FPGA Implementation of Hyperspectral Anomaly Detection Algorithm

Martin Haukali
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 parallelizable 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 $P_{\text{bands}}$ have been made. $P_{\text{bands}}$ 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 $P_{\text{bands}} \times P_{\text{bands}}$, per clock cycle. The design is to be implemented on a Zynq-7000 series System-on-Chip (SoC).
Sammendrag


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.

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 available 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

1 Introduction .............................................................. 1
   1.1 Motivation .......................................................... 1
       1.1.1 Main contributions ........................................... 3
       1.1.2 Problem statement ........................................... 3
       1.1.3 Master thesis overview ....................................... 3

2 Background theory ..................................................... 5
   2.1 Hyperspectral imaging .............................................. 5
       2.1.1 AVIRIS ......................................................... 5
           2.1.1.1 Cuprite scene ......................................... 5
       2.1.2 NTNU SmallSat project ....................................... 6
   2.2 NTNU SmallSat’s hardware platform .................................. 8
       2.2.1 AXI-Stream .................................................... 8
   2.3 Anomaly detection .................................................. 9
       2.3.1 Reed-Xiaoli algorithm ....................................... 9
       2.3.2 Local RX algorithm ....................................... 9
       2.3.3 Adaptive Causal anomaly detection ......................... 10
       2.3.4 Adaptive Local RX ......................................... 13
   2.4 Inverse matrix ..................................................... 14
       2.4.1 Gauss-Jordan elimination ................................... 15
           2.4.1.1 Forward elimination .................................. 15
           2.4.1.2 Backward elimination .................................. 16
           2.4.1.3 Last division ......................................... 17

3 Review of state of the art anomaly detectors .......................... 19
   3.1 Experiments on synthetic images ................................... 19
       3.1.1 RX detection results ....................................... 23
           3.1.1.1 Hsieh-mimicked image ................................ 23
           3.1.1.2 Sim:30 AVIRIS scene .................................. 24
           3.1.1.3 Sim:Aviris01 scene ................................... 24
       3.1.2 LRX detection results ....................................... 25
           3.1.2.1 Hsieh mimicked image ................................ 25
           3.1.2.2 Sim:30 AVIRIS scene .................................. 25
           3.1.2.3 Sim:Aviris01 ........................................... 26
       3.1.3 ALRX detection results ...................................... 26
           3.1.3.1 SIM:30 AVIRIS .......................................... 26
           3.1.3.2 Sim:Aviris01 ........................................... 27
## CONTENTS

3.1.4 ACAD .............................................. 27
   3.1.4.1 SIM AVIRIS_30_30 ......................... 27
   3.1.4.2 SimAviris01 ................................... 28

3.2 Testing on real image data ........................................ 28
   3.2.1 RX ............................................... 29
   3.2.2 LRX ............................................... 29
   3.2.3 ACAD .............................................. 29
   3.2.4 Choice of anomaly detector algorithm ..................... 30

4 Proposed hardware implementation ................................. 33
   4.1 Memory considerations ................................... 33
      4.1.1 Storing and updating matrices in ACAD ................. 33
      4.1.1.1 Using registers .................................. 34
      4.1.1.2 Using BRAM ...................................... 35
   4.2 Proposed implementation ..................................... 39
   4.3 Shiftregister ......................................... 41
   4.4 ACAD correlation ....................................... 42
      4.4.1 Normalizing with k .................................. 45
   4.5 Inverse computation ..................................... 46
      4.5.1 Elimination core ...................................... 48
      4.5.2 FSM inverse .......................................... 50
      4.5.3 Forward elimination ................................... 52
      4.5.4 Backward elimination .................................. 55
      4.5.5 Last division ........................................ 58
      4.5.6 Output inverse matrix ................................. 59
      4.5.7 Inverse pipeline stages ............................... 59
      4.5.8 Execution time expectations inverse computation ....... 61
      4.5.9 Division ............................................. 63
         4.5.9.1 Using the division operator "/" ...................... 63
         4.5.9.2 Adaptive shifting .................................. 64
         4.5.9.3 LUT approach ..................................... 65

5 Results ....................................................................... 69
   5.1 Synthesis ............................................... 69
      5.1.1 Shiftregister ......................................... 70
      5.1.2 ACAD correlation ...................................... 70
         5.1.2.1 Pixel data width = 10 ............................ 73
      5.1.3 ACAD inverse ........................................ 73
      5.1.4 Timing results ....................................... 76
         5.1.4.1 WNS ACAD correlation ......................... 76
         5.1.4.2 WNS division operator ........................... 76
         5.1.4.3 Worst Negative Slack adaptive shifting approach 77
         5.1.4.4 Worst Negative Slack LUT approach ............ 77
   5.2 Simulation ............................................... 77
      5.2.1 Shiftregister ......................................... 77
      5.2.2 ACAD correlation ...................................... 79
      5.2.3 Inverse ............................................... 82
# CONTENTS

6  Discussion .......................................................... 85
   6.1 Resource usage ................................................. 85
       6.1.1 DSP usage $\text{Pixel\_data\_width} = 16$ .................. 85
       6.1.2 $\text{Pixel\_data\_width} = 10$ ............................. 86
   6.2 Timing results .................................................. 86
       6.2.1 ACAD correlation ........................................... 86
       6.2.2 ACAD inverse ............................................... 86
       6.2.3 Simulation results ......................................... 87

7  Conclusion ........................................................... 89
   7.1 Future work ..................................................... 90
       7.1.1 Optimization ............................................... 90

Appendices .............................................................. 93

A  MATLAB hyperspectral ............................................. 95
   A.1 High level models of algorithms ............................... 95
       A.1.1 Gauss-Jordan elimination .................................. 95
       A.1.2 RX anomaly detector ....................................... 97
       A.1.3 LRX anomaly detector ...................................... 98
       A.1.4 ALRX anomaly detector .................................... 99
       A.1.5 ACAD anomaly detector .................................... 101
   A.2 Testing .......................................................... 105
       A.2.1 Hyper demo detectors ...................................... 105
       A.2.2 Generating synthetic images ............................... 108

B  VHDL Code description ............................................ 123

C  VHDL code ........................................................... 125
   C.1 ACAD correlation ............................................... 125
   C.2 Elimination core ............................................... 133
   C.3 BRAM SDP 18kbit .............................................. 142
   C.4 Package Common types and functions .......................... 143
   C.5 Swap rows ..................................................... 152
   C.6 ACAD inverse ................................................... 157
   C.7 Shiftregister .................................................... 195
   C.8 Forward elimination .......................................... 197
   C.9 Last division ................................................... 207
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Functional concept of HSI [1].</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Image of an algae bloom along the coast of Troms in Norway [2].</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Band 220 from the Cuprite scene 02 [3].</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Spectral signatures of minerals from the Cuprite mining district [4].</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>A spectral pixel vector.</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Hyperspectral image cube.</td>
<td>7</td>
</tr>
<tr>
<td>2.5</td>
<td>Push-broom hyperspectral imager mode of operation [5].</td>
<td>7</td>
</tr>
<tr>
<td>2.6</td>
<td>Zynq-7000 architecture [6].</td>
<td>8</td>
</tr>
<tr>
<td>2.7</td>
<td>Visualization of a kernel of size $K \times K$ used in LRX.</td>
<td>10</td>
</tr>
<tr>
<td>2.8</td>
<td>Visualizing processing of pixels in ACAD.</td>
<td>11</td>
</tr>
<tr>
<td>2.9</td>
<td>Results of noise tests [7].</td>
<td>13</td>
</tr>
<tr>
<td>2.10</td>
<td>Pseudo-code for computing the inverse of a matrix by Gauss-Jordan elimination [8].</td>
<td>15</td>
</tr>
<tr>
<td>2.11</td>
<td>Pseudo-code for computing the forward elimination in Gauss-Jordan elimination [8].</td>
<td>16</td>
</tr>
<tr>
<td>2.12</td>
<td>Pseudo-code for computing the backward elimination in Gauss-Jordan elimination [8].</td>
<td>16</td>
</tr>
<tr>
<td>2.13</td>
<td>Pseudo-code for computing the last division in Gauss-Jordan elimination [8].</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Test of RX (GRX) and LRX algorithms [9].</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>First class of synthetic images: 200 $\times$ 200 synthetic image with 25 inserted anomaly panels as describe by Hsueh in [10].</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Second class of synthetic images: Synthetic 30 $\times$ 30 image with an inserted 2x2 anomalous panel inserted into the center.</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>Expected anomaly map for the third class of synthetic images created.</td>
<td>23</td>
</tr>
<tr>
<td>3.5</td>
<td>RX AD result.</td>
<td>23</td>
</tr>
<tr>
<td>3.6</td>
<td>Generated anomaly map.</td>
<td>23</td>
</tr>
<tr>
<td>3.7</td>
<td>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.</td>
<td>23</td>
</tr>
<tr>
<td>3.8</td>
<td>RX AD results for the Sim30 30AVIRIS scene.</td>
<td>24</td>
</tr>
<tr>
<td>3.9</td>
<td>RX AD results for the Sim_AAVIS01 scene.</td>
<td>24</td>
</tr>
<tr>
<td>3.10</td>
<td>Band 220 from the Cuprite scene 02 [3].</td>
<td>28</td>
</tr>
<tr>
<td>3.11</td>
<td>Result from RX AD on Cuprite image data.</td>
<td>29</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

3.12 Result from LRX AD with a kernel size of $K=23$ on Cuprite image scene .......................... 29
3.13 Anomaly map created by ACAD (yellow dots) overlayed over Figure 3.10 .......................... 29
4.1 Matrix $A$ and $A^{-1}$ used in Gauss-Jordan elimination .................................................. 34
4.2 Zynq memory resources [12] ........................................................................................... 34
4.3 Estimated time spent updating $R(x_k)$ ............................................................................. 37
4.4 Maximum memory accessing requirement by the Gauss-Jordan elimination ..................... 37
4.5 BRAM addressing scheme for storage of matrices utilized by ACAD. One column of the matrix is stored per 36kbit BRAM .............................................................. 38
4.6 BRAM hierarchy, showing two 18kbit BRAM blocks contained within one 36kbit BRAM block .................................................................................................................. 39
4.8 Architecture of the Shiftregister block ............................................................................. 41
4.9 Data output of the Shiftregister block .............................................................................. 42
4.10 Data flow within the ACAD correlation module ............................................................. 43
4.11 An example of the data handling done by ACAD correlation. For this example $P_{bands} = 4$ ..................................................................................................................... 44
4.12 The operations computed by the Elimination core, utilized by both the Forward elimination and the Backward elimination block ......................................................... 48
4.13 The operations computed by the Elimination core, utilized by both the Forward elimination and the Backward elimination block ......................................................... 49
4.14 Elimination core part one ............................................................................................... 49
4.15 Elimination core part two ............................................................................................... 50
4.16 FSM controlling ACAD inverse shown in Figure 4.12 ..................................................... 51
4.17 FSM controlling Forward elimination ............................................................................ 53
4.18 The check done in state Check_diagonal_element_is_zero ............................................. 53
4.19 Operations done in Swap rows ........................................................................................ 54
4.20 Even_j_write in the Backward elimination state ............................................................ 54
4.21 Odd_j_write in the forward elimination state ................................................................. 55
4.22 FSM controlling Backward elimination ........................................................................ 56
4.23 Even_j_write in backward elimination ........................................................................ 57
4.24 Odd_j_write in backward elimination ........................................................................... 57
4.25 Odd_i_start .................................................................................................................... 58
4.26 Even_i_start .................................................................................................................. 58
4.27 FSM controlling Last division ....................................................................................... 59
4.28 Showing pipeline operations in the Store_correlation_matrix and Forward_elimination states ......................................................................................................................... 60
4.29 Showing pipeline operations in the Forward_elimination and Last_division states .......... 60
4.30 Showing pipeline operations in the Output_inverse_matrix state .................................. 61
4.31 Estimated execution time for computation of $R^{-1}(x_k)$ for an image of size $1088 \times 576$ in seconds .............................................................................................................. 63
4.32 Dataflow of block Last division using the division operator "/" for division ......................... 64
4.33 Architecture of block Last division, approximating division with an adaptive number of shifts ....................................................................................................................... 65
4.34 Architecture of block Last division, computing division using the LUT approach ............ 67
5.1 Shiftregister synthesis results ............................................................................................ 70
LIST OF FIGURES

5.2 Architecture of the implemented version of ACAD correlation, without normalization. ................................................. 71
5.3 Number of synthesized BRAM36E1 and DSP48E1 as a function of $P_{bands}$ for the ACAD correlation block. ......................... 72
5.4 Number of synthesized Slice Registers and Slice LUTs as a function of $P_{bands}$ for the ACAD correlation block. ......................... 72
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 $=10$. 73
5.6 Number of BRAMs synthesized for the Inverse block. .................. 74
5.7 Numbers of DSP48E1 synthesized for the Inverse block. ............. 74
5.8 Numbers of LUTs synthesized for the Inverse block. .................. 75
5.9 Numbers of registers synthesized for the Inverse block. ............... 75
5.10 Simulation of Shiftregister for $P_{bands} = 12$. .......................... 78

B.1 Two process method. ......................................................... 124
## List of Tables

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Properties of SimAviris01. Row and column locations are location of the center pixel in the kernel of size ( K \times K ).</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>LRX results on Hsueh scene.</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>LRX detection results on SIM30_30AVIRIS scene.</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>LRX results on SimAviris01 scene.</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>ALRX results on SIM30_30AVIRIS scene.</td>
<td>26</td>
</tr>
<tr>
<td>3.6</td>
<td>ALRX results on SimAviris01 scene.</td>
<td>27</td>
</tr>
<tr>
<td>3.7</td>
<td>ACAD results on SIM_AVIRIS_30_30 scene.</td>
<td>27</td>
</tr>
<tr>
<td>3.8</td>
<td>ACAD results on SimAviris01 scene.</td>
<td>28</td>
</tr>
<tr>
<td>3.9</td>
<td>Summary of comparison of anomaly detectors.</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>States of the inverse FSM.</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>States of the forward elimination FSM.</td>
<td>52</td>
</tr>
<tr>
<td>4.3</td>
<td>States of the backward elimination FSM.</td>
<td>55</td>
</tr>
<tr>
<td>4.4</td>
<td>States of the last division FSM.</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Timing results for ACAD correlation Pixel_data_width = 16.</td>
<td>76</td>
</tr>
<tr>
<td>5.2</td>
<td>Timing results for ACAD correlation Pixel_data_width = 10.</td>
<td>76</td>
</tr>
<tr>
<td>5.3</td>
<td>Synthesis results for Zedboard for Last division using the division operator &quot;/&quot;.</td>
<td>77</td>
</tr>
</tbody>
</table>
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 D Igital C omputer
- 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 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.

![Functional concept of HSI](image)

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. MOTIVATION

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.](image-url)
CHAPTER 2. BACKGROUND THEORY

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 2.2: Spectral signatures of minerals from the Cuprite mining district](image)

![Figure 2.3: A spectral pixel vector.](image)
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 an 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.
2.3 ANOMALY DETECTION

The Cube Direct Memory Access (DMA) 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 neighbour 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, $x$, is the observation test vector, the result of the RX algorithm is given by the filter in equation 2.1:

$$RX(x) = (x - u_b)^T \Sigma^{-1} (x - u_b),$$  \hspace{1cm} (2.1)

where $u_b$ is the estimated background clutter sample mean, computed from the set of all pixel vectors in the image (referred to as the global set). $\Sigma$ 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:

$$\delta_{K}^{LRX}(x) = x^T R_{K \times K}^{-1}(x)x, \quad (2.2)$$

where $x$ is the observation test pixel vector, $R_{K \times K}(x)$ is the correlation matrix of pixel vector $x$ 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.](image)

### 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 $R(x_k)$, presented in equation 2.3:

$$R(x_k) = \frac{1}{k} \sum_{i=1}^{k} x_i x_i^T, \quad (2.3)$$

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, $x_k$ 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 $x_1$, $...x_k$.

