                : row_array;
105     row_i                                : row_array;
106     row_even                             : row_array;
107     row_odd                              : row_array;
108     inv_row_even                         : row_array;
109     inv_row_odd                          : row_array;
110     inv_row_j                            : row_array;
111     inv_row_i                            : row_array;
112     state_reg                            : reg_state_type;
113     index_i                              : integer range 0 to
            P_BANDS -1;
114     index_j                              : integer range 0 to
            P_BANDS -1;
115     valid_data                           : std_logic;
116     write_address_even                   : integer range 0 to
            P_BANDS/2-1;
117     write_address_odd                    : integer range 0 to
            P_BANDS/2-1;
118     read_address                         : integer range 0 to
            P_BANDS/2-1;
119     flag_write_to_even_row               : std_logic;
120     flag_write_to_odd_row                : std_logic;
121     write_enable_odd                     : std_logic;
122     write_enable_even                    : std_logic;
123     forward_elimination_write_state      :
            forward_elimination_write_state_type;
124     address_row_i                        : integer range 0 to
            P_BANDS/2-1;
125     address_row_j                        : integer range 0 to
            P_BANDS/2-1;
126     flag_prev_row_i_at_odd_row           : std_logic;  --two
            cycles ahead
127     flag_prev_row_j_at_odd_row           : std_logic;  -- needed
            for flip rows
128     flag_start_swapping_rows             : std_logic;  -- used in
             forward elimination
129     flag_started_swapping_rows           : std_logic;  -- used in
             flip rows and forward elimination
130     flag_wrote_swapped_rows_to_BRAM      : std_logic;  --
131     flag_first_data_elimination          : std_logic;
```

```
132     read_address_even                    : integer range 0 to
            P_BANDS/2-1;
133     read_address_odd                     : integer range 0 to
            P_BANDS/2-1;
134     read_enable                          : std_logic;
135     best_approx                          :
            remainder_after_approximation_record;
136     msb_index                            : integer range 0 to 31;
137     index_i_two_cycles_ahead             : integer range 0 to
            P_BANDS-1;
138     index_j_two_cycles_ahead             : integer range 0 to
            P_BANDS-1;
139     read_address_row_i_two_cycles_ahead : integer range 0 to
            P_BANDS/2-1;
140     wait_counter                         : integer range 0 to 3;
141     flag_waiting_for_bram_update         : std_logic;
142   end record;
143   type inverse_top_level_reg_type is record
144     row_j                                                    :
            row_array;
145     row_i                                                    :
            row_array;
146     inv_row_j                                                :
            row_array;
147     inv_row_i                                                :
            row_array;
148     state_reg                                                :
            reg_state_type;
149     index_i_two_cycles_ahead                                 :
            integer range 0 to P_BANDS-1;
150     index_j_two_cycles_ahead                                 :
            integer range 0 to P_BANDS-1;
151     index_i                                                  :
            integer range 0 to P_BANDS -1;
152     index_j                                                  :
            integer range 0 to P_BANDS -1;
153     valid_data                                               :
            std_logic;
154     write_address_even                                       :
            integer range 0 to P_BANDS/2-1;
155     write_address_odd                                        :
            integer range 0 to P_BANDS/2-1;
156     read_address_even                                        :
            integer range 0 to P_BANDS/2-1;
157     read_address_odd                                         :
            integer range 0 to P_BANDS/2-1;
158     bram_write_data_M                                        :
            std_logic_vector(P_BANDS*PIXEL_DATA_WIDTH*2*2 -1 downto
            0);
159     bram_write_data_M_inv                                    :
```

```
        std_logic_vector(P_BANDS*PIXEL_DATA_WIDTH*2*2 -1 downto
        0);
160     write_enable_even                                   :
        std_logic; -- Remove?
161     write_enable_odd                                    :
        std_logic; -- Remove?
162     read_enable                                         :
        std_logic;
163     writes_done_on_column                               :
        std_logic_vector(0 downto 0); --should
164     --be size log2(P_BANDS/2)downto 0. Need to edit the size
        manually if
165     --changing P_BANDS.
166     flag_first_data_elimination                         :
        std_logic;
167     flag_waited_one_clk                                 :
        std_logic;
168     flag_first_memory_request                           :
        std_logic; -- between each state shift
169     flag_write_to_odd_row                               :
        std_logic; -- row_j might be on both odd and
170                                     -- even rows.
171     flag_write_to_even_row                              :
        std_logic; -- sometimes its necessary to write
172                                     -- both rows.
173     --^ Needed for forward elimination
174     elimination_write_state                             :
        elimination_write_state;
175     read_address_row_i_two_cycles_ahead                 :
        integer range 0 to P_BANDS/2-1;
176     -- read address of the row i
177     address_row_i                                       :
        integer range 0 to P_BANDS/2-1;
178     flag_prev_row_i_at_odd_row                          :
        std_logic; --two cycles ahead
179     flag_wr_row_i_at_odd_row                            :
        std_logic;
180     ---*
181     flag_finished_sending_data_to_BRAM_one_cycle_ago    :
        std_logic;
182     flag_finished_sending_data_to_BRAM_two_cycles_ago   :
        std_logic;
183     flag_finished_sending_data_to_BRAM_three_cycles_ago :
        std_logic;
184     flag_last_read_backward_elimination                 :
        std_logic;
185
186     flag_first_iter_backward_elim : std_logic;
187
188     wait_counter                  : integer range 0 to 3;
```

```
189     flag_waiting_for_bram_update   : std_logic;
190     -- Needed for last division:
191     last_division_write_state      :
            last_division_write_state_type;
192     counter_output_inverse_matrix : integer range 0 to P_BANDS
            /2-1;
193
194   end record;
195
196   type inverse_output_reg_type is record
197     --outputting two rows of the inverse matrix per cycle:
198     two_inverse_rows : std_logic_vector(P_BANDS*PIXEL_DATA_WIDTH
            *2*2-1 downto 0);
199     valid_data       : std_logic;
200     address          : integer range 0 to P_BANDS/2-1;
201   end record;
202
203   type output_forward_elimination_reg_type is record
204     row_j                                : row_array;
205     row_i                                : row_array;
206     inv_row_j                            : row_array;
207     inv_row_i                            : row_array;
208     index_i                              : integer range 0 to
            P_BANDS -1;
209     index_j                              : integer range 0 to
            P_BANDS -1;
210     state_reg                            : reg_state_type;
211     r_addr_next                          : integer range 0 to
            P_BANDS/2-1;
212     write_address_even                   : integer range 0 to
            P_BANDS/2-1;
213     write_address_odd                    : integer range 0 to
            P_BANDS/2-1;
214     valid_data                           : std_logic;
215     flag_write_to_odd_row                : std_logic;  -- row_j
            might be on both odd and
216                                                        -- even
                                                           rows.
217     flag_write_to_even_row               : std_logic;  --
            sometimes its necessary to write
218                                                        -- both
                                                           rows.
219     write_enable_even                    : std_logic;
220     write_enable_odd                     : std_logic;
221     flag_prev_row_i_at_odd_row           : std_logic;  --two
            cycles ahead
222     read_address_even                    : integer range 0 to
            P_BANDS/2-1;
223     read_address_odd                     : integer range 0 to
            P_BANDS/2-1;
```

```
224     read_enable                          : std_logic;
225     forward_elimination_write_state      :
            forward_elimination_write_state_type;
226     flag_started_swapping_rows           : std_logic;  -- used in
            flip rows and forward elimination
227     wait_counter                         : integer range 0 to 3;
228     index_i_two_cycles_ahead             : integer range 0 to
            P_BANDS-1;
229     index_j_two_cycles_ahead             : integer range 0 to
            P_BANDS-1;
230     read_address_row_i_two_cycles_ahead : integer range 0 to
            P_BANDS/2-1;
231   end record;
232
233   type output_backward_elimination_reg_type is record
234     new_row_j                            : row_array;
235     new_inv_row_j                        : row_array;
236     r_addr_next                          : integer range 0 to P_BANDS
            /2-1;
237     write_address_even                   : integer range 0 to P_BANDS
            /2-1;
238     write_address_odd                    : integer range 0 to P_BANDS
            /2-1;
239     valid_data                           : std_logic;
240     state_reg                            : reg_state_type;
241     flag_write_to_odd_row                : std_logic;  -- row_j might
            be on both odd and
242                                                        -- even rows.
243     flag_write_to_even_row               : std_logic;  -- sometimes
            its necessary to write
244                                                        -- both rows.
245     write_enable_even                    : std_logic;
246     write_enable_odd                     : std_logic;
247     forward_elimination_write_state :
            forward_elimination_write_state_type;
248   end record;
249
250   type input_last_division_reg_type is record
251     row_i                    : row_array;
252     inv_row_i                : row_array;
253     state_reg                : reg_state_type;
254     index_i                  : integer range 0 to P_BANDS -1;
255     flag_write_to_even_row : std_logic;  -- Maximum need to
            write one row at the
256     -- time in STATE LAST DIVISION
257     valid_data               : std_logic;
258     write_address_even       : integer range 0 to P_BANDS/2-1;
259     write_address_odd        : integer range 0 to P_BANDS/2-1;
260     --
261     best_approx              :
```

```
        remainder_after_approximation_record;
262     msb_index              : integer range 0 to 31;
263   end record;
264
265   type output_last_division_reg_type is record
266     new_inv_row_i          : row_array;
267     valid_data             : std_logic;
268     index_i                : integer range 0 to P_BANDS -1;
269     write_address_even     : integer range 0 to P_BANDS/2-1;
270     write_address_odd      : integer range 0 to P_BANDS/2-1;
271     flag_write_to_even_row : std_logic;
272     state_reg              : reg_state_type;
273   end record;
274
275   constant INITIAL_BEST_APPROX :
        remainder_after_approximation_record := (
276     remainder        => (PIXEL_DATA_WIDTH*2-1 => '0', others =>
          '1'),
277     number_of_shifts => 0,
278     remainder_valid  => '0'
279     );
280   constant DIV_PRECISION : integer range 0 to 31 := 17;
281
282   function log2(i                    : natural) return integer;
283   function sel (n                    : natural) return integer;
284   function create_identity_matrix (n : natural) return matrix_32
        ;
285
286 end Common_types_and_functions;
287
288 package body Common_types_and_functions is
289   -- Found in SmallSat project description:
290   --constant P_BANDS :  integer := 100;
291
292   constant K : integer := 0;
293
294
295   function log2(i : natural) return integer is
296     variable temp    : integer := i;
297     variable ret_val : integer := 0;
298   begin
299     while (temp > 1) loop
300       ret_val := ret_val + 1;
301       temp    := temp / 2;
302     end loop;
303
304     return ret_val;
305   end function;
306
307   function create_identity_matrix(n : natural) return matrix_32
```

```
      is
308     variable M_identity_matrix : matrix_32(0 to P_BANDS-1, 0 to
          P_BANDS-1);
309   begin
310     M_identity_matrix := (others => (others => (others => '0')))
          ;
311     for i in 0 to n-1 loop
312       M_identity_matrix(i, i) := std_logic_vector(to_unsigned(1,
            32));
313     end loop;
314     return M_identity_matrix;
315   end function;
316
317   function sel(n : natural) return integer is
318   begin
319     return n;
320   end function;
321
322
323
324 end Common_types_and_functions;
```

C.5 Swap rows

Listing C.5: Swap rows

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.Common_types_and_functions.all;
7
8  -- This module is used in the forward elimination
9  -- It flips rows i and j if a zero is detected in row i, and row
       j does not
10 -- contain a 0 at index j
11
12 entity swap_rows_module is
13   port(clk               : in  std_logic;
14        reset_n           : in  std_logic;
15        clk_en            : in  std_logic;
16        input_swap_rows   : in  input_elimination_reg_type;
17        output_swap_rows : out
              output_forward_elimination_reg_type
18        );
19 end swap_rows_module;
20
21 architecture Behavioral of swap_rows_module is
```

```
22    signal r, r_in : input_elimination_reg_type;
23  begin
24
25    comb_process : process(input_swap_rows, r, reset_n)
26      variable v : input_elimination_reg_type;
27
28    begin
29      v := r;
30      case input_swap_rows.forward_elimination_write_state is
31        when SWAP_ROWS =>
32          if input_swap_rows.flag_start_swapping_rows = '1' then
33            v.flag_started_swapping_rows       := '1';
34            v.flag_wrote_swapped_rows_to_BRAM := '0';
35            v.index_j                          := input_swap_rows.
                  index_j;
36            v.index_i                          := input_swap_rows.
                  index_i;
37            v.row_i                            := input_swap_rows.
                  row_i;
38            v.row_j                            := input_swap_rows.
                  row_j;
39            v.address_row_i                    := input_swap_rows.
                  address_row_i;
40            v.address_row_j                    := input_swap_rows.
                  address_row_j;
41            v.flag_write_to_even_row           := input_swap_rows.
                  flag_write_to_even_row;
42            v.flag_write_to_odd_row            := input_swap_rows.
                  flag_write_to_odd_row;
43            v.flag_prev_row_i_at_odd_row       := input_swap_rows.
                  flag_prev_row_i_at_odd_row;
44            v.flag_prev_row_j_at_odd_row       := not(
                  input_swap_rows.flag_prev_row_i_at_odd_row);
45            if v.row_j(v.index_j) /= 0 then
46              -- flip the rows, write_to_BRAM
47              v.row_i                            := v.row_j;
48              v.row_j                            := input_swap_rows
                    .row_i;
49              v.flag_wrote_swapped_rows_to_BRAM := '1';
50              v.flag_write_to_even_row           := '1';
51              v.flag_write_to_odd_row            := '1';
52              if v.flag_prev_row_i_at_odd_row = '1' then
53                v.write_address_odd  := v.address_row_i;
54                v.write_address_even := v.address_row_j;
55              else
56                v.write_address_even := v.address_row_i;
57                v.write_address_odd  := v.address_row_j;
58              end if;
59            else
60              -- need to check next row_j. Issue reads for two
```

```
                cycles ahead
61              if v.index_j <= P_BANDS-3 then
62                if input_swap_rows.flag_write_to_even_row = '1'
                      then
63                  -- need to read an odd row, has already read the
                        even row with the
64                  -- same address as this odd row
65                  v.read_enable       := '1';
66                  v.read_address_even := input_swap_rows.
                        read_address_even;
67                  v.read_address_odd  := input_swap_rows.
                        read_address_odd;
68                else
69                  -- need to read an even row
70                  v.read_enable       := '1';
71                  v.read_address_even := input_swap_rows.
                        read_address_even+1;
72                  v.read_address_odd  := input_swap_rows.
                        read_address_odd +1;
73                end if;
74              --end if;
75              else
76                --all reads has been issued. Need to wait to see
                      if some of the latest
77                --rows can be swapped
78                v.read_enable := '0';
79              end if;
80            end if;
81
82            if r.flag_started_swapping_rows = '1' and r.
                  flag_wrote_swapped_rows_to_BRAM = '0' then
83              --need to check if index_i and index_j is at two
                    even or two odd
84              --indexes. If so, the writes two BRAM will have to
                    continue in two cycles.
85              v.index_j                       := r.index_j +1;
86              v.flag_prev_row_j_at_odd_row := not(r.
                    flag_prev_row_j_at_odd_row);
87              v.flag_write_to_even_row     := not(r.
                    flag_write_to_even_row);
88              v.flag_write_to_odd_row      := not(r.
                    flag_write_to_odd_row);
89              v.row_j                         := input_swap_rows.
                    row_j;  --this is outputted directly from BRAMS.
90              if v.flag_prev_row_j_at_odd_row = '0' then
91                --current row j is at an even index, need to
                      update row_j address
92                v.address_row_j := r.address_row_j+1;
93              end if;
94              if v.row_j(v.index_j) /= 0 then
```

```
95                --flip the rows, write to BRAM
96                v.row_i                                := v.row_j;
97                v.row_j                                :=
                      input_swap_rows.row_i;  --this correct?
98                v.flag_wrote_swapped_rows_to_BRAM :=  '1';
99                v.flag_write_to_even_row          :=  '1';
100               v.flag_write_to_odd_row           :=  '1';
101               if v.flag_prev_row_i_at_odd_row = '1' then
102                 v.write_address_odd  := v.address_row_i;
103                 v.write_address_even := v.address_row_j;
104               else
105                 v.write_address_even := v.address_row_i;
106                 v.write_address_odd  := v.address_row_j;
107               end if;
108             else
109               -- need to read new data
110               if v.index_j <= P_BANDS-3 then
111                 v.valid_data := '0';
112                 if v.flag_write_to_even_row = '1' then
113                   --need to read an odd row, has already read
                          the even row with the
114                   -- same address as this odd row
115                   v.read_enable       := '1';
116                   v.read_address_even := v.read_address_even;
117                   v.read_address_odd  := v.read_address_odd;
118                 else
119                   -- need to read an even row
120                   v.read_enable       := '1';
121                   v.read_address_even := v.read_address_even+1;
122                   v.read_address_odd  := v.read_address_odd +1;
123                 end if;
124               end if;
125             end if;
126           elsif v.index_j = 0 then
127             -- The loop has continued without any swap of rows
128             -- The matrix is singular.
129             v.valid_data := '1';
130           else
131             -- all reads has been issued. Need to wait to see if
                    some of the latest
132             -- rows can be swapped
133             v.read_enable := '0';
134           end if;
135         end if;
136         if r.flag_wrote_swapped_rows_to_BRAM = '1' then
137           -- valid_data is used to signal that the swapping is
                  done
138           -- and that the data is finished written to BRAM
139           v.valid_data             := '1';
140           v.flag_write_to_even_row := '0';
```

```
141             v.flag_write_to_odd_row  := '0';
142             v.read_enable            := '0';
143             -- Start issuing reads
144             if r.flag_prev_row_i_at_odd_row = '0' then
145               -- This means that row j two cycles ahead is at an
                      odd index
146               -- because the first row j is at an odd index
147               v.read_address_odd  := r.address_row_i;
148               v.read_address_even := r.address_row_i;
149             else
150               -- This means that row j two cycles ahead is at an
                      even index
151               -- because the first row j is at an even index
152               v.read_address_odd  := r.address_row_i;
153               v.read_address_even := r.address_row_i+1;
154             end if;
155           end if;
156         when others =>
157           v.index_i                     := 0;
158           v.index_j                     := 1;
159           v.valid_data                  := '0';
160           v.address_row_i               := 0;
161           v.address_row_j               := 1;
162           v.flag_write_to_even_row      := '0';
163           v.flag_write_to_odd_row       := '0';
164           v.flag_started_swapping_rows := '0';
165       end case;
166       if(reset_n = '0') then
167         v.index_i                     := 0;
168         v.index_j                     := 1;
169         v.valid_data                  := '0';
170         v.address_row_i               := 0;
171         v.address_row_j               := 1;
172         v.flag_write_to_even_row      := '0';
173         v.flag_write_to_odd_row       := '0';
174         v.flag_started_swapping_rows := '0';
175       end if;
176       r_in                                        <= v;
177       -- This module needs to write
178 --    output_forward_elimination
179       output_swap_rows.row_j                  <= r.row_j;
180       output_swap_rows.row_i                  <= r.row_i;
181       output_swap_rows.read_address_odd           <= r.
              read_address_odd;
182       output_swap_rows.read_address_even          <= r.
              read_address_even;
183       output_swap_rows.flag_write_to_odd_row      <= r.
              flag_write_to_odd_row;
184       output_swap_rows.flag_write_to_even_row     <= r.
              flag_write_to_even_row;
```

```
185     output_swap_rows.read_enable                <= r.read_enable
            ;
186     output_swap_rows.valid_data                 <= r.valid_data;
187     output_swap_rows.write_address_odd          <= r.
            write_address_odd;
188     output_swap_rows.write_address_even         <= r.
            write_address_even;
189     output_swap_rows.state_reg                  <= r.state_reg;
190     output_swap_rows.flag_prev_row_i_at_odd_row <= r.
            flag_prev_row_i_at_odd_row;
191
192   end process;
193
194
195   sequential_process : process(clk, clk_en)
196   begin
197     if(rising_edge(clk) and clk_en = '1') then
198       r <= r_in;
199     end if;
200   end process;
201
202 end Behavioral;
```