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
2.3. ANOMALY DETECTION

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

$$\tilde{R}(x_k) = R(x_k) - \sum_{t_j \in \Delta(k)} t_j t_j^T,$$  \hspace{1cm} (2.4)

where $$\Delta(k)$$ is the set of all earlier detected anomalous pixel vectors $$t_j$$ prior to the image pixel currently being processed, $$x_k$$. ACAD can then be defined as follows:

$$\delta^{ACAD}(x_k) = x_k^T \tilde{R}^{-1}(x_k)x_k.$$  \hspace{1cm} (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 $$x_1, \ldots, x_{k-1}$$ up to the pixel currently being processed, $$x_k$$, 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.](image)

$$x_{12}$$ 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:

$$u_k = \frac{1}{n_{ACAD}} \sum_{i=1}^{n_{ACAD}} \delta^{ACAD}(x_{k-i}).$$  \hspace{1cm} (2.6)
In order to classify if a pixel vector is anomalous the variable $t_k$ is introduced, defined in equation 2.7:

$$t_k = \delta_{ACAD}(x_k) - u_k. \quad (2.7)$$

If $t_k$ is greater than a predetermined value $\tau$, 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:

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

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 $\beta$, shown in equation 2.9:

$$\beta = \frac{\text{Image size}}{\text{size of anomaly}}. \quad (2.9)$$

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

$$n_{ACAD} = \frac{\text{N_PIXELS_TOT}}{\beta}, \quad (2.10)$$

where $N_{\text{PIXELS TOT}}$ 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 2.7.

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].
2.3. ANOMALY DETECTION

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.

\[
\delta_{k}^{\text{ALRX}}(x) = x^T \hat{R}_{K \times K}(x)^{-1} x. \tag{2.11}
\]

\[
\hat{R}_{K \times K}(x) = R_{K \times K}(x) - \sum_{t_j \in \Delta(k_{K \times K})} t_j t_j^T. \tag{2.12}
\]
where $\Delta(k_K \times K)$ is the set of previously detected anomalous pixel vectors $t_j$ located within the local window of size $K \times K$ with center in the image pixel vector currently being processed, $x$.

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 COordinate 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.
2.4. INVERSE MATRIX

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.
CHAPTER 2. BACKGROUND THEORY

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.

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

//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($j = i + 1; j < n; j + +$) {
  $A^{-1}$[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

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

//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
2.4. INVERSE MATRIX

2.4.1.3 Last division

Last division is the last step of the Gauss-Jordan elimination. The last division starts at the topmost 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.

```plaintext
//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]);
}
```

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.
CHAPTER 3. REVIEW OF STATE OF THE ART ANOMALY DETECTORS

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 equations 3.1 and 3.2:

\[
\text{false\_anomalies} = \text{predicted\_anomalies} - \text{true\_anomalies}, \quad (3.1)
\]

\[
\text{correctly\_predicted\_anomalies} = \frac{\text{predicted\_anomalies\_in\_reference\_map}}{\text{reference\_anomalies}} \quad (3.2)
\]

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. 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 cannot 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 $200 \times 200$ 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](image)

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

Additionally, synthetic images with a size of $30 \times 30$ pixels were created with an anomalous panel of size $2 \times 2$ 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.
CHAPTER 3. REVIEW OF STATE OF THE ART ANOMALY DETECTORS

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 SimAviris01. 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 SimAviris01.

Table 3.1: Properties of SimAviris01. Row and column locations are location of the center pixel in the kernel of size $K \times K$.

<table>
<thead>
<tr>
<th>Scene</th>
<th>Row</th>
<th>Column</th>
<th>Anomaly size [pixels x pixels]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimAviris01</td>
<td>35</td>
<td>50</td>
<td>1x1</td>
</tr>
<tr>
<td>SimAviris01</td>
<td>70</td>
<td>50</td>
<td>1x1</td>
</tr>
<tr>
<td>SimAviris01</td>
<td>35</td>
<td>100</td>
<td>2x2</td>
</tr>
<tr>
<td>SimAviris01</td>
<td>70</td>
<td>100</td>
<td>2x2</td>
</tr>
<tr>
<td>SimAviris01</td>
<td>35</td>
<td>150</td>
<td>5x5</td>
</tr>
<tr>
<td>SimAviris01</td>
<td>70</td>
<td>150</td>
<td>5x5</td>
</tr>
<tr>
<td>SimAviris01</td>
<td>35</td>
<td>250</td>
<td>10x10</td>
</tr>
<tr>
<td>SimAviris01</td>
<td>70</td>
<td>250</td>
<td>10x10</td>
</tr>
<tr>
<td>SimAviris01</td>
<td>35</td>
<td>350</td>
<td>15x15</td>
</tr>
<tr>
<td>SimAviris01</td>
<td>70</td>
<td>350</td>
<td>15x15</td>
</tr>
<tr>
<td>SimAviris01</td>
<td>35</td>
<td>450</td>
<td>20x20</td>
</tr>
<tr>
<td>SimAviris01</td>
<td>70</td>
<td>450</td>
<td>20x20</td>
</tr>
<tr>
<td>SimAviris01</td>
<td>35</td>
<td>550</td>
<td>25x25</td>
</tr>
<tr>
<td>SimAviris01</td>
<td>70</td>
<td>550</td>
<td>25x25</td>
</tr>
</tbody>
</table>
3.1. EXPERIMENTS ON SYNTHETIC IMAGES

Since the RX and LRX AD do not build an anomaly map, the author defines anomalous pixels as pixels having a score of $\geq 75\%$ 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 $\text{false}_{-}\text{anomalies}$ and $\text{correctly}_{-}\text{predicted}_{-}\text{anomalies}$.

3.1.1 RX detection results

3.1.1.1 Hsueh-mimicked image

For this test, $\text{false}_{-}\text{anomalies} = 0$, and $\text{correctly}_{-}\text{predicted}_{-}\text{anomalies} = 0.3714$. A score of 0.3714 for $\text{correctly}_{-}\text{predicted}_{-}\text{anomalies}$ 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 Figure 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_AVIRIS* 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

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. EXPERIMENTS ON SYNTHETIC IMAGES

3.1.2 LRX detection results

LRX was tested for different kernel sizes $K=\{5, 10, 15, 20, 23, 25 \text{ and } 30\}$. 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

<table>
<thead>
<tr>
<th>$K$</th>
<th>false_anomalies</th>
<th>correctly_predicted_anomalies</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>967</td>
<td>0.0429</td>
</tr>
<tr>
<td>10</td>
<td>62</td>
<td>0.1143</td>
</tr>
<tr>
<td>15</td>
<td>58</td>
<td>0.1714</td>
</tr>
<tr>
<td>20</td>
<td>50</td>
<td>0.2857</td>
</tr>
<tr>
<td>23</td>
<td>46</td>
<td>0.4782</td>
</tr>
<tr>
<td>25</td>
<td>34</td>
<td>0.5143</td>
</tr>
<tr>
<td>30</td>
<td>38</td>
<td>0.4571</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$K$</th>
<th>false_anomalies</th>
<th>correctly_predicted_anomalies</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>0.33</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.3: LRX detection results on SIM30_30AVIRIS scene.

LRX performs well for the Sim30_30AVIRIS scene, and yields perfect results for $K\geq 10$, as shown in Table 3.3.
CHAPTER 3. REVIEW OF STATE OF THE ART ANOMALY DETECTORS

3.1.2.3 SimAviris01

<table>
<thead>
<tr>
<th>$K$</th>
<th>false_anomalies</th>
<th>correctly_predicted_anomalies</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>703</td>
<td>0.3562</td>
</tr>
<tr>
<td>10</td>
<td>2584</td>
<td>0.5106</td>
</tr>
<tr>
<td>15</td>
<td>2247</td>
<td>0.5621</td>
</tr>
<tr>
<td>20</td>
<td>2167</td>
<td>0.5710</td>
</tr>
<tr>
<td>23</td>
<td>1379</td>
<td>0.6737</td>
</tr>
<tr>
<td>25</td>
<td>1126</td>
<td>0.7192</td>
</tr>
<tr>
<td>30</td>
<td>1056</td>
<td>0.7208</td>
</tr>
</tbody>
</table>

Table 3.4: LRX results on SimAviris01 scene.

Table 3.4 shows that the LRX AD struggles more on the SimAviris01 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 $\tau$ used in the ALRX algorithm was tested in range [0.5, 100]. The values of $\tau$ 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

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>$K$</th>
<th>false_anomalies</th>
<th>correctly_predicted_anomalies</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>5</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>3.5</td>
<td>10</td>
<td>46</td>
<td>0</td>
</tr>
<tr>
<td>90</td>
<td>15</td>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>90</td>
<td>20</td>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>90</td>
<td>23</td>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>90</td>
<td>25</td>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>90</td>
<td>30</td>
<td>0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 3.5: ALRX results on SIM30_30AVIRIS scene.
3.1. EXPERIMENTS ON SYNTHETIC IMAGES

3.1.3.2 SimAviris01

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>$K$</th>
<th>false_anomalies</th>
<th>correctly_predicted_anomalies</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>5</td>
<td>4760</td>
<td>0.3447</td>
</tr>
<tr>
<td>3.5</td>
<td>10</td>
<td>2953</td>
<td>0.1553</td>
</tr>
<tr>
<td>3.5</td>
<td>15</td>
<td>2871</td>
<td>0.1252</td>
</tr>
<tr>
<td>3.5</td>
<td>20</td>
<td>2985</td>
<td>0.0888</td>
</tr>
</tbody>
</table>

Table 3.6: ALRX results on SimAviris01 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 SimAviris01 scene, to be able to do a comparison to the other ADs performances.

3.1.4.1 SIM_AVIRIS_30_30

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>false_anomalies</th>
<th>correctly_predicted_anomalies</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>8</td>
<td>0.33</td>
</tr>
<tr>
<td>0.3</td>
<td>6</td>
<td>0.4</td>
</tr>
<tr>
<td>0.5</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>0.7</td>
<td>2</td>
<td>0.6667</td>
</tr>
<tr>
<td>0.8</td>
<td>1</td>
<td>0.25</td>
</tr>
<tr>
<td>0.9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 SimAviris01

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>false_anomalies</th>
<th>correctly_predicted_anomalies</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>552</td>
<td>1</td>
</tr>
<tr>
<td>0.2</td>
<td>491</td>
<td>1</td>
</tr>
<tr>
<td>0.3</td>
<td>429</td>
<td>1</td>
</tr>
<tr>
<td>0.4</td>
<td>368</td>
<td>1</td>
</tr>
<tr>
<td>0.5</td>
<td>306</td>
<td>0.6727</td>
</tr>
<tr>
<td>0.6</td>
<td>245</td>
<td>0.3457</td>
</tr>
<tr>
<td>0.7</td>
<td>184</td>
<td>0.1685</td>
</tr>
<tr>
<td>0.8</td>
<td>122</td>
<td>0.0631</td>
</tr>
<tr>
<td>0.9</td>
<td>61</td>
<td>0.0420</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.8: ACAD results on SimAviris01 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 $\tau = 0.4$, 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.
3.2. TESTING ON REAL IMAGE DATA

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.

![Image of RX AD result](image)

**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.

![Image of LRX AD result](image)

**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. $\tau$ is set to 250. The anomaly map is overlayed over Figure 3.10.

![Image of ACAD AD result](image)

**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.

<table>
<thead>
<tr>
<th>AD</th>
<th>false_anomalies (best performance)</th>
<th>correctly_predicted_anomalies (best performance)</th>
<th>Performance on real data</th>
<th>Possibility of implementing in real time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX</td>
<td>Hsueh: 0 SIM30_30AVIRIS: 0 Sim_Aviris01: 0</td>
<td>Hsueh: 0.3714 Sim30_30AVIRIS: 1 Sim_Aviris01: 1</td>
<td>Figure 3.11</td>
<td>Low. Need global covariance matrix before computing inverse.</td>
</tr>
<tr>
<td>LRX</td>
<td>Hsueh: 50 SIM30_30_AVIRIS: 0 SimAviris01: 1056</td>
<td>Hsueh: 0.5143 SIM30_30_AVIRIS: 1 SimAviris01: 0.7208</td>
<td>Figure 3.12</td>
<td>Medium. Need to wait for a window of size $K \times K$ to be captured by the imager before processing can start.</td>
</tr>
<tr>
<td>ALRX</td>
<td>Hsueh: - SIM30_30_AVIRIS: 0 SimAviris01: 4760</td>
<td>Hsueh: - SIM30_30_AVIRIS: 0.5 SimAviris01: 0.3447</td>
<td>-</td>
<td>Medium. Need to wait for a window of size $K \times K$ to be captured by the imager before processing can start.</td>
</tr>
<tr>
<td>ACAD</td>
<td>Hsueh: - SIM30_30_AVIRIS: 2 SimAviris01: 368</td>
<td>Hsueh: - SIM30_30_AVIRIS: 0.667 SimAviris01: 1</td>
<td>Figure 3.13</td>
<td>High. Pixels can be processed as soon as they are captured by the imager.</td>
</tr>
</tbody>
</table>

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 $N_{\text{pixels}} \times N_{\text{rows}}$ which may be transmitted, as opposed to the RX and LRX algorithms which produce output results of size $\text{Pixel\_data\_width} \times 2 \times N_{\text{pixels}} \times N_{\text{rows}}$. 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](image-url)
3.2. TESTING ON REAL IMAGE DATA

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 3. REVIEW OF STATE OF THE ART ANOMALY DETECTORS
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, $P_{\text{bands}} = N_{\text{BANDS}}$, depending upon if preprocessing steps such as Principal Component Analysis (PCA) is done on the image cube. $\text{Pixel}_\text{data}_\text{width}$ 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 $P_{\text{bands}}$ and $\text{Pixel}_\text{data}_\text{width}$ make memory usage an important consideration.

4.1.1 Storing and updating matrices in ACAD

The ACAD algorithm requires storage of the following matrices: $\mathbf{R}(x_k)$, $\tilde{\mathbf{R}}(x_k)$ and $\sum_{t_j \in \Delta(k)} t_j t_j^T$. 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 $P_{\text{bands}} \times P_{\text{bands}}$, with matrix elements of size $\text{Pixel}_\text{data}_\text{width} \times 2$. 
CHAPTER 4. PROPOSED HARDWARE IMPLEMENTATION

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

Storing and updating matrices of size $P_{\text{bands}} \times P_{\text{bands}}$ with matrix elements of size $\text{Pixel\_data\_width} \times 2$ require a lot of memory resources. One of the matrices is the causal correlation matrix $R(x_k)$. 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 $R(x_k)$ is the product of $x \times x^T$, the resulting data width will be $2 \times \text{Pixel\_data\_width}$. For $\text{Pixel\_data\_width} = 16$, using spectral information from all $N_{\text{BANDS}}$ would require $P_{\text{bands}} \times P_{\text{bands}} \times 32 = 100 \times 100 \times 32 \text{ bit} = 320 \text{ kbit}$ 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)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Part Number</td>
<td>XC7Z007S</td>
<td>XC7Z012S</td>
<td>XC7Z014S</td>
<td>XC7Z010</td>
<td>XC7Z015</td>
<td>XC7Z020</td>
<td>XC7Z030</td>
<td>XC7Z035</td>
<td>XC7Z045</td>
<td>XC7Z100</td>
</tr>
<tr>
<td>Xilinx 7 Series Programmable Logic Equivalent</td>
<td>Artix-7 FPGA</td>
<td>Artix-7 FPGA</td>
<td>Artix-7 FPGA</td>
<td>Artix-7 FPGA</td>
<td>Artix-7 FPGA</td>
<td>Artix-7 FPGA</td>
<td>Kintex-7 FPGA</td>
<td>Kintex-7 FPGA</td>
<td>Kintex-7 FPGA</td>
<td>Kintex-7 FPGA</td>
</tr>
<tr>
<td>Programmable Logic Cells</td>
<td>23K</td>
<td>55K</td>
<td>65K</td>
<td>28K</td>
<td>74K</td>
<td>85K</td>
<td>125K</td>
<td>275K</td>
<td>350K</td>
<td>444K</td>
</tr>
<tr>
<td>Look-Up Tables (LUTs)</td>
<td>14,400</td>
<td>34,400</td>
<td>40,600</td>
<td>17,600</td>
<td>46,200</td>
<td>53,200</td>
<td>78,600</td>
<td>171,900</td>
<td>218,600</td>
<td>277,400</td>
</tr>
<tr>
<td>Flip-Flops</td>
<td>28,800</td>
<td>68,800</td>
<td>81,200</td>
<td>35,200</td>
<td>92,400</td>
<td>106,400</td>
<td>157,200</td>
<td>343,800</td>
<td>457,200</td>
<td>554,800</td>
</tr>
<tr>
<td>Block RAM (# 36 Kb Blocks)</td>
<td>1.8 Mb (50)</td>
<td>2.5 Mb (72)</td>
<td>3.8 Mb (107)</td>
<td>2.1 Mb (60)</td>
<td>3.3 Mb (95)</td>
<td>4.9 Mb (140)</td>
<td>9.3 Mb (265)</td>
<td>17.4 Mb (500)</td>
<td>19.2 Mb (545)</td>
<td>26.5 Mb (755)</td>
</tr>
<tr>
<td>DSP Slices (16x25 MACCs)</td>
<td>66</td>
<td>120</td>
<td>170</td>
<td>80</td>
<td>160</td>
<td>220</td>
<td>400</td>
<td>900</td>
<td>900</td>
<td>2,020</td>
</tr>
<tr>
<td>Peak DSP Performance (Symmetric FIR)</td>
<td>73 GMACs</td>
<td>131 GMACs</td>
<td>187 GMACs</td>
<td>100 GMACs</td>
<td>200 GMACs</td>
<td>276 GMACs</td>
<td>593 GMACs</td>
<td>1,334 GMACs</td>
<td>1,334 GMACs</td>
<td>2,622 GMACs</td>
</tr>
</tbody>
</table>

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

$$max_{\text{bands}} = \sqrt{\frac{\text{number of registers}}{2 \times \text{Pixel\_data\_width} \times \text{number of matrices}}}$$

(4.1)
it is possible to do an estimation of the maximum value of $P_{\text{bands}}$ if using registers for storage of the matrices. Parameter \textit{number_of_registers} is the total number of registers (flip flops) available in the device. \textit{number_of_matrices} is the number of matrices of size $P_{\text{bands}} \times P_{\text{bands}}$ with matrix elements of size $\text{Pixel}_\text{data}_\text{width} \times 2$ that need to be stored in memory. $\text{max_bands}$ is the maximum number of $P_{\text{bands}}$ for the matrices used in ACAD. As the ACAD algorithm needs to store five matrices, \textit{number_of_matrices} = 5, which yields $\text{max_bands} = [31, 46]$ 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 $P_{\text{bands}}$ from 100 to 12 or 17 can be done through pre-processing of the data by for example PCA. The benefit of using registers to store matrices is the ability of instantaneous update.

\subsection*{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 \cite{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 \ref{listing:4.1} it is possible to get two writes and two reads per cycle per 36kbit BRAM, with separate read and write addresses.

\begin{lstlisting}[language=VHDL]
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;
  data_out   : out std_logic_vector(B_RAM_BIT_WIDTH-1 downto 0)
);
end block_ram;
architecture Behavioral of block_ram is
end;
\end{lstlisting}

\textbf{Listing 4.1:} Code for inferring a SDP 18 kbit BRAM.
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 \(P_{\text{bands}} \times P_{\text{bands}}\) with matrix elements of size \(\text{Pixel\_data\_width} \times 2\) in BRAM, with acceptable update characteristics, the time spent updating \(\tilde{R}(x_k)\) has been used as a benchmark. Equation 4.2 shows the calculation of number of clock cycles needed to update \(\tilde{R}(x_k)\) of size \(P_{\text{bands}} \times P_{\text{bands}}\), \(n_{\text{clk\_update\_corr\_BRAM}}\):

\[
 n_{\text{clk\_update\_corr\_BRAM}} = \frac{P_{\text{bands}} \times P_{\text{bands}}}{2 \times N_{\text{bram\_correlation}}} .
\] (4.2)

The update is done for each pixel in the image. \(N_{\text{bram\_correlation}}\) is the number of 36 kbit BRAMs used to store \(\tilde{R}(x_k)\). The total time spent updating \(\tilde{R}(x_k)\) for the entire image is given by equation 4.3:

\[
 \text{clk\_corr\_image\_BRAM} = N_{\text{pixels}} \times N_{\text{rows}} \times n_{\text{clk\_update\_corr\_BRAM}}.
\] (4.3)

Updating a matrix with \(P_{\text{bands}} = 100\) using nine BRAMs would require 556 clock cycles. For the entire image, having \(N_{\text{pixels}} = 578\) and \(N_{\text{rows}} = 1088\), 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 \(\tilde{R}(x_k)\) for an image of \(N_{\text{rows}} = 1088 \times N_{\text{pixels}} = 578\), with a target clock frequency of 100 MHz, plotted as a function of \(N_{\text{bram\_correlation}}\). The BRAMs are assumed written to in parallel. Figure 4.3 is plotted for \(P_{\text{bands}} = [20, 30, 40, 50, 60, 70, 80, 90, 100]\).
As shown in Figure 4.3, the time spent updating $\tilde{\mathbf{R}}(\mathbf{x}_k)$ can be reduced by increasing $N_{\text{bram\_correlation}}$. One column of $\mathbf{R}(\mathbf{x}_k)$ will maximum contain $100 \times 32 = 3200$ bit of data. By setting $N_{\text{bram\_correlation}} = P_{\text{bands}}$ it is possible to store one column of the correlation matrix in each BRAM, and enable writing of $P_{\text{bands}}$ 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 $\frac{P_{\text{bands}}}{2}$.

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.

\begin{verbatim}
row [i] = row[j];
row[j] = row[i]; // This operation is done in parallel with
the previous one
\end{verbatim}

The operations shown in Figure 4.4 need to swap two rows of $\mathbf{A}$. By choosing to have $P_{\text{bands}}$ number of 36 kbit BRAMs for storage of $\mathbf{A}$ and $\mathbf{A}^{-1}$ 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 $P_{\text{bands}}$ to store each of the five matrices, this means maximum $P_{\text{bands}}$ will be 53 for the Z7030 and 100 for the Z7035.

$\tilde{R}(x_k)$ is written to $N_{\text{bram\_correlation}}$ in parallel, where the leftmost column (column zero) of $\tilde{R}(x_k)$ is written to BRAM_0, column one to BRAM_1, ... and column $P_{\text{bands}}$-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 $R(x_k)$, $\sum_{t_j \in \Delta(k)} t_j t_j^T$, A and $A^{-1}$ as for $\tilde{R}(x_k)$, 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.
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 $R(x_k)$, $\sum_{t_j \in \Delta(k)} t_j t_j^T$, $A$, $A^{-1}$ and $\tilde{R}(x_k)$ are all stored in BRAM. The ACAD anomaly detector interfaces the Cube DMA via an AXI-Stream interface. The output of the ACAD anomaly detector, anomaly map, is a binary anomaly map.
CHAPTER 4. PROPOSED HARDWARE IMPLEMENTATION

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

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 $\hat{R}(x_k)$. After $\hat{R}(x_k)$ is computed, it is sent to **ACAD inverse**. Two rows of $\hat{R}(x_k)$ are outputted per clock cycle to **ACAD inverse**, which computes $\hat{R}^{-1}(x_k)$.

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

$$
\delta^{ACAD}(x_k) = x_k^T \hat{R}^{-1}(x_k)x_k.
$$

(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 $P_{bands} \times \text{Pixel data width} \times 2$. **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.](image)

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 × 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.](image)

### 4.4 ACAD correlation

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

\[
R(x_k) = \frac{1}{k} \sum_{i=1}^{k} x_i x_i^T.
\] (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.
The dotted squares mark one correlation sub-module. One correlation sub-module computes two elements of $\tilde{R}(x_k)$. $P\_\text{bands}$ correlation sub-modules are synthesized in the design. Input signal $\text{din}$ is a $P\_\text{bands} \times \text{Pixel\_data\_width}$ wide bus. It is sent from the \text{Shiftregister} block and contains data from one pixel vector. Signal $b\_\text{shift\_k}$ is sent from \text{FSM ACAD} and indicates the number of shift operations to be done by the \text{Normalize by k} blocks.

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

![Figure 4.10: Data flow within the ACAD correlation module.](image)
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 \( P_{\text{bands}} \times \text{Pixel\_data\_width} \times 2 \times 2 \) wide bus containing two rows of \( \tilde{R}(x_k) \).

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 \( P_{\text{bands}} = 4 \). \( din' \) is the transposed vector of \( din \).

![Figure 4.11: An example of the data handling done by ACAD correlation. For this example \( P_{\text{bands}} = 4 \).](image)

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 \( \text{Pixel\_data\_width} \). \( a_{01} \) in this cycle is \( \text{din[Pixel\_data\_width-1:0]} \) and \( b_{01} \) is \( \text{din[Pixel\_data\_width \times 2 -1 : Pixel\_data\_width]} \). Each of the sub-modules shown in Figure 4.10 computes two elements of \( \tilde{R}(x_k) \) before inputting the results to BRAMs. Sub-module 0 computes column zero, which is written to \( \text{BRAM\_0\_36\_kbit} \), sub-module 1 computes column one, which is written to \( \text{BRAM\_1\_36\_kbit} \), ..., while sub-module \( P_{\text{bands}-1\_36\_kbit} \) computes column \( P_{\text{bands}-1} \), which is written to \( \text{BRAM\_P_{\text{bands}}-1\_36\_kbit} \). Figure 4.11 illustrates these operations. The \( rhs \) operation is a right shift operation by \( b_{\text{shift\_k}} \) 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 \( \text{din[Pixel\_data\_width \times 3-1: Pixel\_data\_width \times 2]} \) and \( b_{01} \) is \( \text{din[Pixel\_data\_width \times 4-1: Pixel\_data\_width]} \).
4.5. INVERSE COMPUTATION

4.4.1 Normalizing with k

The **Normalizing with k** block approximates the operation of multiplying $\sum_{i=1}^{k} x_i x_i^T$ with $\frac{1}{k}$, where $k = [1, 2, 3,..., N_{\text{PIXELS}_{\text{TOT}}}]$ is the index of the pixel currently being processed. $N_{\text{PIXELS}_{\text{TOT}}}$ is the total amount of pixels in the hyperspectral image, $N_{\text{PIXELS}_{\text{TOT}}} = N_{\text{rows}} \times N_{\text{pixels}}$. 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 $b_{\text{shift}_{\text{k}}}$ 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 $b_{\text{shift}_{\text{k}}}$ array$(k)$ in LUTs. $b_{\text{shift}_{\text{k}}}$ array$(k)$ contains the best shifting approximation for pixel indexes $k = [1, 2, 3,..., N_{\text{PIXELS}_{\text{TOT}}}]$. The array can be generated by the MATLAB-script shown in Listing 4.2.