C.6 ACAD inverse

Listing C.6: ACAD inverse

```
1  ----------------------------------------------------------------------------------
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date: 02/07/2018 02:19:19 PM
6  -- Design Name:
7  -- Module Name: inverse_matrix - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool Versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------------------------------------------------
```

```
20
21
22 library IEEE;
23 use IEEE.STD_LOGIC_1164.all;
24
25
26 library IEEE;
27 use IEEE.STD_LOGIC_1164.all;
28 use ieee.numeric_std.all;
29
30
31 library work;
32 use work.Common_types_and_functions.all;
33
34 -- Uncomment the following library declaration if using
35 -- arithmetic functions with Signed or Unsigned values
36 --use IEEE.NUMERIC_STD.ALL;
37
38 -- Uncomment the following library declaration if instantiating
39 -- any Xilinx leaf cells in this code.
40 --library UNISIM;
41 --use UNISIM.VComponents.all;
42
43 -- This entity is the top-level for computing the inverse of a
     matrix
44 -- It is written using the two-step method, as described by Jiri
     Gaisler,
45 entity inverse_matrix is
46   port (reset_n                : in  std_logic;
47         clk_en                 : in  std_logic;
48         clk                    : in  std_logic;
49         valid                  : in  std_logic;  -- connect this
              to valid_out from
50         -- correlation module
51         -- assumes that data are inputted row-wise, two rows at
              the time
52         din                    : in  std_logic_vector(P_BANDS*
              PIXEL_DATA_WIDTH*2*2 -1 downto 0);
53         --increases by one for every two write to BRAM:
54         writes_done_on_column : in  std_logic_vector(log2(
              P_BANDS/2) downto 0);
55         -- outputting two and two rows of the inverse matrix
56         inverse_rows           : out inverse_output_reg_type
57         );
58 end inverse_matrix;
59
60 architecture Behavioral of inverse_matrix is
61   signal r, r_in                : inverse_top_level_reg_type;
62   signal output_backward_elim   :
       output_backward_elimination_reg_type;
```

```
63    signal output_forward_elim    :
          output_forward_elimination_reg_type;
64    signal output_last_division   : output_last_division_reg_type;
65    signal data_out_brams_M       : std_logic_vector(P_BANDS*
          PIXEL_DATA_WIDTH*2*2-1 downto 0);
66    signal data_out_brams_M_inv   : std_logic_vector(P_BANDS*
          PIXEL_DATA_WIDTH*2*2-1 downto 0);
67    -- write address for 18kbit BRAMs storing even indexes of the
          matrices
68    signal write_address_even     : integer range 0 to B_RAM_SIZE
          -1;
69    -- write address for 18kbit BRAMs storing odd row indexes of
          the matrices
70    signal write_address_odd      : integer range 0 to B_RAM_SIZE
          -1;
71    signal read_address_even      : integer range 0 to B_RAM_SIZE
          -1;
72    signal read_address_odd       : integer range 0 to B_RAM_SIZE
          -1;
73    signal write_enable_odd       : std_logic := '0';
74    signal write_enable_even      : std_logic := '0';
75    -- for BRAMs containing the inverse matrix:
76    signal write_enable_inv_odd   : std_logic := '0';
77    signal write_enable_inv_even  : std_logic := '0';
78
79    -- input record to the forward elimination module
80    signal input_forward_elimination : input_elimination_reg_type;
81    -- input record to the elimination core
82    signal input_elimination         : input_elimination_reg_type;
83    signal input_last_division       :
          input_last_division_reg_type;
84    -- index of the top bit of the even rows in data out from the
          BRAMs:
85    constant EVEN_ROW_TOP_INDEX      : integer range 0 to P_BANDS*
          PIXEL_DATA_WIDTH*2-1    := P_BANDS*PIXEL_DATA_WIDTH*2 -1;
86  -- index of the topper bit of the odd rows in data out from the
      BRAMs:
87    constant ODD_ROW_TOP_INDEX       : integer range 0 to 2*
          P_BANDS*PIXEL_DATA_WIDTH*2-1 := 2*P_BANDS*PIXEL_DATA_WIDTH
          *2 -1;
88  begin
89
90    gen_BRAM_18_for_storing_correlation_matrix : for i in 0 to
          P_BANDS-1 generate
91      -- Generating N_BRAMS = P_BANDS BRAM 36 kbits.
92      -- Storing matrix M in the Gauss Jordan elimination
93      signal data_in_even_i, data_in_odd_i, data_out_even_i,
            data_out_odd_i : std_logic_vector(B_RAM_BIT_WIDTH -1
            downto 0);
94    begin
```

```
95      -- Block ram row for even row indexes of the correlation
            matrix
96      block_ram_even : entity work.block_ram
97        generic map (
98          B_RAM_SIZE       => B_RAM_SIZE,
99          B_RAM_BIT_WIDTH => B_RAM_BIT_WIDTH)
100       port map (
101         clk           => clk,
102         aresetn       => reset_n,
103         data_in       => data_in_even_i,
104         write_enable  => write_enable_even,
105         read_enable   => r.read_enable,
106         read_address  => read_address_even,
107         write_address => write_address_even,
108         data_out      => data_out_even_i);
109     -- Block ram row for odd row indexes of the correlation
            matrix
110     block_ram_odd : entity work.block_ram
111       generic map (
112         B_RAM_SIZE       => B_RAM_SIZE,
113         B_RAM_BIT_WIDTH => B_RAM_BIT_WIDTH)
114       port map (
115         clk           => clk,
116         aresetn       => reset_n,
117         data_in       => data_in_odd_i,
118         write_enable  => write_enable_odd,
119         read_enable   => r.read_enable,
120         write_address => write_address_odd,
121         read_address  => read_address_odd,
122         data_out      => data_out_odd_i);
123
124     -- Process to control data input to BRAMs.
125     process(valid, r, output_last_division, output_forward_elim,
            output_backward_elim)
126     begin
127       if(r.state_reg.state = STATE_STORE_CORRELATION_MATRIX)
              then
128         data_in_even_i <= r.bram_write_data_M(PIXEL_DATA_WIDTH
              *2-1 + i*PIXEL_DATA_WIDTH*2 downto i*PIXEL_DATA_WIDTH
              *2);
129         data_in_odd_i  <= r.bram_write_data_M(PIXEL_DATA_WIDTH
              *2-1 +i*PIXEL_DATA_WIDTH*2+P_BANDS*PIXEL_DATA_WIDTH*2
               downto i*PIXEL_DATA_WIDTH*2 + P_BANDS*
              PIXEL_DATA_WIDTH*2);
130       elsif (r.state_reg.state = STATE_FORWARD_ELIMINATION) then
131         if output_forward_elim.forward_elimination_write_state =
              SWAP_ROWS then
132           if output_forward_elim.flag_started_swapping_rows =
                '1' then
133             if output_forward_elim.flag_prev_row_i_at_odd_row =
```

```
                    '1' then
134                 -- row i at odd index
135                 data_in_even_i <= std_logic_vector(
                        output_forward_elim.row_j(i));
136                 data_in_odd_i  <= std_logic_vector(
                        output_forward_elim.row_i(i));
137               else
138                 data_in_even_i <= std_logic_vector(
                        output_forward_elim.row_i(i));
139                 data_in_odd_i  <= std_logic_vector(
                        output_forward_elim.row_j(i));
140               end if;
141             else
142               data_in_odd_i  <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
143               data_in_even_i <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
144             end if;
145           -- output_backward_elim in state EVEN_j or ODD_j
146           elsif output_backward_elim.flag_write_to_even_row = '1'
                  and output_backward_elim.valid_data = '1' then
147             -- row_j is at even row
148             data_in_even_i <= std_logic_vector(
                    output_backward_elim.new_row_j(i));
149             data_in_odd_i  <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
150           elsif output_backward_elim.flag_write_to_odd_row = '1'
                  and output_backward_elim.valid_data = '1' then
151             -- row_j is at odd row
152             data_in_odd_i  <= std_logic_vector(
                    output_backward_elim.new_row_j(i));
153             data_in_even_i <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
154           else
155             data_in_odd_i  <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
156             data_in_even_i <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
157           end if;
158         elsif (r.state_reg.state = STATE_BACKWARD_ELIMINATION)
                then
159           if(output_backward_elim.valid_data = '1') then
160             -- Received data from backward elimination.
161             if(output_backward_elim.flag_write_to_odd_row = '1')
                    then
162               -- the j-indexed row is an odd row of the matrix
163               data_in_odd_i  <= std_logic_vector(
                      output_backward_elim.new_row_j(i));
164               data_in_even_i <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
```

```
165             elsif(output_backward_elim.flag_write_to_even_row =
                    '1') then
166               -- the j-indexed row is an even row of the matrix
167               data_in_even_i <= std_logic_vector(
                      output_backward_elim.new_row_j(i));
168               data_in_odd_i  <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
169             else
170               data_in_odd_i  <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
171               data_in_even_i <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
172             end if;
173           else
174             data_in_odd_i  <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
175             data_in_even_i <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
176           end if;
177         elsif(r.state_reg.state = STATE_LAST_DIVISION) then
178           data_in_odd_i  <= std_logic_vector(to_signed(0,
                  PIXEL_DATA_WIDTH*2));
179           data_in_even_i <= std_logic_vector(to_signed(0,
                  PIXEL_DATA_WIDTH*2));
180         else
181           data_in_odd_i  <= std_logic_vector(to_signed(0,
                  PIXEL_DATA_WIDTH*2));
182           data_in_even_i <= std_logic_vector(to_signed(0,
                  PIXEL_DATA_WIDTH*2));
183         end if;
184       end process;
185       data_out_brams_M(PIXEL_DATA_WIDTH*2-1 + i*PIXEL_DATA_WIDTH*2
              downto i*PIXEL_DATA_WIDTH*2)

            <= data_out_even_i;
186   -- even row
187       data_out_brams_M(PIXEL_DATA_WIDTH*2-1 + i*PIXEL_DATA_WIDTH*2
              + P_BANDS*PIXEL_DATA_WIDTH*2 downto i*PIXEL_DATA_WIDTH*2
              + P_BANDS*PIXEL_DATA_WIDTH*2) <= data_out_odd_i;
188   -- odd row
189     end generate;
190
191     gen_BRAM_18_for_storing_inv_correlation_matrix : for i in 0 to
          P_BANDS-1 generate
192       -- Generating N_BRAMS = P_BANDS BRAM 36 kbits.
193       -- Storing the inverse matrix in the Gauss-Jordan
              Elimination
194       signal inv_data_in_even_i, inv_data_in_odd_i,
            inv_data_out_even_i, inv_data_out_odd_i :
            std_logic_vector(B_RAM_BIT_WIDTH -1 downto 0);
```

```
195   begin
196     -- Block ram row for even indexes of the inverse matrix
197     block_ram_even : entity work.block_ram
198       generic map (
199         B_RAM_SIZE      => B_RAM_SIZE,
200         B_RAM_BIT_WIDTH => B_RAM_BIT_WIDTH)
201       port map (
202         clk           => clk,
203         aresetn       => reset_n,
204         data_in       => inv_data_in_even_i,
205         write_enable  => write_enable_inv_even,
206         read_enable   => r.read_enable,
207         read_address  => read_address_even,
208         write_address => write_address_even,
209         data_out      => inv_data_out_even_i);
210     -- Block ram row for odd indexes of the inverse matrix
211     block_ram_odd : entity work.block_ram
212       generic map (
213         B_RAM_SIZE      => B_RAM_SIZE,
214         B_RAM_BIT_WIDTH => B_RAM_BIT_WIDTH)
215       port map (
216         clk           => clk,
217         aresetn       => reset_n,
218         data_in       => inv_data_in_odd_i,
219         write_enable  => write_enable_inv_odd,
220         read_enable   => r.read_enable,
221         write_address => write_address_odd,
222         read_address  => read_address_odd,
223         data_out      => inv_data_out_odd_i);
224
225     -- Process to control data input to BRAMs.
226     process(valid, r, output_forward_elim, output_backward_elim,
            output_last_division)
227     begin
228       if(r.state_reg.state = STATE_STORE_CORRELATION_MATRIX)
            then
229         inv_data_in_even_i <= r.bram_write_data_M_inv(
              PIXEL_DATA_WIDTH*2-1 + i*PIXEL_DATA_WIDTH*2 downto i*
              PIXEL_DATA_WIDTH*2);
230         inv_data_in_odd_i  <= r.bram_write_data_M_inv(
              PIXEL_DATA_WIDTH*2-1 +i*PIXEL_DATA_WIDTH*2+P_BANDS*
              PIXEL_DATA_WIDTH*2 downto i*PIXEL_DATA_WIDTH*2 +
              P_BANDS*PIXEL_DATA_WIDTH*2);
231       elsif r.state_reg.state = STATE_FORWARD_ELIMINATION then
232         if output_forward_elim.forward_elimination_write_state =
              SWAP_ROWS and output_forward_elim.valid_data = '1'
              then
233           -- do nothing actually
234           -- Set data in to zero. Should not overwrite data
                anyway, write enable
```

```
235             -- is not active
236             inv_data_in_odd_i  <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
237             inv_data_in_even_i <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
238           elsif output_backward_elim.flag_write_to_even_row = '1'
                and output_backward_elim.valid_data = '1' then
239             -- row_j is at even row
240             inv_data_in_even_i <= std_logic_vector(
                    output_backward_elim.new_inv_row_j(i));
241             -- data in to odd row is not important; write_enable
                    odd is not
242             -- enabled anyhow
243             inv_data_in_odd_i  <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
244           elsif output_backward_elim.flag_write_to_odd_row = '1'
                and output_backward_elim.valid_data = '1' then
245             -- row_j is at odd row
246             inv_data_in_odd_i  <= std_logic_vector(
                    output_backward_elim.new_inv_row_j(i));
247             -- data in to even row is not important; write_enable
                    even is not
248             -- enabled anyhow
249             inv_data_in_even_i <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
250           else
251             inv_data_in_odd_i  <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
252             inv_data_in_even_i <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
253           end if;
254         elsif (r.state_reg.state = STATE_BACKWARD_ELIMINATION)
              then
255           if(output_backward_elim.valid_data = '1') then
256             -- Received data from backward elimination.
257             if(output_backward_elim.flag_write_to_odd_row = '1')
                    then
258               -- the j-indexed row is an odd row of the matrix
259               inv_data_in_odd_i  <= std_logic_vector(
                      output_backward_elim.new_inv_row_j(i));
260               inv_data_in_even_i <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
261             elsif(output_backward_elim.flag_write_to_even_row =
                    '1') then
262               -- the j-indexed row is an even row of the matrix
263               inv_data_in_even_i <= std_logic_vector(
                      output_backward_elim.new_inv_row_j(i));
264               inv_data_in_odd_i  <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
265             else
```

```
266               inv_data_in_odd_i  <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
267               inv_data_in_even_i <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
268             end if;
269           else
270             inv_data_in_odd_i  <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
271             inv_data_in_even_i <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
272           end if;
273         elsif(r.state_reg.state = STATE_LAST_DIVISION) then
274           if output_last_division.valid_data = '1' then
275             if output_last_division.flag_write_to_even_row = '1'
                    then
276               -- index i is at an even index of the matrix
277               inv_data_in_even_i <= std_logic_vector(
                      output_last_division.new_inv_row_i(i));
278               inv_data_in_odd_i  <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
279             elsif output_last_division.flag_write_to_even_row =
                    '0' then
280               -- index i is at an odd index of the matrix
281               inv_data_in_odd_i  <= std_logic_vector(
                      output_last_division.new_inv_row_i(i));
282               inv_data_in_even_i <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
283             else
284               inv_data_in_odd_i  <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
285               inv_data_in_even_i <= std_logic_vector(to_signed(0,
                      PIXEL_DATA_WIDTH*2));
286             end if;
287           else
288             inv_data_in_odd_i  <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
289             inv_data_in_even_i <= std_logic_vector(to_signed(0,
                    PIXEL_DATA_WIDTH*2));
290           end if;
291         else
292           inv_data_in_odd_i  <= std_logic_vector(to_signed(0,
                  PIXEL_DATA_WIDTH*2));
293           inv_data_in_even_i <= std_logic_vector(to_signed(0,
                  PIXEL_DATA_WIDTH*2));
294         end if;
295       end process;
296   -- DATA outputted from the BRAMs
297       data_out_brams_M_inv(PIXEL_DATA_WIDTH*2-1 + i*
              PIXEL_DATA_WIDTH*2 downto i*PIXEL_DATA_WIDTH*2)
```

```
          <= inv_data_out_even_i;
298       data_out_brams_M_inv(PIXEL_DATA_WIDTH*2-1 + i*
              PIXEL_DATA_WIDTH*2 + P_BANDS*PIXEL_DATA_WIDTH*2 downto i*
              PIXEL_DATA_WIDTH*2 + P_BANDS*PIXEL_DATA_WIDTH*2) <=
              inv_data_out_odd_i;
299
300     end generate;
301
302     top_forward_elimination_1 : entity work.
          top_forward_elimination
303       port map (
304         clk                          => clk,
305         reset_n                      => reset_n,
306         clk_en                       => clk_en,
307         input_forward_elimination    => input_forward_elimination,
308         output_forward_elimination => output_forward_elim);
309
310   --backward_elim_core is used by both forward_elimination and
        backward_elimination.
311     elimination_core_1 : entity work.backward_elim_core
312       port map (
313         clk                   => clk,
314         reset_n               => reset_n,
315         clk_en                => clk_en,
316         input_backward_elim   => input_elimination,
317         output_backward_elim => output_backward_elim);
318
319     top_last_division_1 : entity work.top_last_division
320       port map (
321         clk                   => clk,
322         reset_n               => reset_n,
323         clk_en                => clk_en,
324         input_last_division   => input_last_division,
325         output_last_division => output_last_division);
326
327     -- just_for_test : process(data_out_brams_M, r)
328     --   variable row_even, row_odd, inv_row_even, inv_row_odd :
          row_array;
329     -- begin
330     --   for i in 0 to P_BANDS-1 loop
331     --     row_even(i)     := signed(data_out_brams_M(i*
          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1 downto i*
          PIXEL_DATA_WIDTH*2));
332     --     row_odd(i)      := signed(data_out_brams_M(i*
          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 +EVEN_ROW_TOP_INDEX
            downto i*PIXEL_DATA_WIDTH*2 +EVEN_ROW_TOP_INDEX+1));
333   ---- the odd row
334     --     inv_row_even(i) := signed(data_out_brams_M_inv(i*
          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1 downto i*
          PIXEL_DATA_WIDTH*2));
```

```
335     --      inv_row_odd(i)  := signed(data_out_brams_M_inv(i*
            PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 +EVEN_ROW_TOP_INDEX
             downto i*PIXEL_DATA_WIDTH*2 +EVEN_ROW_TOP_INDEX+1));
336     --    end loop;
337     -- end process;
338
339   -- control address inputs and control inputs to BRAMs
340     control_addresses_and_control_BRAM : process(r,
            output_backward_elim, output_last_division,
            output_forward_elim)
341     begin
342       if(r.state_reg.state = STATE_STORE_CORRELATION_MATRIX) then
343         write_address_even    <= r.write_address_even;
344         write_address_odd     <= r.write_address_odd;
345         read_address_even     <= r.read_address_even;
346         read_address_odd      <= r.read_address_odd;
347         write_enable_even     <= '1';
348         write_enable_odd      <= '1';
349         write_enable_inv_even <= '1';
350         write_enable_inv_odd  <= '1';
351       elsif (r.state_reg.state = STATE_FORWARD_ELIMINATION) then
352         -- Set read addresses to output from top elimination
353         read_address_even <= output_forward_elim.read_address_even
                ;
354         read_address_odd  <= output_forward_elim.read_address_odd;
355         if output_forward_elim.forward_elimination_write_state =
                SWAP_ROWS then
356           write_address_even    <= output_forward_elim.
                  write_address_even;
357           write_address_odd     <= output_forward_elim.
                  write_address_odd;
358           write_enable_even     <= output_forward_elim.
                  flag_write_to_even_row;
359           write_enable_odd      <= output_forward_elim.
                  flag_write_to_odd_row;
360           write_enable_inv_even <= '0';
361           write_enable_inv_odd  <= '0';
362         elsif (output_backward_elim.