**Listing 4.2**: Code for creating the array $b_{\text{shift}_{\text{k}}}$ array.

```matlab
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

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\textsuperscript{inverse} is presented in Figure 4.12. This is an implementation of the Gauss-Jordan elimination shown in Figure 2.10. ACAD\textsuperscript{inverse} interfaces ACAD\textsuperscript{correlation}. The outputs from ACAD\textsuperscript{inverse} are sent to dACAD.
Figure 4.12: Top level architecture of the inverse module.
CHAPTER 4. PROPOSED HARDWARE IMPLEMENTATION

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 $A^{-1}_{\text{inv}}$ are two arrays of BRAM36 of size $P_{\text{bands}}$, in which $A$ and $A^{-1}$ are stored.

$$
A[\text{row}[j]] = A[\text{row}[j]] - A[\text{row}[i]] \times (A[\text{row}[j]][i]/A[\text{row}[i]][i]);
$$

$$
A^{-1}[\text{row}[j]] = A^{-1}[\text{row}[j]] - A^{-1}[\text{row}[i]] \times (A[\text{row}[j]][i]/A[\text{row}[i]][i]); \quad \text{// This operation is done in parallel with the previous one}
$$

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** $\geq$ 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.
4.5. INVERSE COMPUTATION

The output signal \( \text{data}_{\text{out}} \) consists of new matrix data for matrices \( A \) and \( A^{-1} \), stored in \( A \) and \( A_{\text{inv}} \), for index \( j \). The bus \( \text{control}_{\text{BRAM}} \) contains control signals to \( A \) and \( A_{\text{inv}} \). The control signals include read and write addresses and write and read-enabling signals. **Elimination core** contains \( P \_\text{bands} \) sub-modules marked by the dotted squares in Figure 4.15. One such sub-module computes one element of \( \text{new}_{\text{row}}_j \) and one element of \( \text{new}_{\text{inv}}_{\text{row}}_j \). \( \text{new}_{\text{row}}_j \) and \( \text{new}_{\text{inv}}_{\text{row}}_j \) are updated row data for the rows indexed by index \( j \) in matrices \( A \) and \( A^{-1} \), which are written to \( A \) and \( A_{\text{inv}} \) respectively. As shown in Figure 4.15 Submodule(0) computes \( \text{new}_{\text{row}}_j(0) \) and \( \text{new}_{\text{inv}}_{\text{row}}_j(0) \), Submodule(1) computes \( \text{new}_{\text{row}}_j(1) \) and \( \text{new}_{\text{inv}}_{\text{row}}_j(1) \),..., and Submodule\((P\_\text{bands} - 1)\) computes \( \text{new}_{\text{row}}_j(P\_\text{bands} - 1) \) and \( \text{new}_{\text{inv}}_{\text{row}}_j(P\_\text{bands} - 1) \). As such, the operations shown in Figure 4.13 are implemented.

Figure 4.14: Elimination core part one.
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.
4.5. INVERSE COMPUTATION

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown state</td>
<td>An unknown state. The behaviour of the ACAD inverse is unknown. The FSM should transition to state Idle. The ACAD inverse is not performing any operations.</td>
</tr>
<tr>
<td>Idle</td>
<td>Writing data inputted from ACAD correlation to A. Two rows are written to BRAMs per clock cycle.</td>
</tr>
<tr>
<td>Forward_elimination</td>
<td>Outputting the completed inverse matrix for the pixel. Two rows are outputted per clock cycle.</td>
</tr>
<tr>
<td>Backward_elimination</td>
<td></td>
</tr>
<tr>
<td>Last_division</td>
<td></td>
</tr>
<tr>
<td>Output_inverse_matrix</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: States of the inverse FSM.

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 \( P_{\text{bands}} \times P_{\text{bands}} \) 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 = \( \frac{P_{\text{bands}}}{2} \), all \( P_{\text{bands}} \) elements of the different BRAMs have been written, and the entire causal anomaly-removed sample spectral correlation matrix

Figure 4.16: FSM controlling ACAD inverse shown in Figure 4.12.
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 $A_{\text{inv}}$. 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.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unknown state</strong></td>
<td>An unknown state. The behaviour of <strong>Forward elimination</strong> is unknown. The FSM should transition to state <strong>Idle</strong>.</td>
</tr>
<tr>
<td><strong>Idle</strong></td>
<td><strong>Forward elimination</strong> is not performing any operations.</td>
</tr>
<tr>
<td><strong>Check_diagonal_element_is_zero</strong></td>
<td>Checking if element row$_i$[index$_i$] = 0 as done in Gauss-Jordan elimination.</td>
</tr>
<tr>
<td><strong>Swap_rows</strong></td>
<td>Swapping row$_i$ and row$_j$ of $A$ stored in $A$.</td>
</tr>
<tr>
<td><strong>Even_j_write</strong></td>
<td>Updating an even indexed row of $A$ and $A^{-1}$.</td>
</tr>
<tr>
<td><strong>Odd_j_write</strong></td>
<td>Updating an odd indexed row of $A$ and $A^{-1}$.</td>
</tr>
</tbody>
</table>

**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.
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.

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

**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 $B_{\text{RAM\_wait\_clk\_cycles}}$ 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.

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 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.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown state</td>
<td>An unknown state. The behaviour of the Backward elimination is unknown.</td>
</tr>
</tbody>
</table>
<pre><code>                  | The FSM should transition to state Idle.                                    |
</code></pre>
<p>| Idle           | <strong>Backward elimination</strong> is not performing any operations.                   |
| First_elimination | Doing the first backward elimination iteration in the inverse computation. |
| Odd_i_start  | Starting at a new iteration of the outermost loop of the backward elimination loop in the Gauss-Jordan elimination. |
| Even_i_start  | Starting at a new iteration of the outermost loop of the backward elimination loop in the Gauss-Jordan elimination. |
| 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} ).                    |</p>

*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.
4.5. INVERSE COMPUTATION

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.

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.
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.
4.5. INVERSE COMPUTATION

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown state</td>
<td>An unknown state. The behaviour of Last division is unknown.</td>
</tr>
<tr>
<td>Idle</td>
<td>Last division is not performing any operations.</td>
</tr>
<tr>
<td>Even_i_write</td>
<td>Updating an even indexed row of $A^{-1}$.</td>
</tr>
<tr>
<td>Odd_i_write</td>
<td>Updating an odd indexed row of $A^{-1}$.</td>
</tr>
</tbody>
</table>

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.

![ FSM diagram controlling Last division ](image)

**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 $P_{bands}$ 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.
CHAPTER 4. PROPOSED HARDWARE IMPLEMENTATION

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.
4.5. INVERSE COMPUTATION

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.

\[
\text{inv}_{\text{worst\_case}} = \sum_{i=0}^{N_{\text{STATES\_INV}}} \text{worst\_case\_ex\_state}(i)
\]

\[
= \frac{P_{\text{bands}}}{2} + \sum_{i=0}^{P_{\text{bands}}-1} i + P_{\text{bands}}
\]

\[
\text{STORE\_CORRELATION\_MATRIX}
\]

\[
+ \sum_{i=0}^{P_{\text{bands}}-1} i
\]

\[
\text{STATE\_FORWARD\_ELIMINATION}
\]

\[
+ \frac{P_{\text{bands}}}{2} + \sum_{i=0}^{P_{\text{bands}}-1} i
\]

\[
\text{LAST\_DIVISION}
\]

\[
\text{STATE\_BACKWARD\_ELIMINATION}
\]

\[
= 3P_{\text{bands}} + 2 \sum_{i=0}^{P_{\text{bands}}-1} i
\]

\[
(4.6)
\]

\[
\text{inv}_{\text{best\_case}} = \sum_{i=0}^{N_{\text{STATES\_INV}}} \text{best\_case\_ex\_state}(i)
\]

\[
= \frac{P_{\text{bands}}}{2} + \sum_{i=0}^{P_{\text{bands}}-1} i
\]

\[
\text{STORE\_CORRELATION\_MATRIX}
\]

\[
+ \sum_{i=0}^{P_{\text{bands}}-1} i
\]

\[
\text{STATE\_FORWARD\_ELIMINATION}
\]

\[
+ \frac{P_{\text{bands}}}{2} + \sum_{i=0}^{P_{\text{bands}}-1} i
\]

\[
\text{LAST\_DIVISION}
\]

\[
\text{STATE\_BACKWARD\_ELIMINATION}
\]

\[
= 2P_{\text{bands}} + 2 \sum_{i=0}^{P_{\text{bands}}-1} i
\]

\[
(4.7)
\]

Figure 4.31 shows the estimated execution time in seconds for computing \(\hat{R}^{-1}(x_k)\) for all \(x_k\) in the hyperspectral image, for an image size of 1088x576, with an operating clock frequency of 100MHz.
4.5. INVERSE COMPUTATION

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 $C = B \times \frac{1}{a}$ where $C, B$ and $a$ are integers in the range of $s=[-2^{\text{Pixel\_data\_width\_width\_x\_2\_1}}, ..., 2^{\text{Pixel\_data\_width\_x\_2\_1}}]$.

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, \( f_{\text{max}} \), is calculated by equation 4.8:

\[
f_{\text{max}} = \frac{1}{-WNS + 10ns}.
\]  

(4.8)

Results for different divisor-and-dividend 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.

![Dataflow of block Last division using the division operator "/" for division.](image)

**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, \( \text{Pixel}_\text{Data}\_\text{Width} \times 2 - 1 \) numbers of shift-operation processes shifts the unsigned divisor by \( n_{\text{shifts}} = [1, 2, ... \text{Pixel}_\text{Data}\_\text{Width} \times 2 - 1] \). These shift-operation processes are illustrated in Figure 4.33 by the blocks Right shift one, Right shift two and Right shift \( \text{PIXEL\_DATA\_WIDTH}^*2-1 \). 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 \( \text{inv}\_\text{row}\_\text{i} \) 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.
4.5. INVERSE COMPUTATION

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 \( \text{divisor\_inv}^{2^{\text{Div\_Precision}}} \), where \( a = [1, 2, \ldots, 2^{\text{Div\_Precision}}] \). \( \text{Div\_Precision} \) is the bit width of the divisor possible to represent with this approach. If the MSB of the divisor \( a \) is at an index \( > \text{Div\_Precision} \), the adaptive shifting approach is used. If not, \( a \) is used as an index to look up in LUTs storing \( \text{divisor\_inv} \). Then, \( \text{divisor\_inv}(a) \) is multiplied by \( B \), which yields product \( C \). \( C \) is right shifted \( \text{Div\_Precision} \) spaces. This can be seen in equation 4.9.

\[
C = \text{shift\_right}(B \times \text{divisor\_inv}(a), \text{Div\_Precision})
\] (4.9)

The code for inferring LUTs for storage of \( \text{divisor\_inv} \) is shown in Listing 4.3, exemplified for \( \text{Div\_Precision}=4 \).

Listing 4.3: LUT division approach exemplified for \( \text{Div\_Precision}=4 \).

```javac
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_division_lut is array (0 to 2**C_NY-1) of integer range 0 to 2**C_NDY-1;
```

```
CHAPTER 4. PROPOSED HARDWARE IMPLEMENTATION

constant C_DIV_LUT : t_division_lut := (
    16,
    8,
    5,
    4,
    3,
    3,
    2,
    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.

```matlab
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 ,
', 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.
4.5. INVERSE COMPUTATION

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 \(\text{Pixel}_\text{data}_\text{width}\) 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 \(P\text{bands} \geq 20\) and \(\text{Pixel}_\text{data}_\text{width} = 16\), 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 \(P\text{bands} \geq 20\) and \(\text{Pixel}_\text{data}_\text{width} = 16\), 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 \(P\text{bands} \geq 60\) and \(\text{Pixel}_\text{data}_\text{width} = 16\). Therefore, ACAD correlation is synthesized for xc7k160tiffv676-2L for \(P\text{bands} \geq 60\) and \(\text{Pixel}_\text{data}_\text{width} = 16\).

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 $P_{\text{bands}} = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]$. The numbers of synthesized Slice registers and Slice LUTs are shown in Figure 5.1 plotted as a function of $P_{\text{bands}}$.

![Shiftregister synthesis results](image)

Figure 5.1: Shiftregister synthesis results.

5.1.2 ACAD correlation

The design was synthesized for $P_{\text{bands}} = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]$. 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 $P_{\text{bands}}$.

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.
CHAPTER 5. RESULTS

Figure 5.3: Number of synthesized BRAM36E1 and DSP48E1 as a function of $P_{\text{bands}}$ for the ACAD correlation block.

Figure 5.4: Number of synthesized Slice Registers and Slice LUTs as a function of $P_{\text{bands}}$ for the ACAD correlation block.
5.1. SYNTHESIS

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\_width = 10.](image)

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.
CHAPTER 5. RESULTS

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

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

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 $P_{\text{bands}} = [10, 20, 30, 40, 50]$ and $\text{Pixel}_{\text{data width}} = 16$ on the Zedboard. The timing results are presented in Table 5.1.

<table>
<thead>
<tr>
<th>$P_{\text{bands}}$</th>
<th>WNS [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.272</td>
</tr>
<tr>
<td>20</td>
<td>0.721</td>
</tr>
<tr>
<td>30</td>
<td>1.446</td>
</tr>
<tr>
<td>40</td>
<td>0.845</td>
</tr>
<tr>
<td>50</td>
<td>0.509</td>
</tr>
</tbody>
</table>

Table 5.1: Timing results for ACAD correlation $\text{Pixel}_{\text{data width}} = 16$.

ACAD correlation was synthesized for $P_{\text{bands}} = [10, 20, 30, 40, 50, 60, 70, 80, 90]$ and $\text{Pixel}_{\text{data width}} = 10$ on the Zedboard. The timing results are presented in Table 5.2.

<table>
<thead>
<tr>
<th>$P_{\text{bands}}$</th>
<th>WNS [ns]</th>
<th>Net delay [ns]</th>
<th>Logic delay [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>-3.074</td>
<td>7.860</td>
<td>5.078</td>
</tr>
<tr>
<td>20</td>
<td>-3.309</td>
<td>7.868</td>
<td>5.305</td>
</tr>
<tr>
<td>40</td>
<td>-3.319</td>
<td>7.698</td>
<td>5.485</td>
</tr>
<tr>
<td>50</td>
<td>-3.324</td>
<td>7.703</td>
<td>5.485</td>
</tr>
<tr>
<td>60</td>
<td>-4.923</td>
<td>8.563</td>
<td>6.224</td>
</tr>
<tr>
<td>70</td>
<td>-4.881</td>
<td>8.521</td>
<td>6.224</td>
</tr>
<tr>
<td>80</td>
<td>-5.224</td>
<td>9.373</td>
<td>5.735</td>
</tr>
</tbody>
</table>

Table 5.2: Timing results for ACAD correlation $\text{Pixel}_{\text{data width}} = 10$.

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 $C = B * \frac{1}{4}$. Width of $B$ is 32 bit. The design was synthesized for dividend and divisor data width of 32, 16, 12 and 10, with $P_{\text{bands}} = 10$ and a target clock constraint of 100 MHz. The target device was the Zedboard.
### 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/srcs/sim_1/new](https://github.com/marthauk/Anomaly-detection/tree/invert_matrix_computation/FPGA_implementation/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 \( \text{modulo}(P\_\text{bands}, 4) = 0 \). Figure 5.10 shows a simulation for \( P\_\text{bands} = 12 \).

\( dout \) is the output signal and outputs a spectral pixel vector of size \( P\_\text{bands} \times \text{Pixel\_data\_width} \). *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.

### 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 \( P\_\text{bands} = [10, 20, 30, 40, 50, 60] \). The worst WNS was 2.034 nanoseconds (ns), which yields \( f_{\text{max}} \) of \( \approx 125 \text{MHz} \).

### 5.1.4.4 Worst Negative Slack LUT approach

The **Inverse** block was synthesized for Zedboard, with \( \text{Div\_Precision} = 17 \) and \( P\_\text{bands} = 10 \). The synthesis results yielded a WNS of -5.972 ns. Net delay accounts for 4.847 of this.

\( P\_\text{bands} = 10 \) was chosen for testing of WNS due to the LUT approach over-utilizing DSP48E1s in the Zedboard for a higher number of \( P\_\text{bands} \).

<table>
<thead>
<tr>
<th>Data width divisor and dividend</th>
<th>WNS [ns]</th>
<th>( f_{\text{max}} ) [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>-80.524</td>
<td>11.046</td>
</tr>
<tr>
<td>16</td>
<td>-11.776</td>
<td>45.934</td>
</tr>
<tr>
<td>12</td>
<td>-7.071</td>
<td>58.578</td>
</tr>
<tr>
<td>10</td>
<td>-5.730</td>
<td>63.572</td>
</tr>
</tbody>
</table>

*Table 5.3:* Synthesis results for Zedboard for **Last division** using the division operator "/".
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 $P_{\text{bands}} = 4$, $P_{\text{pixel \_data \_width}} = 16$.

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. $\text{valid}$ is an input signal signalizing if the input $\text{din}$ is valid. The output $\text{data\_out}$ 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 $\text{data\_out}$, 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 $P_{\text{bands}} = [4,6]$ and $\text{Pixel\_data\_width} = 16$, with constrained random input. Figure [5.13] shows a simulation with $P_{\text{bands}} = 4$. For this simulation the division operator $/\phantom{0}$/ is utilized. Data input signal $\text{din}$ is a $P_{\text{bands}} \times \text{Pixel\_data\_width} \times 2$ wide signal. The output $\text{inverse\_rows}$ is a $P_{\text{bands}} \times 2 \times \text{Pixel\_data\_width} \times 2$ wide signal which outputs two rows of the inverse matrix $A^{-1}$, stored in $A_{\text{inv}}$, per clock cycle, while the signals $\text{data\_out\_brams\_M\_inv}$ and $\text{data\_out\_brams\_M}$ are data that are read from $A_{\text{inv}}$ and $A$ respectively. The signal $\text{state}$ 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 $\text{Pixel\_data\_width} = 16$

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

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 $\text{Div\_Precision} = 17$. As ACAD correlation infers $P\_bands \times 4$ DSPs for $\text{Pixel\_data\_width} \geq 11$, the total number of DSPs synthesized for these two modules for $P\_bands = 30$, $\text{Pixel\_data\_width} = 16$ and $\text{Div\_Precision} = 17$ will be 660. This is a high number, especially considering that $\text{dACAD}$ computes $\delta^{\text{ACAD}}(x_k) = x_k^T \tilde{R}^{-1}(x_k)x_k$, where $x_k$ is a pixel vector of size $P\_bands \times \text{Pixel\_data\_width}$ and $\tilde{R}^{-1}(x_k)$ is a matrix of size $P\_bands \times P\_bands \times \text{Pixel\_data\_width} \times 2$. 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 $\approx 20$. It will be $\approx 40$ 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 $\approx 10$ for the Zedboard.
6.1.2 \textit{Pixel\_data\_width} = 10

When synthesizing \textbf{ACAD correlation} for \textit{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 \textit{P\_bands}.

6.2 Timing results

6.2.1 ACAD correlation

\textbf{ACAD correlation} meets timing demands for \textit{Pixel\_data\_width} = 16. For \textit{Pixel\_data\_width} = 10, the synthesized design infers no DSPs, but maps the logic to LUTs. The WNS of the design is negative for \textit{Pixel\_data\_width} = 10 as can be seen in Table 5.2, meaning \textbf{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 \textit{P\_bands}. This is due to the output ports of \textbf{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 \textbf{ACAD correlation} are connected to \textbf{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 \textbf{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 \textit{Div\_Precision}. The synthesis results for the \textbf{ACAD inverse} block when using LUT approach with \textit{Div\_Precision} = 17 yielded a WNS of -5.972ns, in which 4.847 of this is net delay. As the outputs of \textbf{ACAD inverse} is mapped to output pins when running synthesis using \textbf{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 \textbf{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, $inv_{\text{worst\_case}}$ and $inv_{\text{best\_case}}$, 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.
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. 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 $\delta^{ACAD}(x_k)$ of size $N_{TOT\_PIXELS} \times \text{Pixel\_data\_width} \times 2$, where $\text{Pixel\_data\_width}$ 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 $53 \times 53$ and $100 \times 100$ respectively. These matrices have matrix elements of size $\text{Pixel\_data\_width} \times 2$.

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 $\text{Pixel\_data\_width}$ of 16. For this data width, the correlation module, ACAD correlation, infers $P_{\_bands} \times 4$ DSPs. By setting $\text{Pixel\_data\_width} = 10$, 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 $\tau$. An option is to set $\tau$ 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 $\tau$ 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


Appendices
A.1 High level models of algorithms

A.1.1 Gauss-Jordan elimination

Listing A.1: Gauss Jordan inverse

```matlab
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,i)~=0)
                    % The operations below will be different in hardware, because
                    % of parallell operations
                    %temp_i = row(i);
```
%row(i) = row(j);
%row(j) = temp_i;

temp_i = A(i,:);
A(i,:) = A(j,:);
A(j,:) = temp_i;

end
end
e
if (A(i,i) ==0)
% error('Matrix is singular ');
end
for (j = i +1:1: size_p)
% The operations below will be different in hardware, because
% of parallel operations
A_j_i_temp =A(j,i);
A_i_i_temp = A(i,i);
%A(:, :) = A(:, :) - A(i,:)*A_j_i_temp/A_i_i_temp;
%A_inv(:, :) = A_inv(:, :) = A_inv(i,:)*A_j_i_temp/A_i_i_temp;
for (l= 1: size_p)
A(j,l) = A(j,l) - A(i,l)*A_j_i_temp/A_i_i_temp;
A_inv(j,l) = A_inv(j,l) - A_inv(i,l)*A_j_i_temp/A_i_i_temp;
end
end

if (strcmp(mode, 'forward '))
A_mode_elim = A;
A_mode_elim_inv = A_inv;
end

% Backward elimination to build a diagonal matrix
if(strcmp(mode, 'backward ') | strcmp(mode, 'all '))
for (i=size_p:-1:2)
for (j=i-1:-1: 1)
% The operations below will be different in hardware, because
% of parallel operations
A_j_i_temp =A(j,i);
A_i_i_temp = A(i,i);
%A(:, :) = A(:, :) - A(i,:)*cast(cast(A(j,i)/A(i,i), 'int32 '), 'double ');
%A_inv(:, :) = A_inv(:, :) = A_inv(i,:)*cast(cast(A_j_i_temp/A_i_i_temp, 'int32 '), 'double ');
end
end
endif
A.1. HIGH LEVEL MODELS OF ALGORITHMS

\[
\%A(j,:) = A(j,:) - A(i,:) \ast A(j,i)/A(i,i);
\]
\[
\%A_{\text{inv}}(j,:) = A_{\text{inv}}(j,:) - A_{\text{inv}}(i,:) \ast A_{j_i\text{temp}}/A_{i_i\text{temp}};
\]

\[
\text{for} \ (k=1:\text{size}_p)
\]
\[
A(j,k) = A(j,k) - A(i,k) \ast A(j,i)/A(i,i);
\]
\[
A_{\text{inv}}(j,k) = A_{\text{inv}}(j,k) - A_{\text{inv}}(i,k) \ast A_{j_i\text{temp}}/A_{i_i\text{temp}};
\]

\[
\text{end}
\]

\[
\text{end}
\]

\[
\text{end}
\]

\[
\text{end}
\]

\[
\text{if} \ (\text{strcmp}(\text{mode}, '\text{backward}')\))
\]
\[
A_{\text{mode\_elim\_inv}} = A_{\text{inv}};
\]
\[
A_{\text{mode\_elim}} = A;
\]

\[
\text{end}
\]

\[
\text{if} \ (\text{strcmp}(\text{mode}, '\text{identity}')\))
\]
\[
A_{\text{mode\_elim\_inv}} = \text{zeros}(3);
\]
\[
A_{\text{mode\_elim}} = \text{zeros}(3);
\]

\[
\text{end}
\]

\[
\text{if} \ (\text{strcmp}(\text{mode}, '\text{all}')\))
\]
\[
A_{\text{mode\_elim\_inv}} = \text{zeros}(3);
\]
\[
A_{\text{mode\_elim}} = \text{zeros}(3);
\]

\[
\text{end}
\]

\[
\% \text{Last division to build an identity matrix}
\]
\[
\text{for} \ (i = 1:+1:\text{size}_p)
\]
\[
A_{\text{inv}}(i,:) = A_{\text{inv}}(i,:) \ast 1/A(i,i);
\]

\[
\text{end}
\]

\[
\text{end}
\]

A.1.2 RX anomaly detector

Listing A.2: RX AD

\begin{verbatim}
function \[ result , \sigma , \sigma_{\text{Inv}} \] = hyperRxDetector(M)

%HYPERRX RX anomaly detector

% Usage
% \[ result \] = hyperRxDetector(M)

% Inputs
% M = 2D data matrix (p x N)
\end{verbatim}
% Outputs
% result – Detector output (1 x N)
% sigma – Covariance matrix (p x p)
% sigmaInv – Inverse of covariance matrix (p x p)

% Remove the data mean
[p, N] = size(M);
mMean = mean(M, 2);
M = M - repmat(mMean, 1, N);

% Compute covariance matrix
sigma = hyperCov(M);
sigmaInv = inv(sigma);

result = zeros(N, 1);
for i = 1:N
    result(i) = M(:, i)' * sigmaInv * M(:, i);
end
result = abs(result);
return;

A.1.3 LRX anomaly detector

Listing A.3: LRX AD
function [result, autocorr, sigmaInv] = hyperLRxDetectorCorr(M, K)
% HYPERRX LRX anomaly detector
% hyperLRxDetector performs the Local RX anomaly detector using Correlation
% instead of covariance
% Usage
% [result] = hyperRxDetector(M)
% Inputs
% M – 2D data matrix (p x N)
% K – Size of the kernel window, K x K
% Outputs
% result – Detector output (1 x N)
% sigma – Correlation matrix (p x p)
% sigmaInv – Inverse of correlation matrix (p x p)
[p, N] = size(M);

% Compute correlation matrix of size K
% correlation matrix will be of size p x p
result = zeros(N, 1);
A.1. HIGH LEVEL MODELS OF ALGORITHMS

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    waitbar(j/N,h, 'Updated LRX progress');
30 end
31
32 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,threshold)
2    % LRX anomaly detector, that also removes the detected anomalous targets
3    % hyperLRxDetector performs the Local RX anomaly detector using Correlation
4    % instead of covariance
5    % Usage
6    % [result] = hyperRxDetector(M)
7    % Inputs
8    % M 2D data matrix (p x N)
9    % K Size of the kernel window, K x K
10   % Outputs
11   % result Detector output (1 x N)
12   [p, N] = size(M);
13   h = waitbar(0, 'Initializing waitbar ..');
14   % Compute correlation matrix of size K
15   % correlation matrix will be of size p x p
16   result = zeros(N, 1);
17   anomaly_map = zeros(N,1);
18   anomalies_detected=zeros(p,N/2);
19   anomalies_detected_transpose_sum = zeros(p,p);
20   %tresh_LRX = 6.0000e+14;
21   tresh_LRX = threshold;
22   location_of_anomalies= zeros(N/2,1);
23   local_anomalies_set=0;
last_local_anomalies_set = 0;
ROWS = 100;
t_an = 1;

flag_local_anomaly_found = 0;
for j = 1:N
    autocorr = hyperCorrK(M, K, p);
    adaptive_autocorr_inv = pinv(autocorr -
        local_anomalies_set);
    result(j) = M(:, j)' * adaptive_autocorr_inv * M(:, j);
    if result(j) > thresh_LRX
        % This pixel is an anomaly! Add it to the set of anomalies
        anomalies_detected(:, t_an) = M(:, j);
        anomaly_map(j) = 1;
        location_of_anomalies(t_an) = j;
        anomalies_detected_transpose_sum = M(:, j)' * M(:, j);
        + anomalies_detected_transpose_sum;
        t_an = t_an + 1;
    end
    % if anomalies_detected_transpose_sum contains elements
    % from outside the KERNEL
    lower_limit_matrix = j - floor(K/2);
    higher_limit_matrix = j + floor(K/2);
    % Check if index is out of bounds
    if ( lower_limit_matrix < 1)
        % for edges of the matrix, gonna assume that we just throw out points
        % outside of the edge, and use half the KERNEL
        lower_limit_matrix = 1;
    end
    if ( higher_limit_matrix > N)
        % M(band, neighbouring_pixels) * (M(band, Neighbouring pixels)
        higher_limit_matrix = N;
    end
    if (any(local_anomalies_set))
        % just to check that it works
        last_local_anomalies_set = local_anomalies_set;
    end
    % resetting local_anomalies_set before using it the next
iteration
local_anomalies_set = 0;
flag_local_anomaly_found =0;
for i=1:t_an
    if flag_local_anomaly_found == 0
        for k=1:K
            if (flag_local_anomaly_found==0)
                if (location_of_anomalies( i ) >
                    lower_limit_matrix+(k−1)∗ROWS &
                    location_of_anomalies( i ) <
                    higher_limit_matrix+(k−1)∗ROWS)
                    local_anomalies_set =
                    local_anomalies_set +
                    anomalies_detected(:,i)∗
                    anomalies_detected(:,i),',
                    flag_local_anomaly_found =1;
                    break ;
            end
        end
    end
%if location_of_anomalies(t_an)<j−floor(K/2) |
% location_of_anomalies(t_an)>j+floor(K/2)
% anomalies_detected_transpose_sum =
% anomalies_detected_transpose_sum − M(:,t_an)∗M(:,
t_an),';
%end
%result(j)= result(j) ∗ M(:,i);
waitbar(j/N,h,'Updated progress');
end
result = abs(result);

A.1.5 ACAD anomaly detector

Listing A.5: ACAD
function [d_acad, anomaly_map, threshold_check_values] = 
    hyperACAD(M, thresh)
% HYPERRX Adaptive Causal Anomaly detector
% hyperLRxDetector performs the Adaptive Causal detector using
% correlation matrix
% It is adaptive in the sense that it removes the previously
detected
% anomalies from the correlation set
% Usage
% [result] = hyperACAD(M)
APPENDIX A. MATLAB HYPERSPECTRAL

% Inputs
%  M – 2D data matrix (p x N)
%  thresh – Threshold for a pixel to be considered anomaly
% Outputs
%  d_acad – Detector output (1 x N)
%  anomalies_detected – (1 x t_an)

% t_an is the number of anomalies detected. Since MatLab is 1–index, T is
% initially set to 1, not 0.
t_an = 1;

% bheta is the ratio of the entire image size to the size of
% anomaly
bheta = 100;
% bheta = 50;

% p is number of spectral bands, N is number of pixels
[p, N] = size(M);

% anomalies_detected is the growing set of anomalies detected in
% the image.
% Numbers of anomalies will not exceed N/2. Even that is way to
% much.
% Starting point N/2. Need to include the pixel it was found, j.
% Make some
% kind of map
anomalies_detected = zeros(p, N/2);

% anomalies_detected_transpose_sum is the sum of the transposes
% taken on
% anomalous pixels
anomalies_detected_transpose_sum = zeros(p, p);

% n_acad is used in the process of setting the threshold for
% finding an
% anomaly
n_acad = (N/bheta);

% u_k is the expected value/causal mean in the image. Initial
% value is set
% to the first pixel. This is wrong!!
u_k = M(:,1);

% thresh is the threshold value used to consider if the pixel is
% an anomaly
% or not. I think that it the anomaly detection will be
A.1. HIGH LEVEL MODELS OF ALGORITHMS

normalized...(???)

% Grubbs test for setting threshold?
% 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);

% for all N= m x n pixels
for j=1:N
    % want to store the result of hyperCausalCcorr, in case
    % this pixel is an 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 ');
APPENDIX A. MATLAB HYPERSPECTRAL

% end
%
% if (j>floor(n_acad))
  adaptive_autocorr_inv = pinv((autocorr -
anomalies_detected_transpose_sum)/(n_acad-t_an));
else
  adaptive_autocorr_inv = pinv((autocorr -
anomalies_detected_transpose_sum)/(j-t_an));
end

%temp_acad = M(:,j) .* adaptive_autocorr_inv * M(:,j);
d_acad(j) = M(:,j) .* adaptive_autocorr_inv * M(:,j);

if (j>floor(n_acad))
  u_k_un_normalized = prev_u_k + d_acad(j) - d_acad(j-
  floor(n_acad));
  %u_k_un_normalized = sum(d_acad(j-n_acad:j));
else
  u_k_un_normalized = prev_u_k + d_acad(j);
end
u_k = (1/n_acad) * u_k_un_normalized;
%u_k = abs(u_k);

%disp(d_acad(j)-u_k);
threshold_check_values(j) = d_acad(j)-u_k;
%if (abs(d_acad(j) - u_k)) > tresh
  if ((d_acad(j) - u_k)) > tresh
    % This pixel is an anomaly! Add it to the set of anomalies
    anomalies_detected(:,t_an) = M(:,j);
    location_of_anomalies(t_an)=j;
    anomalies_detected_transpose_sum = M(:,j)*M(:,j).
    + anomalies_detected_transpose_sum;
    t_an = t_an + 1;
  end
prev_u_k = u_k_un_normalized;
waitbar(j/N,h, 'Updated progress ACAD');
end
anomaly_map= zeros(1,N);
for i=1:1:N/2
  if (anomalies_detected(1,i)== 0)
    pixel_pos_anomaly = location_of_anomalies(i);
    anomaly_map(pixel_pos_anomaly) = 1;
  end
end

end
A.2 Testing

A.2.1 Hyper demo detectors

Listing A.6: Hyper Demo Detector

```matlab
function hyperDemo_detectors
% HYPERDEMO_DETECTORS Demonstrates target detector algorithms
clear; clc; dbstop if error; close all;
%
%% Parameters
%resultsDir = 'E:\One Drive\OneDrive for Business\NTNU\Master\Forked_MATLAB_hyperspectral_toolbox\MATLAB_Hyperspectral_toolbox\results';
dataDir = 'E:\One Drive\OneDrive for Business\NTNU\Master\Forked_MATLAB_hyperspectral_toolbox\MATLAB_DEMO_hyperspectral\f970619t01p02r02c';

resultsDir = ['E:\One Drive\OneDrive for Business\NTNU\Master\Anomaly detection results\MATLAB\LRX\real image data Cuprite scene\' , datestr(now, 'dd-mmm-yyyy')];
dataDir = 'E:\One Drive\OneDrive for Business\NTNU\Master\MATLAB_DEMO_hyperspectral\f970619t01p02r02c';
%
mkdir(resultsDir);

%% Read part of AVIRIS data file that we will further process
M = hyperReadAvirisRfl(sprintf('%s\f970619t01p02_r02_sc02.a.rfl', dataDir), [1 100], [1 614], [1 224]);
%M = hyperReadAvirisRfl(sprintf('%s\f970619t01p02_r02_sc04.a.rfl', dataDir), [1 100], [1 614], [1 224]);
M = hyperNormalize(M);

%% Read AVIRIS .spc file
lambdasNm = hyperReadAvirisSpc(sprintf('%s\f970619t01p02_r02_sc02.a.spc', dataDir));