                  forward_elimination_write_state = EVEN_j_WRITE or
                  output_backward_elim.forward_elimination_write_state =
                  ODD_j_WRITE) and output_backward_elim.valid_data = '1'
                  then
363           write_address_even    <= output_backward_elim.
                  write_address_even;
364           write_address_odd     <= output_backward_elim.
                  write_address_odd;
365           write_enable_even     <= output_backward_elim.
                  flag_write_to_even_row;
366           write_enable_odd      <= output_backward_elim.
                  flag_write_to_odd_row;
```

```
367           write_enable_inv_even <= output_backward_elim.
                  flag_write_to_even_row;
368           write_enable_inv_odd  <= output_backward_elim.
                  flag_write_to_odd_row;
369         else
370           write_enable_even     <= '0';
371           write_enable_odd      <= '0';
372           write_enable_inv_even <= '0';
373           write_enable_inv_odd  <= '0';
374           write_address_even    <= 0;
375           write_address_odd     <= 0;
376         end if;
377
378       elsif(r.state_reg.state = STATE_BACKWARD_ELIMINATION) then
379         read_address_even <= r.read_address_even;
380         read_address_odd  <= r.read_address_odd;
381         if(output_backward_elim.valid_data = '1') then
382           -- Received data from backward elimination.
383           if(output_backward_elim.flag_write_to_odd_row = '1')
                  then
384             -- the j-indexed row is an odd row of the matrix
385             write_enable_inv_odd  <= '1';
386             write_enable_inv_even <= '0';
387             write_enable_odd      <= '1';
388             write_enable_even     <= '0';
389             write_address_even    <= 0;
390             write_address_odd     <= output_backward_elim.
                    write_address_odd;
391           elsif(output_backward_elim.flag_write_to_even_row = '1')
                  then
392             -- the j-indexed row is an even row of the matrix
393             write_enable_inv_odd  <= '0';
394             write_enable_inv_even <= '1';
395             write_enable_odd      <= '0';
396             write_enable_even     <= '1';
397             write_address_even    <= output_backward_elim.
                    write_address_even;
398             write_address_odd     <= 0;   -- To avoid latches
399           else
400             write_enable_inv_odd  <= '0';
401             write_enable_inv_even <= '0';
402             write_enable_odd      <= '0';
403             write_enable_even     <= '0';
404             write_address_even    <= 0;
405             write_address_odd     <= 0;
406           end if;
407         else
408           write_enable_inv_odd  <= '0';
409           write_enable_inv_even <= '0';
410           write_enable_even     <= '0';
```

```
411           write_enable_odd      <= '0';
412           write_address_even    <= 0;
413           write_address_odd     <= 0;      -- To avoid latches
414         end if;
415       elsif r.state_reg.state = STATE_LAST_DIVISION then
416         read_address_odd  <= r.read_address_odd;
417         read_address_even <= r.read_address_even;
418         if output_last_division.valid_data = '1' then
419           write_enable_inv_odd  <= not(output_last_division.
                flag_write_to_even_row);
420           write_enable_inv_even <= output_last_division.
                flag_write_to_even_row;
421           write_enable_odd      <= not(output_last_division.
                flag_write_to_even_row);
422           write_enable_even     <= output_last_division.
                flag_write_to_even_row;
423           write_address_even    <= output_last_division.
                write_address_even;
424           write_address_odd     <= output_last_division.
                write_address_odd;
425         else
426           write_enable_inv_even <= '0';
427           write_enable_inv_odd  <= '0';
428           write_enable_even     <= '0';
429           write_enable_odd      <= '0';
430           write_address_even    <= 0;
431           write_address_odd     <= 0;
432         end if;
433       elsif r.state_reg.state = STATE_OUTPUT_INVERSE_MATRIX then
434         read_address_even     <= r.read_address_even;
435         read_address_odd      <= r.read_address_odd;
436         write_enable_inv_even <= '0';
437         write_enable_inv_odd  <= '0';
438         write_enable_even     <= '0';
439         write_enable_odd      <= '0';
440         write_address_even    <= 0;
441         write_address_odd     <= 0;
442       else
443         read_address_even     <= 0;
444         read_address_odd      <= 0;
445         write_enable_inv_even <= '0';
446         write_enable_inv_odd  <= '0';
447         write_enable_even     <= '0';
448         write_enable_odd      <= '0';
449         write_address_even    <= 0;
450         write_address_odd     <= 0;
451       end if;
452     end process;
453
454   -- control inputs to elimination processes and last division
```

```
455   control_input_to_elimination : process(r, output_backward_elim
          , output_last_division, output_forward_elim,
          data_out_brams_M_inv, data_out_brams_M)
456   begin
457     if(r.valid_data = '1') then
458       if(r.state_reg.state = STATE_FORWARD_ELIMINATION) then
459         -- In state forward elimination the reads and writes are
                issued from
460         -- top_forward_elimination, not from top-inverse
461         input_forward_elimination.state_reg                    <=
                r.state_reg;
462         input_forward_elimination.valid_data                   <=
                '1';
463         input_forward_elimination.flag_first_data_elimination <=
                r.flag_first_data_elimination;
464         -- Set input to last division
465         input_last_division.row_i                              <=
                ((others => (others => '0')));
466         input_last_division.inv_row_i                          <=
                ((others => (others => '0')));
467         input_last_division.state_reg.state                    <=
                STATE_IDLE;
468         input_last_division.index_i                            <=
                0;
469         input_last_division.valid_data                         <=
                '0';
470         input_last_division.flag_write_to_even_row             <=
                '0';
471         input_last_division.write_address_even                 <=
                0;
472         input_last_division.write_address_odd                  <=
                0;
473         for i in 0 to P_BANDS-1 loop
474           input_forward_elimination.row_even(i)     <= signed(
                  data_out_brams_M(i*PIXEL_DATA_WIDTH*2 +
                  PIXEL_DATA_WIDTH*2-1 downto i*PIXEL_DATA_WIDTH*2));
475           input_forward_elimination.row_odd(i)      <= signed(
                  data_out_brams_M(i*PIXEL_DATA_WIDTH*2 +
                  PIXEL_DATA_WIDTH*2 +EVEN_ROW_TOP_INDEX downto i*
                  PIXEL_DATA_WIDTH*2 +EVEN_ROW_TOP_INDEX+1));
476           input_forward_elimination.inv_row_even(i) <= signed(
                  data_out_brams_M_inv(i*PIXEL_DATA_WIDTH*2 +
                  PIXEL_DATA_WIDTH*2-1 downto i*PIXEL_DATA_WIDTH*2));
477           input_forward_elimination.inv_row_odd(i)  <= signed(
                  data_out_brams_M_inv(i*PIXEL_DATA_WIDTH*2 +
                  PIXEL_DATA_WIDTH*2 +EVEN_ROW_TOP_INDEX downto i*
                  PIXEL_DATA_WIDTH*2 +EVEN_ROW_TOP_INDEX+1));
478         end loop;
479
480         --if (output_forward_elim.
```

```
                forward_elimination_write_state = EVEN_j_WRITE or
                output_forward_elim.forward_elimination_write_state =
                 ODD_j_WRITE) and output_forward_elim.valid_data =
                '1' then
481             if not(output_forward_elim.
                    forward_elimination_write_state = SWAP_ROWS) and
                    output_forward_elim.valid_data = '1' then
482               -- USE the same elimination-core as backward
                      elimination
483               -- set inputs elimination core elimination
484               input_elimination.row_j                              <=
                      output_forward_elim.row_j;
485               input_elimination.row_i                              <=
                      output_forward_elim.row_i;
486               input_elimination.index_i                            <=
                      output_forward_elim.index_i;
487               input_elimination.index_j                            <=
                      output_forward_elim.index_j;
488               input_elimination.inv_row_j                          <=
                      output_forward_elim.inv_row_j;
489               input_elimination.inv_row_i                          <=
                      output_forward_elim.inv_row_i;
490               input_elimination.valid_data                         <=
                      output_forward_elim.valid_data;
491               input_elimination.state_reg                          <=
                      output_forward_elim.state_reg;
492               input_elimination.write_address_even                 <=
                      output_forward_elim.write_address_even;
493               input_elimination.write_address_odd                  <=
                      output_forward_elim.write_address_odd;
494               input_elimination.flag_write_to_even_row             <=
                      output_forward_elim.flag_write_to_even_row;
495               input_elimination.flag_write_to_odd_row              <=
                      output_forward_elim.flag_write_to_odd_row;
496               input_elimination.forward_elimination_write_state <=
                      output_forward_elim.forward_elimination_write_state
                      ;
497             else
498               -- set input to elimination core
499               input_elimination.row_j                              <= r
                      .row_j;
500               input_elimination.row_i                              <= r
                      .row_i;
501               input_elimination.index_i                            <= r
                      .index_i;
502               input_elimination.index_j                            <= r
                      .index_j;
503               input_elimination.inv_row_j                          <= r
                      .inv_row_j;
504               input_elimination.inv_row_i                          <= r
```

```
                .inv_row_i;
505             input_elimination.valid_data                        <=
                    '0';
506             input_elimination.state_reg.state                   <=
                    STATE_IDLE;
507             input_elimination.write_address_even                <= r
                    .write_address_even;
508             input_elimination.write_address_odd                 <= r
                    .write_address_odd;
509             input_elimination.flag_write_to_even_row            <=
                    '0';
510             input_elimination.flag_write_to_odd_row             <=
                    '0';
511             input_elimination.forward_elimination_write_state <=
                    output_forward_elim.forward_elimination_write_state
                    ;
512           end if;
513         elsif(r.state_reg.state = STATE_BACKWARD_ELIMINATION) then
514           -- set input to forward_elimination
515           input_forward_elimination.valid_data                    <=
                    '0';
516           input_forward_elimination.state_reg                     <=
                    r.state_reg;
517           input_forward_elimination.flag_first_data_elimination <=
                    '0';
518           for i in 0 to P_BANDS-1 loop
519             input_forward_elimination.row_even(i)     <= to_signed
                    (0, PIXEL_DATA_WIDTH*2);
520             input_forward_elimination.row_odd(i)      <= to_signed
                    (0, PIXEL_DATA_WIDTH*2);
521             input_forward_elimination.inv_row_even(i) <= to_signed
                    (0, PIXEL_DATA_WIDTH*2);
522             input_forward_elimination.inv_row_odd(i)  <= to_signed
                    (0, PIXEL_DATA_WIDTH*2);
523           end loop;
524           -- set input to elimination core
525           input_elimination.row_j                               <= r.
                  row_j;
526           input_elimination.row_i                               <= r.
                  row_i;
527           input_elimination.index_i                             <= r.
                  index_i;
528           input_elimination.index_j                             <= r.
                  index_j;
529           input_elimination.inv_row_j                           <= r.
                  inv_row_j;
530           input_elimination.inv_row_i                           <= r.
                  inv_row_i;
531           input_elimination.valid_data                          <= r.
                  valid_data;
```

```
532           input_elimination.state_reg                          <= r.
                  state_reg;
533           input_elimination.write_address_even                 <= r.
                  write_address_even;
534           input_elimination.write_address_odd                  <= r.
                  write_address_odd;
535           input_elimination.flag_write_to_even_row             <= r.
                  flag_write_to_even_row;
536           input_elimination.flag_write_to_odd_row              <= r.
                  flag_write_to_odd_row;
537           input_elimination.forward_elimination_write_state <=
                  output_forward_elim.forward_elimination_write_state;
538           -- Set input to last division
539           input_last_division.row_i                            <= ((
                  others => (others => '0')));
540           input_last_division.inv_row_i                        <= ((
                  others => (others => '0')));
541           input_last_division.state_reg.state                  <=
                  STATE_IDLE;
542           input_last_division.index_i                          <= 0;
543           input_last_division.valid_data                       <=
                  '0';
544           input_last_division.flag_write_to_even_row           <=
                  '0';
545           input_last_division.write_address_even               <= 0;
546           input_last_division.write_address_odd                <= 0;
547         elsif(r.state_reg.state = STATE_LAST_DIVISION) then
548           -- Set input to last division
549           input_last_division.row_i                                 <=
                    r.row_i;
550           input_last_division.inv_row_i                             <=
                    r.inv_row_i;
551           input_last_division.state_reg.state                       <=
                    r.state_reg.state;
552           input_last_division.index_i                               <=
                    r.index_i;
553           input_last_division.valid_data                            <=
                    r.valid_data;
554           input_last_division.flag_write_to_even_row                <=
                    r.flag_write_to_even_row;
555           input_last_division.write_address_even                    <=
                    r.write_address_even;
556           input_last_division.write_address_odd                     <=
                    r.write_address_odd;
557           -- set input to elimination core
558           input_elimination.row_j                                   <=
                    r.row_j;
559           input_elimination.row_i                                   <=
                    r.row_i;
560           input_elimination.index_i                                 <=
```

```
                r.index_i;
561             input_elimination.index_j                                   <=
                r.index_j;
562             input_elimination.inv_row_j                                 <=
                r.inv_row_j;
563             input_elimination.inv_row_i                                 <=
                r.inv_row_i;
564             input_elimination.valid_data                                <=
                '0';
565             input_elimination.state_reg                                 <=
                r.state_reg;
566             input_elimination.write_address_even                        <=
                r.write_address_even;
567             input_elimination.write_address_odd                         <=
                r.write_address_odd;
568             input_elimination.flag_write_to_even_row                    <=
                '0';
569             input_elimination.flag_write_to_odd_row                     <=
                '0';
570             input_elimination.forward_elimination_write_state           <=
                output_forward_elim.forward_elimination_write_state;
571           -- set input to forward_elimination
572             input_forward_elimination.valid_data                        <=
                '0';
573             input_forward_elimination.state_reg                         <=
                r.state_reg;
574             input_forward_elimination.flag_first_data_elimination       <=
                '0';
575             for i in 0 to P_BANDS-1 loop
576               input_forward_elimination.row_even(i)      <= to_signed
                  (0, PIXEL_DATA_WIDTH*2);
577               input_forward_elimination.row_odd(i)       <= to_signed
                  (0, PIXEL_DATA_WIDTH*2);
578               input_forward_elimination.inv_row_even(i) <= to_signed
                  (0, PIXEL_DATA_WIDTH*2);
579               input_forward_elimination.inv_row_odd(i)  <= to_signed
                  (0, PIXEL_DATA_WIDTH*2);
580             end loop;
581           else
582             input_last_division.row_i                                   <=
                ((others => (others => '0')));
583             input_last_division.inv_row_i                               <=
                ((others => (others => '0')));
584             input_last_division.state_reg.state                         <=
                STATE_IDLE;
585             input_last_division.index_i                                 <=
                0;
586             input_last_division.valid_data                              <=
                '0';
587             input_last_division.flag_write_to_even_row                  <=
```

```
588             '0';
589           input_last_division.write_address_even <=
                0;
590           input_last_division.write_address_odd <=
                0;
591           -- set input to elimination core
592           input_elimination.row_j <=
                r.row_j;
593           input_elimination.row_i <=
                r.row_i;
594           input_elimination.index_i <=
                r.index_i;
595           input_elimination.index_j <=
                r.index_j;
596           input_elimination.inv_row_j <=
                r.inv_row_j;
597           input_elimination.inv_row_i <=
                r.inv_row_i;
598           input_elimination.valid_data <=
                '0';
599           input_elimination.state_reg.state <=
                STATE_IDLE;
600           input_elimination.write_address_even <=
                r.write_address_even;
601           input_elimination.write_address_odd <=
                r.write_address_odd;
602           input_elimination.flag_write_to_even_row <=
                '0';
603           input_elimination.flag_write_to_odd_row <=
                '0';
604           input_elimination.forward_elimination_write_state <=
                output_forward_elim.forward_elimination_write_state;
605           -- set input to forward_elimination
606           input_forward_elimination.valid_data <=
                '0';
607           input_forward_elimination.state_reg <=
                r.state_reg;
608           input_forward_elimination.flag_first_data_elimination <=
                '0';
609           for i in 0 to P_BANDS-1 loop
610             input_forward_elimination.row_even(i)     <= to_signed
                  (0, PIXEL_DATA_WIDTH*2);
611             input_forward_elimination.row_odd(i)      <= to_signed
                  (0, PIXEL_DATA_WIDTH*2);
612             input_forward_elimination.inv_row_even(i) <= to_signed
                  (0, PIXEL_DATA_WIDTH*2);
613             input_forward_elimination.inv_row_odd(i)  <= to_signed
                  (0, PIXEL_DATA_WIDTH*2);
614           end loop;
          end if;
```

```
615     else
616       -- set input to forward_elimination
617       input_forward_elimination.valid_data                    <=
              '0';
618       input_forward_elimination.state_reg                     <= r
              .state_reg;
619       input_forward_elimination.flag_first_data_elimination <=
              '0';
620       for i in 0 to P_BANDS-1 loop
621         input_forward_elimination.row_even(i)      <= to_signed
                (0, PIXEL_DATA_WIDTH*2);
622         input_forward_elimination.row_odd(i)       <= to_signed
                (0, PIXEL_DATA_WIDTH*2);
623         input_forward_elimination.inv_row_even(i) <= to_signed
                (0, PIXEL_DATA_WIDTH*2);
624         input_forward_elimination.inv_row_odd(i)  <= to_signed
                (0, PIXEL_DATA_WIDTH*2);
625       end loop;
626       -- set input to elimination core
627       input_elimination.row_j                                 <= r.
              row_j;
628       input_elimination.row_i                                 <= r.
              row_i;
629       input_elimination.index_i                               <= r.
              index_i;
630       input_elimination.index_j                               <= r.
              index_j;
631       input_elimination.inv_row_j                             <= r.
              inv_row_j;
632       input_elimination.inv_row_i                             <= r.