%% Isomorph
[h, w, p] = size(M);
M = hyperConvert2d(M);
KSC_2d = hyperConvert2d(KSC);
```
% M = KSC_2d;
% Resample AVIRIS image.
desiredLambdasNm = 400:(2400−400)/(224−1):2400;
M = hyperResample(M, lambdasNm, desiredLambdasNm);

%% Remove low SNR bands.
goodBands = [10:100 116:150 180:216]; % for AVIRIS with 224 channels
%goodbands_KSC = [10:100 116:150];
KSC_2d = KSC_2d(goodbands_KSC, :);
M = M(goodBands, :);

%% Demonstrate difference spectral similarity measurements
M = hyperConvert3d(M, h, w, p);
target = squeeze(M(11, 77, :));
figure; plot(desiredLambdasNm(goodBands), target); grid on;
title('Target Signature; Pixel (32, 257)');
M = hyperConvert2d(M);

%% RX Anomaly Detector
%r = hyperRxDetector(M);
%r = hyperRxDetectorCor(M);
K = 23;
resultsDir = ['E:\One Drive\OneDrive for Business\NTNU\Master\Anomaly detection results\MATLAB\LRX\real image data Cuprite scene\', datestr(now, 'dd−mmm−yyyy')];
%r = hyperLRxDetectorCorr(M,K);
%g = ground_truth(h,614, M, M_endmembers);
%figure; imagesc(g); colorbar;
treshold = 500;
for treshold = 500:250:2000
[r, anomalies_detected, location_of_anomalies, last_local_anomalies_set] = hyperLRX_anomaly_set_remover(M,K, treshold);
[r, anomalies_detected, location_of_anomalies, last_local_anomalies_set] = hyperLRX_anomaly_set_remover(KSC_2d, K, treshold);
end
figure; imagesc(r); title(['ALRX Detector Results. K=23, tresh =', num2str(treshold)]); axis image;
colorbar;
hyperSaveFigure(gcf, sprintf(['%s\ALRX Detector Results. K=23, tresh =', num2str(treshold)], resultsDir));
% Testing

tresh = num2str(treshold) '.png' ] , resultsDir ));

figure; imagesc(r); title(['LRX removing anomalies, tresh =2000, K=25 .']); axis image;

figure; imagesc(r); title(['LRX Cuprite image data sc02 K=' num2str(K) '.']); axis image;

figure; imagesc(r); title(['Anomaly map ACAD, using LUTs, treshold = ' num2str(treshold) '.']); axis image;

figure; imagesc(abs(r)); title('CEM Detector Results'); axis image;

figure; imagesc(r); title('ACE Detector Results'); axis image;

figure; imagesc(r); title('Signed ACE Detector Results'); axis image;

figure; imagesc(r); title('Matched Filter'); axis image;

%% Constrained Energy Minimization (CEM)
r = hyperCem(M, target);
r = hyperConvert3d(r, h, w, 1);
figure; imagesc(abs(r)); title('CEM Detector Results'); axis image;

%% Adaptive Cosine Estimator (ACE)
r = hyperAce(M, target);
r = hyperConvert3d(r, h, w, 1);
figure; imagesc(r); title('ACE Detector Results'); axis image;

%% Signed Adaptive Cosine Estimator (S−ACE)
r = hyperSignedAce(M, target);
r = hyperConvert3d(r, h, w, 1);
figure; imagesc(r); title('Signed ACE Detector Results'); axis image;

%% Matched Filter
r = hyperMatchedFilter(M, target);
APPENDIX A. MATLAB HYPERSONTICAL

r = hyperConvert3d(r, h, w, 1);
figure; imagesc(r); title('MF Detector Results'); axis image;
colorbar;
hyperSaveFigure(gcf, sprintf('%s\mf detector.png', resultsDir));

%% Generalized Likelihood Ratio Test (GLRT) detector
r = hyperGlrt(M, target);
r = hyperConvert3d(r, h, w, 1);
figure; imagesc(r); title('GLRT Detector Results'); axis image;
colorbar;
hyperSaveFigure(gcf, sprintf('%s\cem detector.png', resultsDir));

%% Estimate background endmembers
U = hyperAtgp(M, 5);

%% Hybrid Unstructured Detector (HUD)
r = hyperHud(M, U, target);
r = hyperConvert3d(r, h, w, 1);
figure; imagesc(abs(r)); title('HUD Detector Results'); axis image;
colorbar;
hyperSaveFigure(gcf, sprintf('%s\hud detector.png', resultsDir));

%% Adaptive Matched Subspace Detector (AMSD)
r = hyperAmsd(M, U, target);
r = hyperConvert3d(r, h, w, 1);
figure; imagesc(abs(r)); title('AMSD Detector Results'); axis image;
colorbar;
hyperSaveFigure(gcf, sprintf('%s\amsd detector.png', resultsDir));
figure; mesh(r); title('AMSD Detector Results');

%% Orthogonal Subspace Projection (OSP)
r = hyperOsp(M, U, target);
r = hyperConvert3d(r, h, w, 1);
figure; imagesc(abs(r)); title('OSP Detector Results'); axis image;
colorbar;
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 аврис 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);
  end
  end

% create kernels with anomalies of size 2x2 with bottom left pixel in 15,15
%column locations
KERNEL_SIZE_TWO_LOCATION = 15;
APPENDIX A. MATLAB HYPERSONTEAL

39 image_30_30(KERNEL_SIZE_TWO_LOCATION,KERNEL_SIZE_TWO_LOCATION,:)
   = M_endmembers(:,3);
40 reference_anomaly_map(KERNEL_SIZE_TWO_LOCATION+1,
     KERNEL_SIZE_TWO_LOCATION)=1;
41 reference_anomaly_map(KERNEL_SIZE_TWO_LOCATION,
     KERNEL_SIZE_TWO_LOCATION)=1;
42 reference_anomaly_map(KERNEL_SIZE_TWO_LOCATION+1,
     KERNEL_SIZE_TWO_LOCATION+1)=1;
43 reference_anomaly_map(KERNEL_SIZE_TWO_LOCATION,
     KERNEL_SIZE_TWO_LOCATION+1)=1;
44 reference_anomaly_map(KERNEL_SIZE_TWO_LOCATION+1,
     KERNEL_SIZE_TWO_LOCATION+1)=1;

45 image_30_30(KERNEL_SIZE_TWO_LOCATION+1,KERNEL_SIZE_TWO_LOCATION
     ,:) = M_endmembers(:,3);
46 image_30_30(KERNEL_SIZE_TWO_LOCATION,KERNEL_SIZE_TWO_LOCATION
     +1,:)= M_endmembers(:,3);
47 image_30_30(KERNEL_SIZE_TWO_LOCATION+1,KERNEL_SIZE_TWO_LOCATION
     +1,:)= M_endmembers(:,3);

48 imnoise(image_30_30,'gaussian',1);
49 matrix=hyperConvert2d(image_30_30);
50 %[d_acad, anomaly_map, threshold_check_values] = hyperACAD(matrix
     ,100);
51 % K is size of kernel
52 K=5;
53 threshold = 0.9;
54 %[r_alrx,anomaly_map,not_used,not_use] =
     hyperLRX_anomaly_set_remover(matrix,K,threshold);
55 [r_alrx,anomaly_map,not_used] = hyperACAD(matrix,threshold);
56 %d_acad_2d = hyperConvert3d(d_acad,'.',30,30,1);
57 r_alrx_2d = hyperConvert3d(r_alrx,'.',30,30,1);
58 anomaly_map_2d = hyperConvert3d(anomaly_map,'.',30,30,1);
59 %figure;imagesc(r_alrx_2d);title(['ALRX AD detector, K= ' num2str(K)]);
60 %figure;imagesc(r_alrx_2d);title(['ACAD result, treshold ' num2str(treshold)]);
61 %figure;imagesc(anomaly_map_2d);title(['ACAD anomaly map, treshold= ' num2str(treshold)]);
62 % Evaluate the performance of the AD by setting objective
63 % find max value outputted from the AD
64 %max_ad_score = max(r_rlx);
65 treshold_percentage = 0.75;
66 predicted_anomalies =0;
67 % for i=1:w
68 % if r_rlx(i)>=treshold_percentage * max_ad_score
A.2. TESTING

```matlab
% predicted_anomalies = predicted_anomalies + 1;
% 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

```matlab
% for Cuprite scene
cle; 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);
```
if dice>4
    image(i,j,:)= M_endmembers(:,1) ; %setting background to alunite
elseif dice>2
    image(i,j,:)= M_endmembers(:,6) ; %setting background to Kalonite
else
    image(i,j,:)= M_endmembers(:,10); % setting background to pyrope
end

% rN=rand ;
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);

end
end

%imnoise(image , ' gaussian ' ,1);

%setting 50 random pixels to be an anomaly
% for i=1: 50
% h_index=randi(h);
% w_index= randi(w);
% signature_index = randi([2 12]);
% image(:,h_index,w_index) = M_endmembers(:,signature_index);
% anomaly_map(h_index,w_index)=1;
% end

%create kernels with anomalies of size 1, 5, 10,15, 20, 25 in columns 5, 20 ,50 ,100, 400, 600, in row 35 and 70
%column locations
KERNEL_SIZE_ONE_LOCATION =50;
KERNEL_SIZE_TWO_LOCATION = 100;
KERNEL_SIZE_FIVE_LOCATION =150;
KERNEL_SIZE_TEN_LOCATION =250;
KERNEL_SIZE_FIFTEEN_LOCATION =350;
KERNEL_SIZE_TWENTY_LOCATION =450;
KERNEL_SIZE_TWENTYFIVE_LOCATION =550;
for i=1:h
    if (mod(i,35)==0)
        image(i,KERNEL_SIZE_ONE_LOCATION,:) = M_endmembers(:,3);
        reference_anomaly_map(i,KERNEL_SIZE_ONE_LOCATION)=1;
    end
    image(i,KERNEL_SIZE_TWO_LOCATION,:) = M_endmembers(:,3);
    reference_anomaly_map(i,KERNEL_SIZE_TWO_LOCATION)=1;
    image(i,KERNEL_SIZE_TWENTY_LOCATION +1,:) = M_endmembers
A.2. TESTING

reference_anomaly_map(i, KERNEL_SIZE_TWO_LOCATION +1) = 1;
image(i+1, KERNEL_SIZE_TWO_LOCATION,:) = M_endmembers(:, 3);
reference_anomaly_map(i+1, KERNEL_SIZE_TWO_LOCATION) = 1;
image(i+1, KERNEL_SIZE_TWO_LOCATION +1,:) = M_endmembers(:,3);
reference_anomaly_map(i+1, KERNEL_SIZE_TWO_LOCATION +1) = 1;
for (j = 5:5:25)
    for row = i-floor(j/2): i+floor(j/2)
        switch j
            case 5
            colcenter = KERNEL_SIZE_FIVE_LOCATION;
            case 10
            colcenter = KERNEL_SIZE_TEN_LOCATION;
            case 15
            colcenter = KERNEL_SIZE_FIFTEEN_LOCATION;
            case 20
            colcenter = KERNEL_SIZE_TWENTY_LOCATION;
            case 25
            colcenter = KERNEL_SIZE_TWENTYFIVE_LOCATION;
        end
        for col = colcenter-floor(j/2): colcenter+floor(j/2)
            image(row, col,:) = M_endmembers(:, 3);
            reference_anomaly_map(row, col) = 1;
        end
    end
end
imnoise(image, 'gaussian', 1);
matrix = hyperConvert2d(image);

%f_r_rx = hyperRxDetector(matrix);
%f_r_rx_2d = hyperConvert3d(r_rx, h, w, 1);
% figure; imagesc(reference_anomaly_map); title ('Expected anomaly map'); axis image; colorbar;
% figure; imagesc(r_rx_2d); title ('RX AD results'); axis image; colorbar;
% max_value_rx= max(r_rx);
% set 75% of max_value as an anomaly
% treshold_rx = max_value_rx*0.75;
% anomaly_map_rx=zeros(h,w);
% for i=1:h
% for j=1:w
% if r_rx_2d(i , j ) >=treshold_rx
% anomaly_map_rx(i , j )=1;
% end
% end
% figure ; imagesc(anomaly_map_rx) ; title ('RX anomaly map'); axis image; colorbar;
% 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
% 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 = (nnz(anomaly_map_rx) - false_anomalies_hsueh_rx)/nnz(reference_anomaly_map);

% K=25;
% r_rlx =hyperLRxDetectorCorr(matrix ,K);
% r_rlx_2d = hyperConvert3d(r_rlx . ' , h, w, 1);
% anomaly_map_2d = hyperConvert3d(anomaly_map. ' , 30, 30, 1);
% figure ; imagesc(r_rlx_2d); title (['`LRX AD detector, K= ' num2str (K) '] ); axis image; colorbar;
% treshold =100;
A.2. TESTING

Listing A.9: Hsueh mimicked image

```matlab
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-mm-yyyy')];
resultsDir = regexprep(resultsDir, ':d*', '');
resultsDir = 'E:\One Drive\OneDrive for Business\NTNU\Master\Anomaly detection results\MATLAB\synthetic_images\lol';
```
```matlab
[status, msg, msgID] = mkdir(resultsDir);

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);

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);

SNR = 20;

% Setting background
for i=1:h
    for j=1:w
        image(i,j,:) = BACKGROUND;
    end
end

%column locations
KERNEL_SIZE_TWO_LOCATION = 40; %column one
KERNEL_SIZE_TWO_2_LOCATION = 70; %column two, mixed pixels
KERNEL_SIZE_TWO_MIXED_LOCATION = 100; %column three, mixed pixel
KERNEL_SIZE_ONE_BKG_MIXED_LOCATION = 130; %column four, mixed pixel and background, 50/50
KERNEL_SIZE_ONE_BKG_75_MIX_LOCATION = 160; %column five, mixed pixel and background, 25/75

M_anomaly_pure = M_endmembers(:,10);
M_A = M_endmembers(:,1);
M_B = M_endmembers(:,3);
M_K = M_endmembers(:,5);
M_M = M_endmembers(:,7);
M_C = M_endmembers(:,12);

for i=KERNEL_SIZE_TWO_LOCATION:30:
    %first column
    image(i,KERNEL_SIZE_TWO_LOCATION,:) = M_anomaly_pure;
    reference_anomaly_map(i,KERNEL_SIZE_TWO_LOCATION) = 1;
end

image(i,KERNEL_SIZE_TWO_LOCATION +1,:) = M_anomaly_pure;
reference_anomaly_map(i,KERNEL_SIZE_TWO_LOCATION +1) = 1;

image(i+1,KERNEL_SIZE_TWO_LOCATION,:) = M_anomaly_pure;
reference_anomaly_map(i+1,KERNEL_SIZE_TWO_LOCATION) = 1;

image(i+1,KERNEL_SIZE_TWO_LOCATION +1,:) = M_anomaly_pure;
reference_anomaly_map(i+1,KERNEL_SIZE_TWO_LOCATION +1) = 1;
```

A.2. TESTING

```matlab
% second column
image(i ,KERNEL_SIZE_TWO_2_LOCATION,: ) = M_anomaly_pure;
reference_anomaly_map(i ,KERNEL_SIZE_TWO_2_LOCATION)=1;
image(i ,KERNEL_SIZE_TWO_2_LOCATION +1,: ) = M_anomaly_pure;
reference_anomaly_map(i ,KERNEL_SIZE_TWO_2_LOCATION +1)=1;
image(i+1,KERNEL_SIZE_TWO_2_LOCATION,: ) = M_anomaly_pure;
reference_anomaly_map(i+1,KERNEL_SIZE_TWO_2_LOCATION)=1;
image(i+1,KERNEL_SIZE_TWO_2_LOCATION +1,: ) = M_anomaly_pure;
reference_anomaly_map(i+1,KERNEL_SIZE_TWO_2_LOCATION +1)=1;

for (j=KERNEL_SIZE_TWO_MIXED_LOCATION:30:
    KERNEL_SIZE_ONE_BKG_75_MIX_LOCATION)
    switch j
        case KERNEL_SIZE_TWO_MIXED_LOCATION
            switch i
                case 40
                    image(i+1,j ,:)= 0.5*M_A + 0.5 *M_B;
                    image(i+1,j +1,:)=0.5*M_A + 0.5*M_C;
                    image(i , j ,:) = 0.5*M_A + 0.5*M_K;
                    image(i , j +1,:)= 0.5*M_A + 0.5*M_M;
                case 70
                    image(i+1,j ,:)= 0.5*M_A + 0.5 *M_B;
                    image(i+1,j +1,:)=0.5*M_A + 0.5*M_C;
                    image(i , j ,:) = 0.5*M_B + 0.5*M_K;
                    image(i , j +1,:)= 0.5*M_B + 0.5*M_M;
                case 100
                    image(i+1,j ,:)= 0.5*M_A + 0.5 *M_C;
                    image(i+1,j +1,:)=0.5*M_B + 0.5*M_C;
                    image(i , j ,:) = 0.5*M_C + 0.5*M_K;
                    image(i , j +1,:)= 0.5*M_C + 0.5*M_M;
                case 130
                    image(i+1,j ,:)= 0.5*M_A + 0.5 *M_K;
                    image(i+1,j +1,:)=0.5*M_B + 0.5*M_K;
                    image(i , j ,:) = 0.5*M_C + 0.5*M_K;
                    image(i , j +1,:)= 0.5*M_K + 0.5*M_M;
                case 160
                    image(i+1,j ,:)= 0.5*M_A + 0.5 *M_M;
                    image(i+1,j +1,:)=0.5*M_B + 0.5*M_M;
                    image(i , j ,:) = 0.5*M_C + 0.5*M_M;
                    image(i , j +1,:)= 0.5*M_K + 0.5*M_M;
            end
        end
    end
reference_anomaly_map(i ,j )=1;
reference_anomaly_map(i+1,j )=1;
reference_anomaly_map(i+1,j +1)=1;
reference_anomaly_map(i ,j +1)=1;
```

case 40
    image(i,j,:) = 0.5*M_A + 0.5*BACKGROUND;
    reference_anomaly_map(i,j)=1;
    end
case 70
    image(i,j,:) = 0.5*M_B + 0.5*BACKGROUND;
    reference_anomaly_map(i,j)=1;
    end
case 100
    image(i,j,:) = 0.5*M_C + 0.5*BACKGROUND;
    reference_anomaly_map(i,j)=1;
    end
case 130
    image(i,j,:) = 0.5*M_K + 0.5*BACKGROUND;
    reference_anomaly_map(i,j)=1;
    end
case 160
    image(i,j,:) = 0.5*M_M + 0.5*BACKGROUND;
    reference_anomaly_map(i,j)=1;
end
end
end
end
for (i=1:n_bands)
    image(:, :, i) = awgn(image(:, :, i),SNR);
end
figure; imagesc(image(:, :, 160)); title('Band 160 of Hsueh−mimicked image, with gaussian noise'); axis image;
%
RX testing
matrix = hyperConvert2d(image);
r_rx=hyperRxDetector(matrix); r_rx = hyperConvert3d([r_rx.', h, w, 1]);
figure; imagesc(reference_anomaly_map); title(['Expected anomaly map']); axis image; colorbar;
hyperSaveFigure(gcf, sprintf(['%s\hsueh_expected_anomaly_map']) );
A.2. TESTING

max_value_rx = max(r_rx);
max_value_rx = max(max_value_rx);
% set 90% of max_value as an anomaly
threshold_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) >= threshold_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

figure; imagesc(anomaly_map_rx); title(['RX anomaly map']);
axis image; colorbar;
hyperSaveFigure(gcf, sprintf(['%s\RX anomaly_map' '.png'], resultsDir));

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_hsueh_rx / nnz(reference_anomaly_map);
%% LRX without anomaly removal

difference_from_reference_lrx = zeros(h,w);
counter_i=1;

anomaly_map_alrx=zeros(1,h∗w);

for K=5:5:30 % 35 bugged
    difference_from_reference_lrx = zeros(h,w);
    r_lrx=hyperLRxDetectorCorr(matrix ,K);
    r_lrx = hyperConvert3d(r_lrx .', h, w, 1);
    figure; imagesc(r_lrx); title(['LRX K=' num2str(K) ] );
          axis image; colorbar;
    hyperSaveFigure(gcf , sprintf(['%s\LRX K=' num2str(K) '.png ' ], resultsDir ));%
    max_value_lrx= max(r_lrx);
    max_value_lrx = max(max_value_lrx);
    % set 75% of max_value as an anomaly
    treshold_lrx = max_value_lrx ∗0.75;
    anomaly_map_lrx=zeros(h,w);
    correctly_predicted_anomalies_lrx =0;
    for i=1:h
        for j=1:w
            if r_lrx(i , j ) >=treshold_lrx
                anomaly_map_lrx(i , j )=1;
                if reference_anomaly_map(i , j )==1
                    correctly_predicted_anomalies_lrx =
                    correctly_predicted_anomalies_lrx+1;
                end
            end
        end
    end

    for i=1:h
        for j=1:w
            difference_from_reference_lrx(i , j )= (reference_anomaly_map(i , j )− anomaly_map_lrx(i , j ));
        end
    end

    figure; imagesc(difference_from_reference_lrx); title(['False or undetected anomalies LRX, K=' num2str(K) ] );
          axis image;
          colorbar;
    hyperSaveFigure(gcf , sprintf(['%s\false anomalies LRX K=' num2str(K) '.png ' ], resultsDir ));%
    false_anomalies_hsu_eh_lrx(counter_i)=nnz(anomaly_map_lrx)− nnz( reference_anomaly_map );
    if (false_anomalies_hsu_eh_lrx(counter_i)<0)
        false_anomalies_hsu_eh_lrx(counter_i)=0;
    end
    percent_predicted_anomalies_hsu_eh_lrx(counter_i) =
    correctly_predicted_anomalies_lrx/nnz(reference_anomaly_map);
    percent_predicted_anomalies_hsu_eh_lrx(counter_i) = (nnz( anomaly_map_lrx ) − false_anomalies_hsu_eh_lrx(counter_i))/nnz( anomaly_map_lrx ) − false_anomalies_hsu_eh_lrx(counter_i);
A.2. TESTING

```matlab
reference_anomaly_map);
  nnz_lrx(counter_i) = nnz(difference_from_reference_lrx);
counter_i = counter_i +1;
end

%LRX with anomaly removal
difference_from_reference_lrx_ad_remov = zeros(h,w);
  counter_i=1;
correctly_predicted_anomalies_alrx=0;
treshold =250;
for K=5:5:35
  difference_from_reference_lrx_ad_remov = zeros(h,w);
  [r_lrx_ad_remov ,anomaly_map_alrx , location_of_anomalies , lsls] =
  hyperLRX_anomaly_set_remover(matrix ,K, treshold);
  r_lrx_ad_remov = hyperConvert3d(r_lrx_ad_remov, ' , h, w, 1);
  figure ; imagesc(r_lrx_ad_remov) ; title(['ALRX AD K=' num2str(K) ] ) ; axis image; colorbar ;
  hyperSaveFigure(gcf , sprintf(['%s\ALRX AD K=' num2str(K) '.png ' ] , resultsDir ));%
  anomaly_map_alrx_2d = hyperConvert3d(anomaly_map_alrx, ' , h,w,1);
  for i=1:h
    for j=1:w
      if anomaly_map_alrx_2d(i , j ) ==1 &&
        reference_anomaly_map(i , j ) ==1
        correctly_predicted_anomalies_alrx =
        correctly_predicted_anomalies_alrx +1;
      end
    end
  end
false_anomalies_hsueh_alrx(counter_i)=nnz(anomaly_map_alrx) −
correctly_predicted_anomalies_alrx;
  if (false_anomalies_hsueh_alrx(counter_i)<0)
  false_anomalies_hsueh_alrx=0;
  end
percent_predicted_anomalies_hsueh_alrx(counter_i) = (nnz( anomaly_map_alrx) − false_anomalies_hsueh_alrx)/nnz( reference_anomaly_map);
percent_predicted_anomalies_hsueh_alrx(counter_i) =
correctly_predicted_anomalies_alrx/nnz(reference_anomaly_map)
;
figure ; imagesc(difference_from_reference_rx); title(['False or undetected anomalies ALRX AD, K=' num2str(K) ] ) ; axis image; colorbar;
hyperSaveFigure(gcf , sprintf(['%s\false anomalies ALRX AD K=' num2str(K) '.png ' ] , resultsDir ));%
nz_lrx(counter_i) = nnz(difference_from_reference_lrx_ad_remov);
counter_i = counter_i +1;
end
```
APPENDIX A. MATLAB HYPERSPECTRAL

```matlab
%% ACAD
matrix = hyperConvert2d(image);
treshold= 0.9;
[r_acad,anomaly_map,not_used]=hyperACAD(matrix,treshold);
r_acad = hyperConvert3d(r_acad.', h, w, 1);
figure; imagesc(reference_anomaly_map); title(['Expected anomaly map']); axis image; colorbar;
figure; imagesc(r_acad); title(['ACAD ']); axis image; colorbar;
figure; imagesc(anomaly_map); title(['Anomaly map ']); axis image; colorbar;

%% format data
r_acad_formatted = zeros(w,h);
for i =1: w
    for j=1:h
        if r_acad(i,j) <0
            r_acad_formatted(i,j)=0;
        elseif r_acad(i,j) >0 && ~isinf(r_acad(i,j))
            r_acad_formatted(i,j) = r_acad(i,j);
        else
            r_acad_formatted(i,j)=0;
        end
    end
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 webpage: "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 B.1: Two process method.
Appendix C

VHDL code

C.1 ACAD correlation

Listing C.1: ACAD correlation

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.Common_types_and_functions.all;

−− Correlation module with AXI lite stream interface
entity acad_correlation is
  port(din : in std_logic_vector(P_BANDS∗PIXEL_DATA_WIDTH−1 downto 0); −− Horizontal
       −− input vector
       valid : in std_logic;
       clk : in std_logic;
       clk_en : in std_logic;
       reset_n : in std_logic;
       dout : out std_logic_vector(P_BANDS∗PIXEL_DATA_WIDTH∗2∗2 downto 0); −− writing two 32−bit elements per cycle
       valid_out : out std_logic;
       writes_done_on_column : out std_logic_vector(log2(P_BANDS/2) downto 0)
   ) ;
end acad_correlation;

architecture Behavioral of acad_correlation is
  −− using 18kbit BRAM, one for odd indexes, one for even per row
```

125
of the correlation matrix. This results in P_BANDS 36kbit BRAMs actually being synthesized.

\[
\text{constant NUMBER_OF_WRITES_PER_CYCLE : integer range 0 to 2 := 2; }
\]

\[
\text{constant NUMBER_OF_WRITES_PER_COLUMN : integer range 0 to P_BANDS/2 := P_BANDS/2;}
\]

\[
\text{signal r_write_address : integer range 0 to B_RAM_SIZE−1 := 0;}
\]

\[
\text{signal write_done_on_column : integer range 0 to P_BANDS/2 := 0;}
\]

\[
\text{signal flag_has_read_first : std_logic := '0'; --- first element in the read−write pipeline}
\]

\[
\text{signal flag_has_read_second : std_logic := '0'; --- second element in the read−write pipeline}
\]

\[
\text{signal write_enable : std_logic := '0';}
\]

\[
\text{signal read_enable : std_logic := '1';}
\]

\[
\text{signal read_address : integer range 0 to B_RAM_SIZE−1;}
\]

\[
\text{signal write_address : integer range 0 to B_RAM_SIZE−1;}
\]

\[
\text{signal flag_first_pixel : std_logic := '1'; --- indicates that the current pixel working on is the first pixel}
\]

\[
\text{signal r_dout_prev : std_logic_vector(P_BANDS*PIXEL_DATA_WIDTH ∗2∗2−1 downto 0); --- Previous value outputted from the BRAMs.}
\]

\[
\text{signal r_read_address : integer range 0 to B_RAM_SIZE−1 := 0;}
\]

\[
\text{constant EVEN_ROW_TOP_INDEX_INPUT : integer range 0 to P_BANDS*PIXEL_DATA_WIDTH−1 := P_BANDS*PIXEL_DATA_WIDTH−1;}
\]

\[
\text{constant EVEN_ROW_TOP_INDEX_CORRELATION : integer range 0 to P_BANDS*PIXEL_DATA_WIDTH*2−1 := P_BANDS*PIXEL_DATA_WIDTH*2−1;}
\]

\[
\text{signal dout_BRAMS : std_logic_vector(P_BANDS*PIXEL_DATA_WIDTH*2∗2−1 downto 0):= (others=>'}0'");
\]
C.1. ACAD CORRELATION

begin

GEN_BRAM_18_updates : for i in 0 to P_BANDS−1 generate

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);

begin

begin

−− Block ram row for even addresses and row indexes of the correlation matrix
block_ram_even : entity work.block_ram

generic map (
    B_RAM_SIZE => B_RAM_SIZE,
    B_RAM_BIT_WIDTH => B_RAM_BIT_WIDTH)

port map (
    clk => clk ,
    aresetn => reset_n ,
    data_in => data_in_even_i ,
    write_enable => write_enable ,
    read_enable => read_enable ,
    read_address => read_address ,
    write_address => write_address ,
    data_out => data_out_even_i);

begin

−− Block ram row for odd addresses and row indexes of the correlation matrix
block_ram_odd : entity work.block_ram

generic map (
    B_RAM_SIZE => B_RAM_SIZE,
    B_RAM_BIT_WIDTH => B_RAM_BIT_WIDTH)

port map (
    clk => clk ,
    aresetn => reset_n ,
    data_in => data_in_odd_i ,
    write_enable => write_enable ,
    read_enable => read_enable ,
    read_address => read_address ,
    write_address => write_address ,
    data_out => data_out_odd_i);

begin

−− generate P BAND write PROCESSES writes on clock cycle after 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)

−− a_factor_0x is the even indexed row factor
variable a_factor_01_i : std_logic_vector(PIXEL_DATA_WIDTH−1 downto 0);
APPENDIX C. VHDL CODE

variable a_factor_01_i : std_logic_vector(
PIEXL_DATA_WIDTH−1 downto 0);
variable b_factor_01_i : std_logic_vector(
PIEXL_DATA_WIDTH−1 downto 0);
variable b_factor_02_i : std_logic_vector(
PIEXL_DATA_WIDTH−1 downto 0);
variable v_input_even_i, v_input_odd_i : std_logic_vector(
B_RAM_BIT_WIDTH−1 downto 0);
variable v_data_out_prev_even_i : std_logic_vector(
PIXEL_DATA_WIDTH∗2−1 downto 0);
variable v_data_out_prev_odd_i : std_logic_vector(
PIXEL_DATA_WIDTH∗2−1 downto 0); begin

if rising_edge(clk) and clk_en = '1' then
  if reset_n = '0' or valid = '0' then
    a_factor_01_i := (others => '0');
    a_factor_02_i := (others => '0');
    b_factor_01_i := (others => '0');
    b_factor_02_i := (others => '0');
  elsif valid = '1' and write_done_on_column <=
    NUMBER_OF_WRITES_PER_COLUMN−1 and write_enable = '1'
    and to_integer(unsigned(write_done_on_column) > 0 then
    if flag_first_pixel = '0' then
      if flag_first_pixel = '0' then
        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);
        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);

    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));
    b_factor_02_i := a_factor_02_i;

    v_input_even_i := std_logic_vector(to_signed(
    to_integer(signed(a_factor_01_i))∗to_integer(

    v_data_out_prev_even_i := std_logic_vector(
    B_RAM_BIT_WIDTH−1 downto 0);
    v_data_out_prev_odd_i := std_logic_vector(
    B_RAM_BIT_WIDTH−1 downto 0);
\[\text{signed}(a_{\text{factor}_02_i}), v_{\text{input}\_even_i}'\text{length})\]
\[v_{\text{input}\_odd_i} := \text{std\_logic\_vector}(\text{to\_signed}(\text{to\_integer}(\text{signed}(b_{\text{factor}_01_i}))\star\text{to\_integer}(\text{signed}(b_{\text{factor}_02_i}), v_{\text{input}\_odd_i}'\text{length}));\]
\[v_{\text{data\_out\_prev\_even_i}} := r_{\text{dout\_prev}}(P_{\text{BANDS}}\star\text{PIXEL\_DATA\_WIDTH}\star2-(P_{\text{BANDS}}-i)\star\text{PIXEL\_DATA\_WIDTH}\star2+\text{PIXEL\_DATA\_WIDTH}\star2-1\text{downto }P_{\text{BANDS}}\star\text{PIXEL\_DATA\_WIDTH}\star2-(P_{\text{BANDS}}-i)\star\text{PIXEL\_DATA\_WIDTH}\star2);\]
\[v_{\text{data\_out\_prev\_odd_i}} := r_{\text{dout\_prev}}(P_{\text{BANDS}}\star\text{PIXEL\_DATA\_WIDTH}\star\text{NUMBER\_OF\_WRITES\_PER\_CYCLE}\star2-(P_{\text{BANDS}}-i)\star\text{PIXEL\_DATA\_WIDTH}\star2+\text{PIXEL\_DATA\_WIDTH}\star2-1\text{downto }P_{\text{BANDS}}\star\text{PIXEL\_DATA\_WIDTH}\star\text{NUMBER\_OF\_WRITES\_PER\_CYCLE}\star2-(P_{\text{BANDS}}-i)\star\text{PIXEL\_DATA\_WIDTH}\star2);\]
\[\text{data\_in\_even_i} <= \text{std\_logic\_vector}(\text{to\_signed}(\text{to\_integer}(\text{signed}(v_{\text{input}\_even_i})+\text{to\_integer}(\text{signed}(v_{\text{data\_out\_prev\_even_i}})), \text{data\_in\_even_i}'\text{length}));\]
\[\text{data\_in\_odd_i} <= \text{std\_logic\_vector}(\text{to\_signed}(\text{to\_integer}(\text{signed}(v_{\text{input}\_odd_i})+\text{to\_integer}(\text{signed}(v_{\text{data\_out\_prev\_odd_i}})), \text{data\_in\_odd_i}'\text{length}));\]
\[\text{elsif flag\_first\_pixel} = '1' \text{ then}\]
\[\text{−− special case for the first pixel written, where}\]
\[\text{−− the data contained in the BRAM is not}\]
\[\text{−− initialized to something known.}\]
\[\text{−−input din is horizontal vector. A/B\_factor 01 is}\]
\[\text{−−the transposed}\]
\[\text{−−vertical element factor of the product din.\'} \ast \text{ din}\]
\[\text{−−the horizontal element.}\]
\[a_{\text{factor}_01_i} := \text{din}(\text{PIXEL\_DATA\_WIDTH}-1+\text{PIXEL\_DATA\_WIDTH}\star\text{write\_done\_on\_column}\star\text{NUMBER\_OF\_WRITES\_PER\_CYCLE}\text{downto}\text{PIXEL\_DATA\_WIDTH}\star\text{write\_done\_on\_column}\star\text{NUMBER\_OF\_WRITES\_PER\_CYCLE});\]
\[b_{\text{factor}_01_i} := \text{din}(\text{PIXEL\_DATA\_WIDTH}\star2-1+\text{PIXEL\_DATA\_WIDTH}\star\text{write\_done\_on\_column}\star\text{NUMBER\_OF\_WRITES\_PER\_CYCLE}\text{downto}\text{PIXEL\_DATA\_WIDTH}\star\text{PIXEL\_DATA\_WIDTH}+\text{PIXEL\_DATA\_WIDTH}\star\text{write\_done\_on\_column}\star\text{NUMBER\_OF\_WRITES\_PER\_CYCLE});\]
\[\text{−− "Horizontal" element}\]
\[a_{\text{factor}_02_i} := \text{din}(P_{\text{BANDS}}\star\text{PIXEL\_DATA\_WIDTH}-(P_{\text{BANDS}}-i)\star\text{PIXEL\_DATA\_WIDTH}+\text{PIXEL\_DATA\_WIDTH}-1\text{downto }P_{\text{BANDS}}\star\text{PIXEL\_DATA\_WIDTH}-(P_{\text{BANDS}}-i)\star\text{PIXEL\_DATA\_WIDTH});\]
b_factor_02_i := a_factor_02_i;