              inv_row_i;
633       input_elimination.valid_data                            <= '0';
634       input_elimination.state_reg.state                       <=
              STATE_IDLE;
635       input_elimination.write_address_even                    <= r.
              write_address_even;
636       input_elimination.write_address_odd                     <= r.
              write_address_odd;
637       input_elimination.flag_write_to_even_row                <= '0';
638       input_elimination.flag_write_to_odd_row                 <= '0';
639       input_elimination.forward_elimination_write_state <=
              output_forward_elim.forward_elimination_write_state;
640       -- Input to last division
641       input_last_division.row_i                               <= ((
              others => (others => '0')));
642       input_last_division.inv_row_i                           <= ((
              others => (others => '0')));
643       input_last_division.state_reg.state                     <=
              STATE_IDLE;
644       input_last_division.index_i                             <= 0;
```

```
645         input_last_division.valid_data                    <= '0';
646         input_last_division.flag_write_to_even_row        <= '0';
647         input_last_division.write_address_even            <= 0;
648         input_last_division.write_address_odd             <= 0;
649       end if;
650     end process;
651
652     control_inverse_output : process(r, data_out_brams_M_inv)
653     begin
654       case r.state_reg.state is
655         when STATE_OUTPUT_INVERSE_MATRIX =>
656           inverse_rows.valid_data       <= '1';
657           inverse_rows.address          <= r.
                counter_output_inverse_matrix;
658           inverse_rows.two_inverse_rows <= data_out_brams_M_inv;
659         when others =>
660           inverse_rows.valid_data       <= '0';
661           inverse_rows.address          <= 0;
662           inverse_rows.two_inverse_rows <= (others => '0');
663       end case;
664     end process;
665
666
667     comb : process(reset_n, valid, r, data_out_brams_M_inv,
           data_out_brams_M, output_forward_elim, output_backward_elim
           , output_last_division, din, writes_done_on_column,
           write_address_odd, write_address_even)  -- combinatorial
           process
668       variable v : inverse_top_level_reg_type;
669     begin
670       v := r;
671       case v.state_reg.state is
672         when STATE_IDLE =>
673           v.read_enable             := '0';
674           v.flag_write_to_even_row  := '0';
675           v.flag_write_to_odd_row   := '0';
676           v.valid_data              := '0';
677           if(valid = '1') then
678             v.valid_data

                  := '1';
679             v.state_reg.state

                  := STATE_STORE_CORRELATION_MATRIX;
680                                       -- Set write address to
                                            BRAMS
681             v.write_address_even

                  := 0;
682             v.read_address_odd
```

```
                := 0;
683             v.read_address_even

                := 0;
684             v.flag_write_to_odd_row

                := '1';
685             v.flag_write_to_even_row

                := '1';
686             v.read_enable

                := '1';
687             v.bram_write_data_M

                := din;
688             v.writes_done_on_column

                := writes_done_on_column;
689             v.bram_write_data_M_inv

                := (others => '0');
690             v.bram_write_data_M_inv

                := (others => '0');
691             v.bram_write_data_M_inv((to_integer(unsigned(r.
                    writes_done_on_column))*2)*PIXEL_DATA_WIDTH*2)
                                                := '1';  -- creating the
                     identity matrix
692             v.bram_write_data_M_inv((to_integer(unsigned(
                    writes_done_on_column))*2+1)*PIXEL_DATA_WIDTH*2+
                   P_BANDS*PIXEL_DATA_WIDTH*2) := '1';
693             v.wait_counter

                := 0;
694             v.flag_waiting_for_bram_update

                := '0';
695           end if;
696         -- need to wait until valid data on all
697         when STATE_STORE_CORRELATION_MATRIX =>
698                                             -- SET BRAM to write
                                                   input data
699           v.writes_done_on_column

              := writes_done_on_column;
700           v.write_address_even

              := r.write_address_even +1;
```

```
701             v.write_address_odd

                    := r.write_address_odd +1;
702             v.read_address_odd

                    := 0;
703             v.read_address_even

                    := 0;
704             v.read_enable

                    := '1';
705             v.bram_write_data_M

                    := din;
706             v.bram_write_data_M_inv

                    := (others => '0');
707             v.bram_write_data_M_inv((to_integer(unsigned(
                    writes_done_on_column))*2)*PIXEL_DATA_WIDTH*2)
                                                := '1'; -- creating the
                     identity matrix
708             v.bram_write_data_M_inv((to_integer(unsigned(
                    writes_done_on_column))*2+1)*PIXEL_DATA_WIDTH*2+
                    P_BANDS*PIXEL_DATA_WIDTH*2) := '1';
709             v.flag_waiting_for_bram_update

                    := '0';
710             if (to_integer(unsigned(r.writes_done_on_column)) +1 <
                    P_BANDS/2) then
711               v.bram_write_data_M_inv((to_integer(unsigned(
                      writes_done_on_column))*2)*PIXEL_DATA_WIDTH*2)
                                                  := '1'; -- creating
                      the identity matrix
712               v.bram_write_data_M_inv((to_integer(unsigned(
                      writes_done_on_column))*2+1)*PIXEL_DATA_WIDTH*2+
                      P_BANDS*PIXEL_DATA_WIDTH*2) := '1';
713             end if;
714             if to_integer(unsigned(r.writes_done_on_column)) =
                    P_BANDS/2-1 then
715
```

```
716                                          -- in BRAM before
                                                 starting to edit it.
717             v.read_enable                          := '1';
718             v.read_address_even                    := 0;
719             v.read_address_odd                     := 0;
720             v.state_reg.state                      :=
                    STATE_FORWARD_ELIMINATION;
721             v.write_enable_even                    := '0';
722             v.write_enable_odd                     := '0';
723             v.wait_counter                         := 0;
724             v.flag_last_read_backward_elimination  := '0';
725             v.flag_first_data_elimination          := '1';
726             v.valid_data                           := '1';
727
728           end if;
729           if valid = '0' then
730             v.state_reg.state := STATE_IDLE;
731 --            v.state_reg.drive := STATE_IDLE_DRIVE;
732           end if;
733         when STATE_FORWARD_ELIMINATION =>
734           -- Set first memory_request?
735           -- Set write_state?
736           v.flag_first_data_elimination := '0';
737           if output_forward_elim.index_j = P_BANDS-1 and
                output_forward_elim.index_i = P_BANDS-2 then
738             -- finished forward elimination
739
740             v.state_reg.state                 :=
                    STATE_BACKWARD_ELIMINATION;
741             v.flag_first_iter_backward_elim := '1';
742             -- Request data for BACKWARD elimination
743             v.read_address_even               := P_BANDS/2-1;  --
                    read toppermost address, contains
744                                                --row P_BANDS-1 and
                                                     P_BANDS-2
745             v.read_address_odd                := P_BANDS/2-1;
```

```
746           end if;
747         when STATE_BACKWARD_ELIMINATION =>
748           if(r.flag_first_iter_backward_elim = '1') then
749             -- Read first data from BRAMs
750             v.write_address_even                                    :=
                    P_BANDS/2-1;  -- first write will happen
751                                           -- to even row, located
752                                           -- in even BRAMs.
753             v.write_address_odd                                     :=
                    P_BANDS/2-1;
754             v.read_enable                                           :=
                    '1';
755             v.write_enable_even                                     :=
                    '0';
756             v.write_enable_odd                                      :=
                    '0';
757             v.flag_first_data_elimination                           :=
                    '0';
758             --v.flag_waited_one_clk
                  := '0';
759             v.flag_first_memory_request                             :=
                    '1';
760             v.index_j_two_cycles_ahead                              :=
                    P_BANDS-2;
761             v.index_i_two_cycles_ahead                              :=
                    P_BANDS-1;
762             v.read_address_row_i_two_cycles_ahead                   :=
                    P_BANDS/2-1;
763             v.read_address_even                                     :=
                    P_BANDS/2-1;
764             v.read_address_odd                                      :=
                    P_BANDS/2-1;
765             v.address_row_i                                         :=
                    P_BANDS/2-1;
766             v.flag_finished_sending_data_to_BRAM_one_cycle_ago      :=
                    '0';
767             v.flag_finished_sending_data_to_BRAM_two_cycles_ago     :=
                    '0';
768             v.flag_wr_row_i_at_odd_row                              :=
                    '1';
769             v.flag_prev_row_i_at_odd_row                            :=
                    '1';
770             v.flag_first_iter_backward_elim                         :=
                    '0';
771             v.wait_counter                                          :=
                    0;
772             v.flag_waiting_for_bram_update                          :=
                    '0';
773             v.flag_last_read_backward_elimination                   :=
                    '0';
```

```
774           end if;
775
776           if(r.flag_first_memory_request = '1') then
777             v.flag_first_memory_request := '0';
778             --v.flag_waited_one_clk       := '1';
779             v.index_j_two_cycles_ahead  := r.
                    index_j_two_cycles_ahead -1;
780             v.read_address_even         := r.read_address_even -1;
781             -- need to read an odd row
782             v.read_address_odd          := r.read_address_odd -1;
783
784             v.flag_first_data_elimination := '1';
785
786           end if;
787           -- if(r.flag_waited_one_clk = '1') then
788           --   v.flag_first_data_elimination := '1'; -- the next
                  clock cycle the BRAM
789           --                                    -- will have the
                  correct output, for
790           --   -- the first input of the inverse matrix
791           --   v.flag_waited_one_clk :='0';
792           --   if(r.index_j_two_cycles_ahead-1 >= 0) then
793           --     -- need to read an even row, do not change read
                  address
794           --     v.index_j_two_cycles_ahead := r.
                  index_j_two_cycles_ahead-1;
795           --   end if;
796           -- end if;
797           if (r.flag_first_data_elimination = '1') then  --
                  received the first
798             --input_data to backward elimination from BRAM
799 --            v.state_reg.fsm_start_signal  :=
    START_BACKWARD_ELIMINATION;
800             -- must set the flag low again
801             v.flag_first_data_elimination := '0';
802             for i in 0 to P_BANDS-1 loop
803               v.row_j(i)      := signed(data_out_brams_M(i*
                      PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1 downto
                      i*PIXEL_DATA_WIDTH*2));
804               v.row_i(i)      := signed(data_out_brams_M(i*
                      PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 +
                      EVEN_ROW_TOP_INDEX downto i*PIXEL_DATA_WIDTH*2 +
                      EVEN_ROW_TOP_INDEX+1));
805               -- the odd row
806               v.inv_row_j(i) := signed(data_out_brams_M_inv(i*
                      PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1 downto
                      i*PIXEL_DATA_WIDTH*2));
807               v.inv_row_i(i) := signed(data_out_brams_M_inv(i*
                      PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 +
                      EVEN_ROW_TOP_INDEX downto i*PIXEL_DATA_WIDTH*2 +
```

```
                EVEN_ROW_TOP_INDEX+1));
808           end loop;
809           v.index_i                    := P_BANDS-1;
810           v.index_j                    := P_BANDS-2;
811           v.address_row_i              := P_BANDS/2-1;
812           v.valid_data                 := '1';
813           -- The first written j-row will alway be at an even-
                row.
814           v.flag_write_to_even_row     := '1';
815           v.flag_write_to_odd_row      := '0';
816           v.write_enable_even          := '1';
817           v.write_enable_odd           := '0';
818           v.write_address_even         := P_BANDS/2 -1;
819           v.write_address_odd          := P_BANDS/2-1;
820           v.flag_wr_row_i_at_odd_row   := '1';
821           v.elimination_write_state    := ODD_j_WRITE;
822           -- read new data
823           if(r.read_address_odd >= 0 and r.
                index_j_two_cycles_ahead >= 1) then
824             -- need to read an even row
825             v.read_address_odd          := r.read_address_odd;
826             v.read_address_even         := r.read_address_even;
827             v.index_j_two_cycles_ahead := r.
                  index_j_two_cycles_ahead-1;
828           elsif r.index_j_two_cycles_ahead < 1 then
829             -- new i or finished, update
830             if r.index_i_two_cycles_ahead >= 2 then
831               v.index_i_two_cycles_ahead := r.
                    index_i_two_cycles_ahead-1;
832               v.index_j_two_cycles_ahead := r.
                    index_i_two_cycles_ahead-1;
833               if r.flag_prev_row_i_at_odd_row = '1' then
834                 -- next row i will be located in an even indexed
                      row
835                 v.read_address_even                    := r.
                      read_address_row_i_two_cycles_ahead;
836                 v.read_address_odd                     := r.
                      read_address_row_i_two_cycles_ahead-1;
837                 v.read_address_row_i_two_cycles_ahead := r.
                      read_address_row_i_two_cycles_ahead;
838                 v.flag_prev_row_i_at_odd_row           := '0';
839               else
840                 -- next row_i will be located in an odd indexed
                      row
841                 v.read_address_odd                     := r.
                      read_address_row_i_two_cycles_ahead -1;
842                 v.read_address_even                    := r.
                      read_address_row_i_two_cycles_ahead -1;
843                 v.read_address_row_i_two_cycles_ahead := r.
                      read_address_row_i_two_cycles_ahead-1;
```

```
844                 v.flag_prev_row_i_at_odd_row           := '1';
845               end if;
846             end if;
847           end if;
848         end if;
849         case r.elimination_write_state is
850           when ODD_j_WRITE =>
851             if r.flag_waiting_for_bram_update = '0' then
852               v.flag_write_to_even_row := '0';
853               v.flag_write_to_odd_row  := '1';
854               -- row_j is outputted from odd BRAMs(located at
                      higher end of output)
855               for i in 0 to P_BANDS-1 loop
856                 v.row_j(i)      := signed(data_out_brams_M(i*
                        PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 +
                        EVEN_ROW_TOP_INDEX downto i*PIXEL_DATA_WIDTH
                        *2 +EVEN_ROW_TOP_INDEX+1));
857                 v.inv_row_j(i) := signed(data_out_brams_M_inv(i*
                        PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 +
                        EVEN_ROW_TOP_INDEX downto i*PIXEL_DATA_WIDTH
                        *2 +EVEN_ROW_TOP_INDEX+1));
858               end loop;
859               v.write_enable_even := '0';
860               v.write_enable_odd  := '1';
861               v.index_j           := r.index_j -1;
862               if r.index_j >= 1 then
863                 v.write_address_odd  := r.write_address_odd -1;
864                 v.write_address_even := r.write_address_even -1;
865               end if;
866             end if;
867             -- do not really understand how the -2 got in this
                    if... check
868             if v.index_j <= 1 and r.index_i_two_cycles_ahead-2 -
                      v.index_j < B_RAM_WAIT_CLK_CYCLES and r.
                    wait_counter < B_RAM_WAIT_CLK_CYCLES-(r.
                    index_i_two_cycles_ahead-2 -v.index_j) then
869               -- Need to wait for the row to update before
                      reading it.
870               v.wait_counter                  := r.wait_counter
                      +1;
871               v.flag_waiting_for_bram_update := '1';
872             else
873               v.flag_waiting_for_bram_update := '0';
874               v.wait_counter                  := 0;
875               if(v.index_j >= 1) then
876                 --v.index_j                  := r.index_j -1;
877                 v.elimination_write_state := EVEN_j_WRITE;
878               elsif v.index_j < 1 then
879                 --v.index_i := r.index_i -1;
880                 --v.index_j := r.index_i -2;
```

```
881                 if(r.flag_wr_row_i_at_odd_row = '0') then
882                   v.elimination_write_state := ODD_i_START;
883                 else
884                   v.elimination_write_state := EVEN_i_START;
885                 end if;
886               end if;
887               -- read new data. Data need to be read two clock
                     cycles in advance
888               if(r.read_address_odd >= 1 and r.
                     index_j_two_cycles_ahead >= 2) then
889                 -- need to read an odd row
890                 v.read_address_odd          := r.read_address_odd
                       -1;
891                 v.read_address_even         := r.
                       read_address_even-1;
892                 v.index_j_two_cycles_ahead := r.
                       index_j_two_cycles_ahead-1;
893               elsif v.index_j < 2 then
894                 -- new i, update
895                 if r.index_i_two_cycles_ahead >= 2 then
896                   v.index_i_two_cycles_ahead := r.
                         index_i_two_cycles_ahead-1;
897                   v.index_j_two_cycles_ahead := r.
                         index_i_two_cycles_ahead-2;
898                   if r.flag_prev_row_i_at_odd_row = '1' then
899                     -- next row i will be located in an even
                           indexed row
900                     v.read_address_even                    := r.
                           address_row_i;
901                     v.read_address_odd                     := r.
                           address_row_i-1;
902                     v.read_address_row_i_two_cycles_ahead := r.
                           read_address_row_i_two_cycles_ahead;
903                     v.flag_prev_row_i_at_odd_row           :=
                           '0';
904                   else
905                     -- next row_i will be located in an odd
                           indexed row
906                     v.read_address_odd                     := r.
                           read_address_row_i_two_cycles_ahead -1;
907                     v.read_address_even                    := r.
                           read_address_row_i_two_cycles_ahead -1;
908                     v.read_address_row_i_two_cycles_ahead := r.
                           read_address_row_i_two_cycles_ahead-1;
909                     v.flag_prev_row_i_at_odd_row           :=
                           '1';
910                   end if;
911                 end if;
912               end if;
913             end if;
```

```
914             when EVEN_j_WRITE =>
915               if r.flag_waiting_for_bram_update = '0' then
916                 for i in 0 to P_BANDS-1 loop
917                   -- data is located in the even part of the
                          output from BRAM
918                   v.row_j(i)      := signed(data_out_brams_M(i*
                          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1
                          downto i*PIXEL_DATA_WIDTH*2));
919                   v.inv_row_j(i) := signed(data_out_brams_M_inv(i*
                          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1
                          downto i*PIXEL_DATA_WIDTH*2));
920                 end loop;
921                 v.flag_write_to_even_row := '1';
922                 v.flag_write_to_odd_row  := '0';
923                 v.write_address_even     := r.write_address_even;
924                 v.write_address_odd      := r.write_address_odd;
925                 v.write_enable_even      := '1';
926                 v.write_enable_odd       := '0';
927                 v.index_j                := r.index_j -1;
928                 v.index_i                := r.index_i;
929               end if;
930
931               if v.index_j <= 1 and r.index_j_two_cycles_ahead-1-
                      v.index_j < B_RAM_WAIT_CLK_CYCLES and r.
                      wait_counter < B_RAM_WAIT_CLK_CYCLES-(r.
                      index_j_two_cycles_ahead-1 -v.index_j) then
932                 v.wait_counter                := r.wait_counter
                        +1;
933                 v.flag_waiting_for_bram_update := '1';
934               else
935                 v.wait_counter                := 0;
936                 v.flag_waiting_for_bram_update := '0';
937                 if(v.index_j >= 2) then
938                   v.elimination_write_state := ODD_j_WRITE;
939                 elsif v.index_j < 2 then
940                   if(r.flag_wr_row_i_at_odd_row = '0') then
941                     v.elimination_write_state := ODD_i_START;
942                   else
943                     v.elimination_write_state := EVEN_i_START;
944                   end if;
945                 end if;
946                 -- read new data
947                 --if(r.read_address_odd >= 1 and v.index_j >= 1)
                        then
948                 if r.flag_last_read_backward_elimination = '0'
                        then
949                   if(r.read_address_odd >= 0 and r.
                          index_j_two_cycles_ahead >= 1) then
950                     -- need to read an even row("two clock cycles
                            ahead")
```

```
951                     -- Even and odd read addresses will be equal
                            in backward
952                     -- elimination except for when reading the
                            first read for a even
953                     -- indexed i.
954                     v.read_address_odd          := r.
                            read_address_odd;
955                     --v.read_address_even         := r.
                            read_address_even;
956                     v.read_address_even         := r.
                            read_address_odd;
957                     v.index_j_two_cycles_ahead := r.
                            index_j_two_cycles_ahead-1;
958                   elsif v.index_j < 2 then
959                     -- new i or finished all necessary reads,
                            update
960                     if v.index_i >= 2 then
961                       v.index_i_two_cycles_ahead := r.
                              index_i_two_cycles_ahead-1;
962                       v.index_j_two_cycles_ahead := r.
                              index_i_two_cycles_ahead-2;
963                       if r.flag_prev_row_i_at_odd_row = '1' then
964                         -- next row i will be located in an even
                                indexed row
965                         v.read_address_even                    := r
                                .read_address_row_i_two_cycles_ahead;
966                         v.read_address_odd                     := r
                                .read_address_row_i_two_cycles_ahead-1;
967                         v.read_address_row_i_two_cycles_ahead := r
                                .read_address_row_i_two_cycles_ahead;
968                         v.flag_prev_row_i_at_odd_row           :=
                                '0';
969                       else
970                         -- next row_i will be located in an odd
                                indexed row
971                         v.read_address_odd                     := r
                                .read_address_row_i_two_cycles_ahead
                                -1;
972                         v.read_address_even                    := r
                                .read_address_row_i_two_cycles_ahead
                                -1;
973                         v.read_address_row_i_two_cycles_ahead := r
                                .read_address_row_i_two_cycles_ahead-1;
974                         v.flag_prev_row_i_at_odd_row           :=
                                '1';
975                       end if;
976                     end if;
977                   end if;
978                 end if;
979               end if;
```

```
980             when ODD_i_START =>
981               if(r.
                      flag_finished_sending_data_to_BRAM_one_cycle_ago
                      = '0') then
982                 for i in 0 to P_BANDS-1 loop
983                   v.row_j(i)      := signed(data_out_brams_M(i*
                          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1
                          downto i*PIXEL_DATA_WIDTH*2));
984                   v.row_i(i)      := signed(data_out_brams_M(i*
                          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 +
                          EVEN_ROW_TOP_INDEX downto i*PIXEL_DATA_WIDTH
                          *2 +EVEN_ROW_TOP_INDEX+1));
985                   -- the odd row
986                   v.inv_row_j(i) := signed(data_out_brams_M_inv(i*
                          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1
                          downto i*PIXEL_DATA_WIDTH*2));
987                   v.inv_row_i(i) := signed(data_out_brams_M_inv(i*
                          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 +
                          EVEN_ROW_TOP_INDEX downto i*PIXEL_DATA_WIDTH
                          *2 +EVEN_ROW_TOP_INDEX+1));
988                 end loop;
989                 v.flag_write_to_even_row    := '1';
990                 v.flag_wr_row_i_at_odd_row := '1';
991                 v.flag_write_to_odd_row     := '0';
992                 v.index_i                   := r.index_i-1;
993                 v.index_j                   := r.index_i-2;
994                 v.address_row_i             := r.address_row_i-1;
995                 v.write_address_even        := r.address_row_i-1;
996                 v.write_address_odd         := r.address_row_i-1;
997                 v.write_enable_even         := '1';
998                 v.write_enable_odd          := '0';
999                 if(v.index_j > 1) then    -- the first two indexes
                        are contained
1000                  -- within address 0
1001                  --v.index_j                  := r.index_j -1;
1002                  v.elimination_write_state := ODD_j_WRITE;
1003                elsif v.index_j = 0 and v.index_i = 1 then
1004                  -- In two clock cycles the data will be written
                        to B_RAM.
1005                  -- and it is possible to change state to
                        TOP_LAST_DIVISON.
1006                  v.
                        flag_finished_sending_data_to_BRAM_one_cycle_ago
                         := '1';
1007                end if;
1008
1009                if r.flag_last_read_backward_elimination = '0'
                       then
1010                  if(r.read_address_odd >= 0 and r.
                        index_j_two_cycles_ahead >= 1) then
```

```
1011                    --if(r.read_address_odd >= 1 and r.
                            index_j_two_cycles_ahead >= 2) then
1012                    -- need to read an even row
1013                    v.read_address_odd          := r.
                            read_address_odd;
1014                    v.read_address_even         := r.
                            read_address_even;
1015                    v.index_j_two_cycles_ahead := r.
                            index_j_two_cycles_ahead-1;
1016                  elsif r.index_j_two_cycles_ahead < 1 then
1017                    -- new i or finished, update
1018                    if r.index_i_two_cycles_ahead >= 2 then
1019                      v.index_i_two_cycles_ahead := r.
                              index_i_two_cycles_ahead-1;
1020                      v.index_j_two_cycles_ahead := v.
                              index_i_two_cycles_ahead-2;
1021                      if r.flag_prev_row_i_at_odd_row = '1' then
1022                        -- next row i will be located in an even
                              indexed row
1023                        v.read_address_even                    := r
                              .read_address_row_i_two_cycles_ahead;
1024                        v.read_address_odd                     := r
                              .read_address_row_i_two_cycles_ahead-1;
1025                        v.read_address_row_i_two_cycles_ahead := r
                              .read_address_row_i_two_cycles_ahead;
1026                        v.flag_prev_row_i_at_odd_row           :=
                              '0';
1027                      else
1028                        -- next row_i will be located in an odd
                              indexed row
1029                        v.read_address_odd                     := r
                              .read_address_row_i_two_cycles_ahead
                              -1;
1030                        v.read_address_even                    := r
                              .read_address_row_i_two_cycles_ahead
                              -1;
1031                        v.read_address_row_i_two_cycles_ahead := r
                              .read_address_row_i_two_cycles_ahead-1;
1032                        v.flag_prev_row_i_at_odd_row           :=
                              '1';
1033                      end if;
1034                    end if;
1035                  end if;
1036                end if;
1037              else
1038                v.read_address_even
                                                       := 0;
1039                v.read_address_odd
                                                        := 0;
1040                v.
```

```
                    flag_finished_sending_data_to_BRAM_two_cycles_ago
                     := '1';
1041            end if;
1042
1043            if(r.
                    flag_finished_sending_data_to_BRAM_two_cycles_ago
                     = '1') then
1044              v.read_address_even
                                                    := 0;
1045              v.read_address_odd
                                                     := 0;
1046              v.
                    flag_finished_sending_data_to_BRAM_three_cycles_ago
                     := '1';
1047            end if;
1048            if r.
                    flag_finished_sending_data_to_BRAM_three_cycles_ago
                     = '1' then
1049              v.state_reg.state
                                                      :=
                    STATE_LAST_DIVISION;
1050              v.last_division_write_state
                                          := EVEN_i_WRITE;
1051              v.valid_data
                                                           :=
                    '0';
1052              v.index_i_two_cycles_ahead
                                           := 2;
1053              v.flag_first_memory_request
                                          := '1'; -- used to
                    indicate that the
1054                                     -- next cycle the first
                                            write
1055                                     -- will happen from
                                            STATE_LAST_DIVISION
1056              v.read_address_even
                                                    := 0;
1057              v.read_address_odd
                                                     := 0;
1058              v.flag_finished_sending_data_to_BRAM_one_cycle_ago
                        := '0';
1059              v.
                    flag_finished_sending_data_to_BRAM_two_cycles_ago
                       := '0';
1060              v.
                    flag_finished_sending_data_to_BRAM_three_cycles_ago
                     := '0';
1061            end if;
1062
1063          when EVEN_i_START =>
```

```
1064                for i in 0 to P_BANDS-1 loop
1065                  v.row_i(i)      := signed(data_out_brams_M(i*
                          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1
                          downto i*PIXEL_DATA_WIDTH*2));
1066                  v.row_j(i)      := signed(data_out_brams_M(i*
                          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 +
                          EVEN_ROW_TOP_INDEX downto i*PIXEL_DATA_WIDTH*2
                          +EVEN_ROW_TOP_INDEX+1));
1067                  -- the odd row
1068                  v.inv_row_i(i) := signed(data_out_brams_M_inv(i*
                          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1
                          downto i*PIXEL_DATA_WIDTH*2));
1069                  v.inv_row_j(i) := signed(data_out_brams_M_inv(i*
                          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 +
                          EVEN_ROW_TOP_INDEX downto i*PIXEL_DATA_WIDTH*2
                          +EVEN_ROW_TOP_INDEX+1));
1070                end loop;
1071                v.flag_wr_row_i_at_odd_row := '0';
1072                v.flag_write_to_even_row   := '0';
1073                v.flag_write_to_odd_row    := '1';
1074                v.index_i                  := r.index_i-1;
1075                v.index_j                  := r.index_i-2;
1076                v.address_row_i            := r.address_row_i;
1077                v.write_address_even       := r.address_row_i -1;
1078                v.write_address_odd        := r.address_row_i -1;
1079                v.write_enable_even        := '0';
1080                v.write_enable_odd         := '1';
1081                if(v.index_j >= 1) then
1082                  v.elimination_write_state := EVEN_j_WRITE;
1083                end if;
1084                -- read new data.
1085                if r.flag_last_read_backward_elimination = '0' then
1086                  if(r.read_address_odd >= 1 and r.
                          index_j_two_cycles_ahead >= 2) then
1087                    -- need to read an odd row
1088                    v.read_address_odd          := r.read_address_odd
                            -1;
1089                    v.read_address_even         := r.
                            read_address_even-1;
1090                    v.index_j_two_cycles_ahead := r.
                            index_j_two_cycles_ahead-1;
1091                  elsif r.index_j_two_cycles_ahead < 1 then
1092                    -- new i, update
1093                    if r.index_i_two_cycles_ahead >= 2 then
1094                      v.index_i_two_cycles_ahead := r.
                              index_i_two_cycles_ahead-1;
1095                      v.index_j_two_cycles_ahead := r.
                              index_i_two_cycles_ahead-2;
1096                      if r.flag_prev_row_i_at_odd_row = '1' then
1097                        -- next row i will be located in an even
```

```
                        indexed row
1098                    v.read_address_even                    := r.
                            read_address_row_i_two_cycles_ahead;
1099                    v.read_address_odd                     := r.
                            read_address_row_i_two_cycles_ahead-1;
1100                    v.read_address_row_i_two_cycles_ahead := r.
                            read_address_row_i_two_cycles_ahead;
1101                    v.flag_prev_row_i_at_odd_row           :=
                            '0';
1102                  else
1103                    -- next row_i will be located in an odd
                            indexed row
1104                    v.read_address_odd                     := r.
                            read_address_row_i_two_cycles_ahead -1;
1105                    v.read_address_even                    := r.
                            read_address_row_i_two_cycles_ahead -1;
1106                    v.read_address_row_i_two_cycles_ahead := r.
                            read_address_row_i_two_cycles_ahead-1;
1107                    v.flag_prev_row_i_at_odd_row           :=
                            '1';
1108                  end if;
1109                end if;
1110                if v.index_i_two_cycles_ahead = 1 then
1111                  -- Finished reading data for backward
                        elimination
1112                  v.flag_last_read_backward_elimination := '1';
1113                end if;
1114              end if;
1115            end if;
1116          when others =>
1117        end case;
1118      when STATE_LAST_DIVISION =>
1119        case r.last_division_write_state is
1120          when EVEN_i_WRITE =>
1121            if (r.flag_first_memory_request = '1') then
1122              -- First write is to a even row
1123              v.index_i                    := 0;
1124              v.flag_first_memory_request := '0';
1125              v.write_address_even         := 0;
1126              v.write_address_odd          := 0;
1127              v.write_address_even         := 0;
1128              v.write_address_odd          := 0;
1129              v.valid_data                 := '1';
1130            else
1131              v.index_i            := r.index_i+1;
1132              v.write_address_even := write_address_even +1;
1133              v.write_address_odd  := write_address_odd+1;
1134            end if;
1135            v.flag_write_to_even_row := '1';
1136            v.flag_write_to_odd_row  := '0';
```

```
1137                if r.read_address_even < P_BANDS/2-1 then
1138                  v.read_address_even := r.read_address_even+1;
1139                  v.read_address_odd  := r.read_address_even+1;
1140                end if;
1141                for i in 0 to P_BANDS-1 loop
1142                  -- data is located in the even part of the output
                          from BRAM
1143                  v.row_i(i)     := signed(data_out_brams_M(i*
                          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1
                          downto i*PIXEL_DATA_WIDTH*2));
1144                  v.inv_row_i(i) := signed(data_out_brams_M_inv(i*
                          PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1
                          downto i*PIXEL_DATA_WIDTH*2));
1145                end loop;
1146                if v.index_i <= P_BANDS-2 then
1147                  v.last_division_write_state := ODD_i_WRITE;
1148                end if;
1149              when ODD_i_WRITE =>
1150                v.index_i := r.index_i +1;
1151                if v.index_i >= P_BANDS-1 then
1152                  -- top_last_division is finished written
1153                  v.state_reg.state                :=
                          STATE_OUTPUT_INVERSE_MATRIX;
1154                  v.read_address_odd               := 0;
1155                  v.read_address_even              := 0;
1156                  v.read_enable                    := '1';
1157                  v.counter_output_inverse_matrix := 0;
1158                else
1159                  -- row_i is outputted from odd BRAMs(located at
                          higher end of output)
1160                  for i in 0 to P_BANDS-1 loop
1161                    v.row_i(i)     := signed(data_out_brams_M(i*
                            PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 +
                            EVEN_ROW_TOP_INDEX downto i*PIXEL_DATA_WIDTH
                            *2 +EVEN_ROW_TOP_INDEX+1));
1162                    v.inv_row_i(i) := signed(data_out_brams_M_inv(i*
                            PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 +
                            EVEN_ROW_TOP_INDEX downto i*PIXEL_DATA_WIDTH
                            *2 +EVEN_ROW_TOP_INDEX+1));
1163                  end loop;
1164                  v.read_address_even       := r.read_address_even;
1165                  v.read_address_odd        := r.read_address_odd;
1166                  v.flag_write_to_even_row := '0';
1167                  v.flag_write_to_odd_row  := '1';
1168                end if;
1169                if v.index_i <= P_BANDS-3 then
1170                  v.last_division_write_state := EVEN_i_WRITE;
1171                end if;
1172              when others =>
1173                v.read_address_even       := 0;
```

```
1174              v.read_address_odd        := 0;
1175              v.flag_write_to_even_row  := '0';
1176              v.flag_write_to_odd_row   := '0';
1177            end case;
1178          when STATE_OUTPUT_INVERSE_MATRIX =>
1179            -- Read all BRAMs to output data
1180            -- Already read the first two addresses?
1181            if r.counter_output_inverse_matrix < P_BANDS/2-1 then
1182              v.read_address_even             := r.read_address_even
                      +1;
1183              v.read_address_odd              := r.read_address_odd
                      +1;
1184              v.counter_output_inverse_matrix := r.
                      counter_output_inverse_matrix+1;
1185              v.valid_data                    := '1';
1186            else
1187            -- Finished! Signal, then go to STATE_IDLE
1188            end if;
1189
1190          when others =>
1191            v.read_enable                := '0';
1192            v.write_enable_even          := '0';
1193            v.write_enable_odd           := '0';
1194            v.elimination_write_state    := STATE_IDLE;
1195            v.state_reg.state            := STATE_IDLE;
1196            v.last_division_write_state  := STATE_IDLE;
1197            v.valid_data                 := '0';
1198
1199      end case;
1200      if(reset_n = '0') then
1201        v.read_enable                := '0';
1202        v.write_enable_even          := '0';
1203        v.write_enable_odd           := '0';
1204        v.elimination_write_state    := STATE_IDLE;
1205        v.state_reg.state            := STATE_IDLE;
1206        v.last_division_write_state  := STATE_IDLE;
1207        v.valid_data                 := '0';
1208      end if;
1209      r_in <= v;
1210    end process;
1211
1212
1213    regs : process(clk, reset_n, clk_en)
1214    begin
1215      if rising_edge(clk) and clk_en = '1' then
1216        if(reset_n = '0') then
1217        else
1218          r <= r_in;
1219        end if;
1220      end if;
```

```
1221
1222   end process;
1223
1224 end Behavioral;
```

C.7 Shiftregister

Listing C.7: Shiftregister

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.Common_types_and_functions.all;
7
8  -- A serial(n bit at a time) in parallell out shift register
9  -- Inputes four bands at a time, until a whole pixel is shifted
       in
10 entity shiftregister_four_pixels is
11   port (din           : in    std_logic_vector (PIXEL_DATA_WIDTH
         *ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA-1 downto 0);
12         valid         : in    std_logic;
13         clk           : in    std_logic;
14         clk_en        : in    std_logic;
15         reset_n       : in    std_logic;
16         shift_counter : out   std_logic_vector(log2(P_BANDS*
               PIXEL_DATA_WIDTH/(PIXEL_DATA_WIDTH*4)) downto 0);
17 -- Assuming PIXEL_DATA_WIDTH*4 (maximum 64) bit per input cycle,
       4 pixel components at max 16 bit. Important to
18 -- know delay of first input pixel in clock cycles
19         valid_out     : out   std_logic;
20         dout          : inout std_logic_vector(P_BANDS*
               PIXEL_DATA_WIDTH -1 downto 0)
21         );
22 end shiftregister_four_pixels;
23
24 architecture Behavioral of shiftregister_four_pixels is
25   signal r, r_in                                : std_logic_vector(
         P_BANDS*PIXEL_DATA_WIDTH -1 downto 0);
26   signal r_shift_counter_in, r_shift_counter : std_logic_vector(
         log2(P_BANDS*PIXEL_DATA_WIDTH/(PIXEL_DATA_WIDTH*
         ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA)) downto 0) := (others =>
          '0');
27   signal r_in_valid_out                         : std_logic;
28
29 begin
30   comb_proc : process(din, valid, dout, r_shift_counter)
31     variable v_shift_counter      : integer   := to_integer(
           unsigned(r_shift_counter));
```

```
32      variable v_temp_shift_data_in : std_logic_vector(P_BANDS*
            PIXEL_DATA_WIDTH-1 -ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA*
            PIXEL_DATA_WIDTH downto 0);
33      variable v                    : std_logic_vector(P_BANDS *
            PIXEL_DATA_WIDTH -1 downto 0);
34      variable v_valid_out          : std_logic := '0';
35    begin
36      if(valid = '1')then
37        v_shift_counter

              := to_integer(unsigned(r_shift_counter)) + 1;
38        v

              := dout;
39        v_temp_shift_data_in

              := v(P_BANDS*PIXEL_DATA_WIDTH-1 downto
              ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA*PIXEL_DATA_WIDTH);
40        v(P_BANDS*PIXEL_DATA_WIDTH-1 -
              ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA*PIXEL_DATA_WIDTH
              downto 0)                       :=
              v_temp_shift_data_in;
41        v(P_BANDS*PIXEL_DATA_WIDTH-1 downto P_BANDS*
              PIXEL_DATA_WIDTH - ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA*
              PIXEL_DATA_WIDTH) := din;
42        if v_shift_counter = P_BANDS/
              ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA then
43          v_valid_out := '1';
44        else
45          v_valid_out := '0';
46        end if;
47      else
48        v_shift_counter := 0;
49        v               := (others => '0');
50        v_valid_out     := '0';
51      end if;
52      if(reset_n = '0') then
53        v               := (others => '0');
54        v_shift_counter := 0;
55        v_valid_out     := '0';
56      end if;
57      r_shift_counter_in <= std_logic_vector(to_unsigned(
            v_shift_counter, r_shift_counter_in'length));
58      shift_counter      <= r_shift_counter;
59      r_in_valid_out     <= v_valid_out;
60      r_in               <= v;
61      dout               <= r;
62    end process;
63
64
```

```
65    sequential_proc : process(clk, clk_en)
66    begin
67      if (rising_edge(clk) and clk_en = '1') then
68        r               <= r_in;
69        valid_out       <= r_in_valid_out;
70        r_shift_counter <= r_shift_counter_in;
71      end if;
72    end process;
73
74
75  end Behavioral;
```

C.8 Forward elimination

Listing C.8: Forward elimination

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   library work;
6   use work.Common_types_and_functions.all;
7
8   -- This module controls the forward elimination stage. It issues
        reads and
9   -- writes to BRAM
10  entity top_forward_elimination is
11    port(clk                          : in  std_logic;
12         reset_n                      : in  std_logic;
13         clk_en                       : in  std_logic;
14         input_forward_elimination    : in
               input_elimination_reg_type;
15         output_forward_elimination : out
               output_forward_elimination_reg_type
16         );
17  end top_forward_elimination;
18
19  architecture Behavioral of top_forward_elimination is
20
21    signal r, r_in           : input_elimination_reg_type;
22    signal output_swap_rows : output_forward_elimination_reg_type;
23    signal input_swap_rows   : input_elimination_reg_type;
24    signal output_top_level : output_forward_elimination_reg_type;
25
26  begin
27    -- Instance to swap the rows if needed.
28    swap_rows_1 : entity work.swap_rows_module
29      port map (
30        clk              => clk,
```

```
31        reset_n           => reset_n,
32        clk_en            => clk_en,
33        input_swap_rows   => input_swap_rows,
34        output_swap_rows  => output_swap_rows);
35
36    --process to set output to inverse top level
37    set_outputs : process(r, output_swap_rows, output_top_level)
38    begin
39      case r.forward_elimination_write_state is
40        when STATE_IDLE =>
41          output_forward_elimination <= output_top_level;
42        when CHECK_DIAGONAL_ELEMENT_IS_ZERO =>
43          output_forward_elimination <= output_top_level;
44        when SWAP_ROWS =>
45          output_forward_elimination <= output_swap_rows;
46        when EVEN_j_WRITE =>
47          output_forward_elimination <= output_top_level;
48        when ODD_j_WRITE =>
49          output_forward_elimination <= output_top_level;
50        when others =>
51          output_forward_elimination <= output_top_level;
52      end case;
53    end process;
54
55    set_inputs_to_swap_rows : process(input_forward_elimination, r
          )
56    begin
57      case r.forward_elimination_write_state is
58        when STATE_IDLE =>
59          input_swap_rows.forward_elimination_write_state <=
                STATE_IDLE;
60          input_swap_rows.row_i                                <= ((
                others => (others => '0')));
61          input_swap_rows.row_j                                <= ((
                others => (others => '0')));
62          input_swap_rows.index_i                              <= 0;
63          input_swap_rows.index_j                              <= 0;
64          input_swap_rows.address_row_i                        <= 0;
65          input_swap_rows.address_row_j                        <= 0;
66          input_swap_rows.flag_write_to_even_row               <= '0';
67          input_swap_rows.flag_write_to_odd_row                <= '0';
68          input_swap_rows.flag_prev_row_i_at_odd_row           <= '0';
69        when SWAP_ROWS =>
70          input_swap_rows.forward_elimination_write_state <= r.
                forward_elimination_write_state;
71          if r.flag_start_swapping_rows = '1' then
72            -- input from top level
73            input_swap_rows.row_i                          <= r.row_i;
74            input_swap_rows.row_j                          <= r.row_j;
75            input_swap_rows.index_i                        <= r.
```

```
                index_i;
76              input_swap_rows.index_j                     <= r.
                    index_j;
77              input_swap_rows.address_row_i               <= r.
                    address_row_i;
78              input_swap_rows.address_row_j               <= r.
                    address_row_j;
79              input_swap_rows.flag_write_to_even_row      <= r.
                    flag_write_to_even_row;
80              input_swap_rows.flag_write_to_odd_row       <= r.
                    flag_write_to_odd_row;
81              input_swap_rows.flag_prev_row_i_at_odd_row <= r.
                    flag_prev_row_i_at_odd_row;
82            else
83              -- receive row i and row j from BRAM directly
84              -- Not been simulated and tested that this works.
85              input_swap_rows.row_i                       <=
                    input_forward_elimination.row_i;
86              input_swap_rows.row_j                       <=
                    input_forward_elimination.row_j;
87              input_swap_rows.index_i                     <=
                    input_forward_elimination.index_i;
88              input_swap_rows.index_j                     <=
                    input_forward_elimination.index_j;
89              input_swap_rows.address_row_i               <=
                    input_forward_elimination.address_row_i;
90              input_swap_rows.address_row_j               <=
                    input_forward_elimination.address_row_j;
91              input_swap_rows.flag_write_to_even_row      <=
                    input_forward_elimination.flag_write_to_even_row;
92              input_swap_rows.flag_write_to_odd_row       <=
                    input_forward_elimination.flag_write_to_odd_row;
93              input_swap_rows.flag_prev_row_i_at_odd_row <=
                    input_forward_elimination.
                    flag_prev_row_i_at_odd_row;
94            end if;
95          when others =>
96            input_swap_rows.forward_elimination_write_state <=
                  STATE_IDLE;
97            input_swap_rows.row_i                             <= ((
                  others => (others => '0')));
98            input_swap_rows.row_j                             <= ((
                  others => (others => '0')));
99            input_swap_rows.index_i                           <= 0;
100           input_swap_rows.index_j                           <= 0;
101           input_swap_rows.address_row_i                     <= 0;
102           input_swap_rows.address_row_j                     <= 0;
103           input_swap_rows.flag_write_to_even_row            <= '0';
104           input_swap_rows.flag_write_to_odd_row             <= '0';
105           input_swap_rows.flag_prev_row_i_at_odd_row        <= '0';
```

```
106       end case;
107     end process;
108
109
110
111     comb_process : process(output_swap_rows,
            input_forward_elimination, r, reset_n)
112
113       variable v : input_elimination_reg_type;
114
115     begin
116       v           := r;
117       v.state_reg := input_forward_elimination.state_reg;
118       if(input_forward_elimination.state_reg.state =
              STATE_FORWARD_ELIMINATION and input_forward_elimination.
              valid_data = '1') then
119         case r.forward_elimination_write_state is
120           when STATE_IDLE =>
121             v.valid_data := '0';
122             -- input_elimination.flag_first_data_elimination is to
                    be sent only
123             -- once, by the top level inverse
124             if input_forward_elimination.
                    flag_first_data_elimination = '1' then
125               v.forward_elimination_write_state :=
                      CHECK_DIAGONAL_ELEMENT_IS_ZERO;
126               v.flag_first_data_elimination     := '1';
127             end if;
128           when CHECK_DIAGONAL_ELEMENT_IS_ZERO =>
129             if r.flag_first_data_elimination = '1' then
130               -- First iteration of the forward-elimination
131               -- for the current processed pixel
132               v.index_i                                := 0;
133               v.index_j                                := 0;
134               -- Has already read the first j
135               v.index_j_two_cycles_ahead               := 2;
136               v.index_i_two_cycles_ahead               := 0;
137               v.read_address_row_i_two_cycles_ahead := 0;
138               v.flag_write_to_even_row                 := '0';
139               v.flag_write_to_odd_row                  := '1';
140               v.write_address_even                     := 0;
141               v.write_address_odd                      := 0;
142               v.valid_data                             := '0';
143               v.flag_first_data_elimination            := '0';
144               -- First iteration row_i is located at even index=0
145               v.row_i                                  :=
                      input_forward_elimination.row_even;
146               v.row_j                                  :=
                      input_forward_elimination.row_odd;
147               v.inv_row_i                              :=
```

```
                input_forward_elimination.inv_row_even;
148             v.inv_row_j                         :=
                    input_forward_elimination.inv_row_odd;
149             v.address_row_i                     := 0;
150             v.flag_prev_row_i_at_odd_row        := '0';
151             v.wait_counter                      := 0;
152             v.flag_waiting_for_bram_update      := '0';
153           elsif r.index_i >= P_BANDS-2 and r.index_j >= P_BANDS
                  -3 then
154             -- Forward elimination is finished.
155             v.valid_data              := '0';
156             v.flag_write_to_odd_row   := '0';
157             v.flag_write_to_even_row  := '0';
158             v.read_address_even       := P_BANDS/2-1;
159             v.read_address_even       := P_BANDS/2-1;
160           else
161             v.valid_data := '0';
162             v.index_i    := r.index_i + 1;
163             -- Set v.index_j to be the same as v.index i as
                    index_j gets updated in EVEN_j_WRITE and
                    ODD_j_WRITE anyways
164             v.index_j    := r.index_i + 1;
165             -- flag_prev_row_i_at_odd_row set by EVEN_j_WRITE
166             -- or if previous index_j = P_BANDS-1 and index_i=
                    P_BANDS-3, then
167             -- it was set by ODD_j_WRITE
168             if r.flag_prev_row_i_at_odd_row = '1' then
169               v.flag_write_to_odd_row  := '0';
170               v.flag_write_to_even_row := '1';
171             else
172               v.flag_write_to_odd_row  := '1';
173               v.flag_write_to_even_row := '0';
174             end if;
175             v.wait_counter                  := 0;
176             v.flag_waiting_for_bram_update := '0';
177             if v.flag_write_to_even_row = '1' then
178               -- index_i at odd row i
179               -- address row i?
180               v.address_row_i       := r.address_row_i;
181               v.address_row_j       := r.address_row_i+1;
182               -- write address is changed in EVEN_j_WRITE before
                      writing
183               v.write_address_even := r.address_row_i;
184               v.write_address_odd  := r.address_row_i;
185               v.row_i               := input_forward_elimination.
                      row_odd;
186               v.row_j               := input_forward_elimination.
                      row_even;
187               v.inv_row_i           := input_forward_elimination.
                      inv_row_odd;
```

```
188                 v.inv_row_j           := input_forward_elimination.
                        inv_row_even;
189               else
190                 -- index i at even row
191                 v.address_row_i       := r.address_row_i+1;
192                 v.write_address_odd   := r.write_address_odd +1;
193                 v.address_row_j       := r.address_row_i+1;
194                 v.write_address_even  := r.write_address_even;
195                 v.row_i               := input_forward_elimination.
                        row_even;
196                 v.row_j               := input_forward_elimination.
                        row_odd;
197                 v.inv_row_i           := input_forward_elimination.
                        inv_row_even;
198                 v.inv_row_j           := input_forward_elimination.
                        inv_row_odd;
199               end if;
200             end if;
201             if v.row_i(v.index_i) = 0 then
202               v.forward_elimination_write_state := SWAP_ROWS;
203               v.flag_start_swapping_rows        := '1';
204               -- insecure about the reading process here...
205               v.read_address_even               := r.
                      read_address_even +1;
206               v.read_address_odd                := r.
                      read_address_odd +1;
207             else
208               if v.flag_write_to_even_row = '1' then  -- and data
                      is ready
209                 v.forward_elimination_write_state := EVEN_j_WRITE;
210                 v.read_address_even               := r.
                        read_address_even;
211                 v.read_address_odd                := r.
                        read_address_odd+1;
212               else
213                 v.forward_elimination_write_state := ODD_j_WRITE;
214                 v.read_address_even               := r.
                        read_address_even+1;
215                 v.read_address_odd                := r.
                        read_address_odd+1;
216               end if;
217             end if;
218           when SWAP_ROWS =>
219             -- wait until received new swapped rows from swapped
                    row module
220             v.flag_start_swapping_rows := '0';
221             if output_swap_rows.valid_data = '1' then
222               -- A swap of rows have happened. The forward
                      elimination can continue
223               if output_swap_rows.flag_prev_row_i_at_odd_row = '1'
```

```
                then
224               v.forward_elimination_write_state := EVEN_j_WRITE;
225               -- read data. Need to read an odd row
226               v.read_address_odd                := output_swap_rows.read_address_odd;
227               v.read_address_even               := output_swap_rows.read_address_even;
228             else
229               v.forward_elimination_write_state := ODD_j_WRITE;
230               --read data. Need to read an even row
231               v.read_address_odd                := output_swap_rows.read_address_odd +1;
232               v.read_address_even               := output_swap_rows.read_address_even +1;
233             end if;
234           end if;
235         when EVEN_j_WRITE =>
236           -- Need to check if i two cycles forward is at new place..
237           if r.flag_waiting_for_bram_update = '0' then
238             v.valid_data              := '1';
239             v.flag_write_to_even_row  := '1';
240             v.flag_write_to_odd_row   := '0';
241             v.valid_data              := '1';
242             v.index_j                 := r.index_j+1;
243             v.row_j                   := input_forward_elimination.row_even;
244             v.inv_row_j               := input_forward_elimination.inv_row_even;
245             v.address_row_i           := r.address_row_i;
246             if r.index_j              <= P_BANDS-2 then
247               v.write_address_even := r.write_address_even+1;
248               v.write_address_odd  := r.write_address_odd +1;
249             end if;
250             if v.index_j >= P_BANDS-2 then
251               v.index_i_two_cycles_ahead := r.index_i+1;
252               v.index_j_two_cycles_ahead := r.index_i+2;
253             end if;
254           end if;
255
256
257           if v.index_j >= P_BANDS-2 and v.index_j -v.index_i_two_cycles_ahead < B_RAM_WAIT_CLK_CYCLES and r.wait_counter < B_RAM_WAIT_CLK_CYCLES-(v.index_j-v.index_i_two_cycles_ahead) then
258             -- Need to wait for the row to update before reading it
259             v.wait_counter                    := r.wait_counter+1;
260             v.flag_waiting_for_bram_update := '1';
261           else
```

```
262             v.wait_counter                    := 0;
263             v.flag_waiting_for_bram_update := '0';
264             if v.index_j                      <= P_BANDS-2 then
265               v.forward_elimination_write_state := ODD_j_WRITE;
266             end if;
267             -- read new data. Data need to be read two clock
                    cycles in advance
268             if(r.read_address_even <= P_BANDS/2-1 and r.
                    index_j_two_cycles_ahead <= P_BANDS-3) then
269               -- need to read an even row
270               v.read_address_even         := r.read_address_even
                      +1;
271               v.read_address_odd          := r.read_address_odd
                      +1;
272               v.index_j_two_cycles_ahead := r.
                      index_j_two_cycles_ahead +1;
273             elsif v.index_j >= P_BANDS-3 then
274               -- new i, update
275               if r.index_i_two_cycles_ahead <= P_BANDS-3 then
276                 --v.index_i_two_cycles_ahead := r.
                        index_i_two_cycles_ahead+1;
277                 --v.index_j_two_cycles_ahead := r.
                        index_i_two_cycles_ahead+2;
278                 if r.flag_prev_row_i_at_odd_row = '1' then
279                   --next row i will be located in an even
                          indexed row
280                   v.read_address_even                    := r.
                          address_row_i+1;
281                   v.read_address_odd                     := r.
                          address_row_i+1;
282                   v.read_address_row_i_two_cycles_ahead := r.
                          read_address_row_i_two_cycles_ahead+1;
283                   v.flag_prev_row_i_at_odd_row           := '0';
284                 else
285                   -- next row i will be located in an odd
                          indexed row
286                   -- Row even will be located at an address one
                          increment ahead
287                   v.read_address_odd                     := r.
                          read_address_row_i_two_cycles_ahead;
288                   v.read_address_even                    := r.
                          read_address_row_i_two_cycles_ahead+1;
289                   v.read_address_row_i_two_cycles_ahead := r.
                          read_address_row_i_two_cycles_ahead;
290                   v.flag_prev_row_i_at_odd_row           := '1';
291                 end if;
292               end if;
293             end if;
294           end if;
295
```

```
296
297             when ODD_j_WRITE =>
298               -- Need to check if i two cycles forward is at new
                      place..
299               if r.flag_waiting_for_bram_update = '0' then
300                 v.valid_data                := '1';
301                 v.flag_write_to_even_row := '0';
302                 v.flag_write_to_odd_row  := '1';
303                 v.valid_data                := '1';
304                 v.row_j                     :=
                        input_forward_elimination.row_odd;
305                 v.inv_row_j                 :=
                        input_forward_elimination.inv_row_odd;
306                 v.index_j                   := r.index_j+1;
307                 v.address_row_i             := r.address_row_i;
308                 v.write_address_even        := r.write_address_even;
309                 v.write_address_odd         := r.write_address_odd;
310               end if;
311
312               if v.index_j >= P_BANDS-1 and v.index_j-r.
                      index_i_two_cycles_ahead < B_RAM_WAIT_CLK_CYCLES
                      and r.wait_counter < B_RAM_WAIT_CLK_CYCLES-(v.
                      index_j-r.index_i_two_cycles_ahead) then
313                 -- Need to wait for the row to update before reading
                        it
314                 v.wait_counter                  := r.wait_counter+1;
315                 v.flag_waiting_for_bram_update := '1';
316               else
317                 v.wait_counter                  := 0;
318                 v.flag_waiting_for_bram_update := '0';
319                 -- Set next state
320                 if v.index_j                    <= P_BANDS-3 then
321                   v.forward_elimination_write_state := EVEN_j_WRITE;
322                 else
323                   --New iteration of the outermost loop
324                   if v.index_j = P_BANDS-1 and v.index_i = P_BANDS-3
                          then
325                     -- The last row_i of the forward elimination is
                            located at an
326                     -- even indexed row.
327                     v.flag_prev_row_i_at_odd_row := '0';
328                   end if;
329                   v.forward_elimination_write_state :=
                          CHECK_DIAGONAL_ELEMENT_IS_ZERO;
330                 end if;
331                 -- read new data. Data need to be read two clock
                        cycles in advance
332                 if(r.read_address_odd <= P_BANDS/2-1 and r.
                        index_j_two_cycles_ahead <= P_BANDS-1 and v.
                        index_j < P_BANDS-1) then
```

```
333                 -- need to read an odd row
334                 v.read_address_even         := r.read_address_even;
335                 v.read_address_odd          := r.read_address_odd;
336                 v.index_j_two_cycles_ahead := r.
                        index_j_two_cycles_ahead +1;
337               elsif v.index_j >= P_BANDS-1 then
338                 -- In the previous clock cycle a new index i was
                        read
339                 v.read_address_even         := r.read_address_even;
340                 v.read_address_odd          := r.read_address_odd;
341                 v.index_j_two_cycles_ahead := r.
                        index_j_two_cycles_ahead +1;
342
343               --   if r.flag_prev_row_i_at_odd_row = '1' then
344               --     v.read_address_even         := r.
                      read_address_even;
345               --     v.read_address_odd          := r.
                      read_address_even;
346               --     v.index_j_two_cycles_ahead := r.
                      index_j_two_cycles_ahead +1;
347               --   else
348               --     v.read_address_even         := r.
                      read_address_even;
349               --     v.read_address_odd          := r.
                      read_address_odd;
350               --     v.index_j_two_cycles_ahead := r.
                      index_j_two_cycles_ahead +1;
351               --   end if;
352                 end if;
353             end if;
354           when others =>
355             v.forward_elimination_write_state := STATE_IDLE;
356             v.flag_write_to_odd_row           := '0';
357             v.flag_write_to_even_row          := '0';
358       end case;
359     end if;
360
361     if(reset_n = '0') then
362       v.index_i                         := 0;
363       v.index_j                         := 1;
364       v.valid_data                      := '0';
365       v.address_row_i                   := 0;
366       v.flag_write_to_even_row          := '0';
367       v.flag_write_to_odd_row           := '0';
368       v.forward_elimination_write_state := STATE_IDLE;
369     end if;
370     r_in                                          <= v;
371     --data
372     output_top_level.row_j                        <= r.row_j;
373     output_top_level.row_i                        <= r.row_i;
```

```
374       output_top_level.inv_row_j                          <= r.
              inv_row_j;
375       output_top_level.inv_row_i                          <= r.
              inv_row_i;
376       --control
377       output_top_level.index_i                            <= r.
              index_i;
378       output_top_level.index_j                            <= r.
              index_j;
379       output_top_level.read_address_even                  <= r.
              read_address_even;
380       output_top_level.read_address_odd                   <= r.
              read_address_odd;
381       output_top_level.write_address_even                 <= r.
              write_address_even;
382       output_top_level.write_address_odd                  <= r.
              write_address_odd;
383       output_top_level.valid_data                         <= r.
              valid_data;
384       output_top_level.forward_elimination_write_state <= r.
              forward_elimination_write_state;
385       output_top_level.flag_write_to_odd_row              <= r.
              flag_write_to_odd_row;
386       output_top_level.flag_write_to_even_row             <= r.
              flag_write_to_even_row;
387       output_top_level.state_reg                          <= r.
              state_reg;
388
389     end process;
390
391
392     sequential_process : process(clk, clk_en)
393     begin
394       if(rising_edge(clk) and clk_en = '1') then
395         r <= r_in;
396       end if;
397     end process;
398
399   end Behavioral;
```

C.9 Last division

Listing C.9: Last division

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
```

```
6  use work.Common_types_and_functions.all;
7
8  entity top_last_division is
9    port(clk                  : in  std_logic;
10        reset_n              : in  std_logic;
11        clk_en               : in  std_logic;
12        input_last_division  : in  input_last_division_reg_type;
13        output_last_division : out output_last_division_reg_type)
              ;
14 end top_last_division;
15
16 architecture Behavioral of top_last_division is
17
18   signal r, r_in             : input_last_division_reg_type;
19 -- number of shifts required to approximate the division
20   signal divisor_is_negative : std_logic;
21   -- If the divisor is negative, we need to take two's
       complement of the divisor
22   signal divisor             : std_logic_vector(PIXEL_DATA_WIDTH
       *2 -1 downto 0);
23   signal divisor_valid       : std_logic
                                    := '0';
24   signal remainder_valid     : std_logic
                                    := '0';
25   type remainders_array is array(0 to PIXEL_DATA_WIDTH*2-2) of
       std_logic_vector(PIXEL_DATA_WIDTH*2-1 downto 0);
26   signal remainders          : remainders_array;
27   constant ONE               : signed(PIXEL_DATA_WIDTH*2-1
       downto 0) := (0 => '1', others => '0');
28   signal msb_index           : integer range 0 to 31;  -- msb of
        the divisor(unsigned)
29   signal msb_valid           : std_logic
                                    := '0';
30   -- to be used in two's complement.
31   signal divisor_lut         : unsigned(DIV_PRECISION-1 downto
       0);
32   signal divisor_inv         : unsigned(DIV_PRECISION-1 downto
       0);
33
34
35 begin
36
37   division_lut_1 : entity work.division_lut
38     port map (
39       y     => divisor_lut,
40       y_inv => divisor_inv);
41
42   input_to_divisor_lut : process(msb_valid, msb_index)
43   begin
44     if msb_valid = '1' and msb_index<=DIV_PRECISION then
```

```
45        divisor_lut <= to_unsigned(to_integer(unsigned(divisor)),
              DIV_PRECISION);
46      else
47        divisor_lut <= to_unsigned(0, DIV_PRECISION);
48      end if;
49    end process;
50
51    check_if_divisor_is_negative : process(input_last_division.
          state_reg.state, input_last_division.row_i,
          input_last_division.valid_data, reset_n)
52    begin
53      if reset_n = '0' or not(input_last_division.state_reg.state
            = STATE_LAST_DIVISION) then
54        divisor_valid       <= '0';
55        divisor_is_negative <= '0';
56        divisor             <= std_logic_vector(to_signed(1,
              PIXEL_DATA_WIDTH*2));
57      elsif(input_last_division.row_i(input_last_division.index_i)
          (PIXEL_DATA_WIDTH*2-1) = '1' and input_last_division.
          valid_data = '1') then
58        -- row[i][i] is negative
59        -- using the absolute value
60        divisor_is_negative <= '1';
61        divisor             <= std_logic_vector(abs(signed(
              input_last_division.row_i(input_last_division.index_i))
              ));
62        divisor_valid       <= '1';
63      elsif input_last_division.valid_data = '1' then
64        divisor_is_negative <= '0';
65        divisor             <= std_logic_vector(
              input_last_division.row_i(input_last_division.index_i))
              ;
66        divisor_valid       <= '1';
67      else
68        divisor_valid       <= '0';
69        divisor_is_negative <= '0';
70        divisor             <= std_logic_vector(to_signed(1,
              PIXEL_DATA_WIDTH*2));
71      end if;
72    end process;
73
74
75  -- generate PIXEL_DATA_WIDTH*2-1 number of shifters that shifts
76  -- A[i][i] n places in order to see how many shifts yield the
        best
77  -- approximation to the division. Don't need to shift the
78  -- 31 bit as this is the sign bit.
79    generate_shifters : for i in 1 to PIXEL_DATA_WIDTH*2-1
          generate
80      signal remainder_after_approximation_i :
```

```
        remainder_after_approximation_record;
81    begin
82      process(divisor, divisor_valid, reset_n, input_last_division
            .state_reg)
83      begin
84        if reset_n = '0' or not(input_last_division.state_reg.
              state = STATE_LAST_DIVISION) then
85          remainder_after_approximation_i.remainder        <=
                std_logic_vector(shift_right(signed(divisor), i));
86          remainder_after_approximation_i.number_of_shifts <= i;
87          remainder_after_approximation_i.remainder_valid  <= '0';
88        elsif divisor_valid = '1' then
89          remainder_after_approximation_i.remainder        <=
                std_logic_vector(shift_right(signed(divisor), i));
90          remainder_after_approximation_i.number_of_shifts <= i;
91          remainder_after_approximation_i.remainder_valid  <= '1';
92        else
93          remainder_after_approximation_i.remainder        <=
                std_logic_vector(shift_right(signed(divisor), i));
94          remainder_after_approximation_i.number_of_shifts <= i;
95          remainder_after_approximation_i.remainder_valid  <= '0';
96        end if;
97      end process;
98      remainders(i-1) <= remainder_after_approximation_i.remainder
            ;
99      remainder_valid <= remainder_after_approximation_i.
            remainder_valid;
100   end generate;
101
102
103   find_msb : process(divisor_valid, input_last_division, reset_n
          , divisor)
104   begin
105     if divisor_valid = '1' and reset_n = '1' then
106       --For PIXEL_DATA_WIDTH = 16.
107       if divisor(30) = '1' then
108         msb_index <= 30;
109         msb_valid <= '1';
110       elsif divisor(29) = '1' then
111         msb_index <= 29;
112         msb_valid <= '1';
113       elsif divisor(28) = '1' then
114         msb_index <= 28;
115         msb_valid <= '1';
116       elsif divisor(27) = '1' then
117         msb_index <= 27;
118         msb_valid <= '1';
119       elsif divisor(26) = '1'then
120         msb_index <= 26;
121         msb_valid <= '1';
```

```
122         elsif divisor(25) = '1' then
123           msb_index <= 25;
124           msb_valid <= '1';
125         elsif divisor(24) = '1' then
126           msb_index <= 24;
127           msb_valid <= '1';
128         elsif divisor(23) = '1' then
129           msb_index <= 23;
130           msb_valid <= '1';
131         elsif divisor(22) = '1' then
132           msb_index <= 22;
133           msb_valid <= '1';
134         elsif divisor(21) = '1' then
135           msb_index <= 21;
136           msb_valid <= '1';
137         elsif divisor(20) = '1' then
138           msb_index <= 20;
139           msb_valid <= '1';
140         elsif divisor(19) = '1' then
141           msb_index <= 19;
142           msb_valid <= '1';
143         elsif divisor(18) = '1' then
144           msb_index <= 18;
145           msb_valid <= '1';
146         elsif divisor(17) = '1'then
147           msb_index <= 17;
148           msb_valid <= '1';
149         elsif divisor(16) = '1' then
150           msb_index <= 16;
151           msb_valid <= '1';
152         elsif divisor(15) = '1' then
153           msb_index <= 15;
154           msb_valid <= '1';
155         elsif divisor(14) = '1' then
156           msb_index <= 14;
157           msb_valid <= '1';
158         elsif divisor(13) = '1' then
159           msb_index <= 13;
160           msb_valid <= '1';
161         elsif divisor(12) = '1' then
162           msb_index <= 12;
163           msb_valid <= '1';
164         elsif divisor(11) = '1' then
165           msb_index <= 11;
166           msb_valid <= '1';
167         elsif divisor(10) = '1'then
168           msb_index <= 10;
169           msb_valid <= '1';
170         elsif divisor(9) = '1' then
171           msb_index <= 9;
```

```
172           msb_valid <= '1';
173         elsif divisor(8) = '1' then
174           msb_index <= 8;
175           msb_valid <= '1';
176         elsif divisor(7) = '1' then
177           msb_index <= 7;
178           msb_valid <= '1';
179         elsif divisor(6) = '1' then
180           msb_index <= 6;
181           msb_valid <= '1';
182         elsif divisor(5) = '1' then
183           msb_index <= 5;
184           msb_valid <= '1';
185         elsif divisor(4) = '1' then
186           msb_index <= 4;
187           msb_valid <= '1';
188         elsif divisor(3) = '1' then
189           msb_index <= 3;
190           msb_valid <= '1';
191         elsif divisor(2) = '1' then
192           msb_index <= 2;
193           msb_valid <= '1';
194         elsif divisor(1) = '1' then
195           msb_index <= 1;
196           msb_valid <= '1';
197         elsif divisor(0) = '1' then
198           msb_index <= 0;
199           msb_valid <= '1';
200         else
201           msb_valid <= '0';
202           msb_index <= 0;
203         end if;
204       else
205         msb_index <= 0;
206         msb_valid <= '0';
207       end if;
208     end process;
209
210
211     comb_process : process(input_last_division, r, reset_n,
          divisor_is_negative, divisor, remainder_valid, remainders,
          msb_valid, divisor, divisor_inv, msb_index)
212       variable v                    : input_last_division_reg_type
            ;
213       variable divisor_inv_from_lut : integer range 0 to 2**
            DIV_PRECISION := 0;
214     begin
215
216       v := r;
217       if(input_last_division.state_reg.state = STATE_LAST_DIVISION
```

```
          and input_last_division.valid_data = '1' and
          remainder_valid = '1' and msb_valid = '1' and reset_n =
          '1') then
218       v              := input_last_division;
219       v.best_approx  := INITIAL_BEST_APPROX;
220       v.msb_index    := msb_index;
221
222
223       if v.msb_index <= DIV_PRECISION then
224         divisor_inv_from_lut := to_integer(divisor_inv);
225       else
226         --Using shifting approach
227         divisor_inv_from_lut := to_integer(divisor_inv);
228
229         -- The best approximation may be either the msb-shifted
              division, or the
230         -- msb+1 shifted division.
231         v.best_approx.remainder         := remainders(v.msb_index
              );
232         v.best_approx.number_of_shifts := v.msb_index;
233         -- The best approximation to the divisor may be larger
              than the divisor.
234         if to_integer(signed(divisor))- to_integer(shift_left(
              to_signed(1, PIXEL_DATA_WIDTH*2), v.best_approx.
              number_of_shifts)) > to_integer(shift_left(to_signed
              (1, PIXEL_DATA_WIDTH*2), v.best_approx.
              number_of_shifts+1))- to_integer(signed(divisor))
              then
235           -- This is a better approximation
236           v.best_approx.remainder         := std_logic_vector(
                to_signed(to_integer(shift_left(to_signed(1,
                PIXEL_DATA_WIDTH*2), v.best_approx.number_of_shifts
                +1))-to_integer(signed(divisor)), PIXEL_DATA_WIDTH
                *2));
237           v.best_approx.number_of_shifts := v.best_approx.
                number_of_shifts+1;
238         end if;
239       end if;
240
241
242   -- Doing division
243       if divisor_is_negative = '1' then
244         for i in 0 to P_BANDS-1 loop
245           if v.msb_index <= DIV_PRECISION then
246             v.inv_row_i(i) := shift_right(input_last_division.
                  inv_row_i(i)*divisor_inv_from_lut, DIV_PRECISION)
                  ;
247             --v.inv_row_i(i) := shift_right(input_last_division.
                  inv_row_i(i), v.best_approx.number_of_shifts);
248             -- Negating the number with two's complement
```

```
249             v.inv_row_i(i) := not(v.inv_row_i(i)) + ONE;
250           else
251             v.inv_row_i(i) := shift_right(input_last_division.
                    inv_row_i(i), v.best_approx.number_of_shifts);
252             -- Negating the number with two's complement
253             v.inv_row_i(i) := not(v.inv_row_i(i)) + ONE;
254           end if;
255         end loop;
256       else
257         for i in 0 to P_BANDS-1 loop
258           if v.msb_index <= DIV_PRECISION then
259             v.inv_row_i(i) := shift_right(input_last_division.
                    inv_row_i(i)*divisor_inv_from_lut, DIV_PRECISION)
                    ;
260           else
261             v.inv_row_i(i) := shift_right(input_last_division.
                    inv_row_i(i), v.best_approx.number_of_shifts);
262           end if;
263         end loop;
264       end if;
265     end if;
266
267     if(reset_n = '0' or input_last_division.state_reg.state /=
          STATE_LAST_DIVISION) then
268       v.valid_data  := '0';
269       v.best_approx := INITIAL_BEST_APPROX;
270       v.msb_index   := 31;
271     end if;
272     r_in <= v;
273   end process;
274
275   output_last_division.new_inv_row_i          <= r.inv_row_i;
276   output_last_division.valid_data             <= r.valid_data;
277   output_last_division.index_i                <= r.index_i;
278   output_last_division.write_address_even     <= r.
          write_address_even;
279   output_last_division.write_address_odd      <= r.
          write_address_odd;
280   output_last_division.flag_write_to_even_row <= r.
          flag_write_to_even_row;
281   output_last_division.state_reg              <= r.state_reg;
282
283   sequential_process : process(clk)
284   begin
285     if rising_edge(clk) then
286       if clk_en = '1' then
287         r <= r_in;
288       end if;
289     end if;
290   end process;
```

```
291
292 end Behavioral;
```