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));

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));

data_in_even_i <= v_input_even_i;
data_in_odd_i <= v_input_odd_i;

end if;

end if;

end if;

end process;

--- Even row of output
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)
<= data_out_even_i;
dout(P_BANDS*PIXEL_DATA_WIDTH*2 - (P_BANDS-i)*PIXEL_DATA_WIDTH*2 +
    PIXEL_DATA_WIDTH*2 - (P_BANDS-i)*PIXEL_DATA_WIDTH*2)
<= data_in_even_i;

--- Odd row of output
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;
dout(P_BANDS*PIXEL_DATA_WIDTH*2 - (P_BANDS-i)*PIXEL_DATA_WIDTH*2 +
    PIXEL_DATA_WIDTH*2 - (P_BANDS-i)*
    PIXEL_DATA_WIDTH*2 + EVEN_ROW_TOP_INDEX_CORRELATION + 1)
<= data_in_odd_i;

end generate;

--- Register in old values of dout
process (clk, clk_en, dout)
begin
    if rising_edge(clk) and clk_en = '1' then
        r_dout_prev <= dout;
        r_dout_prev <= dout_BRAMS;
    end if;
end process;
C.1. ACAD CORRELATION

process to drive address and control

begin
if rising_edge(clk) and clk_en = '1' then
if reset_n = '0' then
  or valid = '0' then
    r_write_address <= 0;
    read_address <= 0;
    write_enable <= '0';
    read_enable <= '1';
    write_done_on_column <= 0;
    flag_has_read_first <= '0';
    flag_first_pixel <= '1';
    valid_out <= '0';
else valid = '0' then
  write_enable <= '0';
  flag_has_read_first <= '0';
  flag_has_read_second <= '0';
  valid_out <= '0';
elsif valid = '1' and write_done_on_column <= NUMBER_OF_WRITES_PER_COLUMN-1 and flag_first_pixel = '1' then
  if flag_has_read_first = '0' then
    -- Need to read first element of the pixel before starting any writes
    flag_has_read_first <= '1';
    read_address <= r_write_address;
    write_address <= r_write_address;
    read_enable <= '1';
    write_enable <= '1';
    valid_out <= '0';
elsif flag_has_read_first = '1' and write_enable = '1' then
  r_write_address <= r_write_address +1;
  write_address <= r_write_address;
  read_address <= r_write_address+1;
  write_enable <= '1';
  write_done_on_column <= write_done_on_column + 1;
  valid_out <= '1';
end if;
-- Going to buffer two read elements.
elsif valid = '1' and write_done_on_column <= NUMBER_OF_WRITES_PER_COLUMN-1 and flag_first_pixel = '0' then
  if flag_has_read_first = '0' and flag_has_read_second = '0' then
    -- Need to read first element of the pixel before
starting any writes
flag_has_read_first <= '1';
read_address <= r_write_address;
write_address <= r_write_address;
read_enable <= '1';
write_enable <= '0';
valid_out <= '0';
elif flag_has_read_first = '1' and write_enable = '0'
and flag_has_read_second = '0' then
read_address <= r_write_address +1;
read_enable <= '1';
flag_has_read_second <= '1';
r_read_address <= r_read_address +1;
valid_out <= '0';
end if;
if flag_has_read_second = '1' and write_enable = '0'
then
write_address <= r_write_address;
read_address <= r_write_address+2;
write_enable <= '1';
r_read_address <= r_read_address +1;
valid_out <= '0';
elif flag_has_read_second = '1' and write_enable = '1'
then
r_write_address <= r_write_address +1;
write_address <= r_write_address;
read_address <= r_read_address;
r_read_address <= r_read_address +1;
write_done_on_column <= write_done_on_column + 1;
valid_out <= '1';
end if;
elif valid = '1' and write_done_on_column >
NUMBER_OF_WRITES_PER_COLUMN−1 then
  -- New pixel coming on data_in input
  -- Assuming consequent pixels are hold valid, starting
  -- working on
  -- next pixel next cycle;
  valid_out <= '1';
r_write_address <= 0;
r_read_address <= 0;
read_address <= 0;
write_enable <= '0';
write_done_on_column <= 0;
flag_has_read_first <= '0';
flag_has_read_second <= '0';
  -- Now one pixel has been finished processed, the
  contents of the
  -- BRAM is at least known
  flag_first_pixel <= '0';
end if;
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

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;
--use IEEE.fixed_pkg.all;
library work;
use work.Common_types_and_functions.all;

-- This core is utilized by both backward and forward elimination
entity backward_elim_core is
port(clk : in std_logic;
reset_n : in std_logic;
clk_en : in std_logic;
input_backward_elim : in input_elimination_reg_type;
output_backward_elim : out
output_backward_elimination_reg_type);
end backward_elim_core;

architecture Behavioral of backward_elim_core is
signal r, r_in : input_elimination_reg_type;
constant ONE : signed(PIXEL_DATA_WIDTH*2−1 downto 0) := (0 => '1', others => '0');
constant PRECISION_SHIFT : integer range 0 to 3 := 3; -- Used to specify numbers of shift of r_j_i
signal divisor_is_negative : std_logic;
-- If the divisor is negative, we need to take two's complement of the divisor
signal divisor : std_logic_vector(PIXEL_DATA_WIDTH *2−1 downto 0);
```
APPENDIX C. VHDL CODE

```vhdl
signal divisor_valid : std_logic := '0';
signal remainder_valid : std_logic := '0';
type remainders_array is array(0 to PIXEL_DATA_WIDTH*2−2) of std_logic_vector(PIXEL_DATA_WIDTH*2−1 downto 0);
signal remainders : remainders_array;
signal msb_index : integer range 0 to 31; -- msb of the divisor(unsigned)
signal msb_valid : std_logic := '0';
-- to be used in two’s complement.
signal divisor_lut : unsigned(DIV_PRECISION−1 downto 0);
signal divisor_inv : unsigned(DIV_PRECISION−1 downto 0);
begin

division_lut_2 : entity work.division_lut
    port map (
y => divisor_lut,
y_inv => divisor_inv);
input_to_divisor_lut : process (msb_valid, msb_index)
begin
    if msb_valid = '1' and msb_index<=DIV_PRECISION then
        divisor_lut <= to_unsigned(to_integer(unsigned(divisor)), DIV_PRECISION);
    else
        divisor_lut <= to_unsigned(0, DIV_PRECISION);
    end if;
end process;

check_if_divisor_is_negative : process(input_backward_elim.state_reg.state, input_backward_elim.row_i, input_backward_elim.valid_data, reset_n)
begin
    if reset_n = '0' or not(input_backward_elim.state_reg.state = STATE_LAST_DIVISION) then
        divisor_valid <= '0';
        divisor_is_negative <= '0';
        divisor <= std_logic_vector(to_signed(1, PIXEL_DATA_WIDTH*2));
    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
        -- row[i][i] is negative
        -- using the absolute value
        divisor_is_negative <= '1';
```
C.2. ELIMINATION CORE

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

remainders(i−1) <= remainder_after_approximation_i . remainder;
remainder_valid <= remainder_after_approximation_i . remainder_valid;
end generate;

find_msb : process(divisor_valid, input_backward_elim, reset_n, divisor)
begin
if divisor_valid = '1' and reset_n = '1' then
−−For PIXEL_DATA_WIDTH = 16.
if divisor(30) = '1' then
msb_index <= 30;
msb_valid <= '1';
el if divisor(29) = '1' then
msb_index <= 29;
msb_valid <= '1';
el if divisor(28) = '1' then
msb_index <= 28;
msb_valid <= '1';
el if divisor(27) = '1' then
msb_index <= 27;
msb_valid <= '1';
el if divisor(26) = '1' then
msb_index <= 26;
msb_valid <= '1';
el if divisor(25) = '1' then
msb_index <= 25;
msb_valid <= '1';
el if divisor(24) = '1' then
msb_index <= 24;
msb_valid <= '1';
el if divisor(23) = '1' then
msb_index <= 23;
msb_valid <= '1';
el if divisor(22) = '1' then
msb_index <= 22;
msb_valid <= '1';
el if divisor(21) = '1' then
msb_index <= 21;
msb_valid <= '1';
el if divisor(20) = '1' then
msb_index <= 20;
msb_valid <= '1';
el if divisor(19) = '1' then
msb_index <= 19;
msb_valid <= '1';
el if divisor(18) = '1' then
msb_index <= 18;
C.2. ELIMINATION CORE

```vhdl
msb_valid <= '1';
elsif divisor (17) = '1' then
  msb_index <= 17;
  msb_valid <= '1';
elsif divisor (16) = '1' then
  msb_index <= 16;
  msb_valid <= '1';
elsif divisor (15) = '1' then
  msb_index <= 15;
  msb_valid <= '1';
elsif divisor (14) = '1' then
  msb_index <= 14;
  msb_valid <= '1';
elsif divisor (13) = '1' then
  msb_index <= 13;
  msb_valid <= '1';
elsif divisor (12) = '1' then
  msb_index <= 12;
  msb_valid <= '1';
elsif divisor (11) = '1' then
  msb_index <= 11;
  msb_valid <= '1';
elsif divisor (10) = '1' then
  msb_index <= 10;
  msb_valid <= '1';
elsif divisor (9) = '1' then
  msb_index <= 9;
  msb_valid <= '1';
elsif divisor (8) = '1' then
  msb_index <= 8;
  msb_valid <= '1';
elsif divisor (7) = '1' then
  msb_index <= 7;
  msb_valid <= '1';
elsif divisor (6) = '1' then
  msb_index <= 6;
  msb_valid <= '1';
elsif divisor (5) = '1' then
  msb_index <= 5;
  msb_valid <= '1';
elsif divisor (4) = '1' then
  msb_index <= 4;
  msb_valid <= '1';
elsif divisor (3) = '1' then
  msb_index <= 3;
  msb_valid <= '1';
elsif divisor (2) = '1' then
  msb_index <= 2;
  msb_valid <= '1';
elsif divisor (1) = '1' then
  msb_index <= 1;
  msb_valid <= '1';
end if;
```

APPENDIX C. VHDL CODE

msb_index <= 1;
msb_valid <= '1';
elsif divisor(0) = '1' then
  msb_index <= 0;
  msb_valid <= '1';
else
  msb_index <= 0;
  msb_valid <= '0';
end if;
else
  msb_index <= 0;
  msb_valid <= '0';
end if;
end process;

comb_process : process (input_backward_elim, r, reset_n,
  divisor_is_negative, divisor, remainder_valid, remainders,
  msb_valid, divisor, divisor_inv, msb_index)
  variable v :
    input_elimination_reg_type;
  variable r_j_i : signed(
    PIXEL_DATA_WIDTH*2 + PRECISION_SHIFT−1 downto 0);
  variable r_i_i : integer;
  variable temp : integer;
  --variable r_j_i_divided : signed (PIXEL_DATA_WIDTH*2+PRECISION_SHIFT−1 downto 0);
  variable inner_product : signed(
    PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2+PRECISION_SHIFT−1 downto 0);
  variable shifted_down_inner_product : signed(
    PIXEL_DATA_WIDTH*2−1 downto 0);
  --variable r_i_i_halv : integer;
  variable r_i_i_halv : signed(
    PIXEL_DATA_WIDTH*2+PRECISION_SHIFT−1 downto 0);
  variable divisor_inv_from_lut : integer range 0 to 2**DIV_PRECISION := 0;

begin
  v := r;
  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
    -- Load data set index_j
    v.row_j := input_backward_elim.row_j;
    v.row_i := input_backward_elim.row_i;
    v.inv_row_j := input_backward_elim.inv_row_j;
    v.inv_row_i := input_backward_elim.inv_row_i;
C.2. ELIMINATION CORE

v.index_i := input_backward_elim.index_i;
v.index_j := input_backward_elim.index_j;
v.best_approx := INITIAL_BEST_APPROX;
v.msb_index := msb_index;

r_i_i := to_integer(input_backward_elim.row_i(input_backward_elim.index_i));
r_i_i_halv := shift_left((shift_right(to_signed(r_i_i, r_i_i_halv'length), 1)), PRECISION_SHIFT);

-- dividing by two, then shifting up again with precision shift.

r_j_i := shift_left(resize(input_backward_elim.row_j(input_backward_elim.index_i), r_j_i'length), PRECISION_SHIFT);

-- For more precise integer division (in Vivado the rounding is always downwards)
-- r_j_i := r_j_i+r_i_i_halv;

if v.msb_index <= DIV_PRECISION then
divisor_inv_from_lut := to_integer(divisor_inv);
else
-- Using shifting approach
divisor_inv_from_lut := to_integer(divisor_inv);

-- The best approximation may be either the msb-shifted division, or the
-- msb+1 shifted division.
v.best_approx.remainder := remainders(v.msb_index);
v.best_approx.number_of_shifts := v.msb_index;

-- The best approximation to the divisor may be larger than the divisor.
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

-- This is a better approximation
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));
number_of_shifts+1;
end if;
end if;

for i in 0 to P_BANDS−1 loop
inner_product := to_signed(to_integer(input_backward_elim.row_i(i))*to_integer(r_j_i_divided), inner_product'length);

if v.msb_index <= DIV_PRECISION then
  Using lut-table
  inner_product := resize(input_backward_elim.row_i(i)*r_j_i, inner_product'length);
  shifted_down_inner_product := resize(shift_right(inner_product, PRECISION_SHIFT+DIV_PRECISION), shifted_down_inner_product'length);
  To matrix A
  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));

  inner_product := resize(input_backward_elim.inv_row_i(i)*r_j_i, inner_product'length);
  shifted_down_inner_product := resize(shift_right(inner_product, PRECISION_SHIFT+DIV_PRECISION), shifted_down_inner_product'length);
  To matrix A_inv
  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));

else
  Using shifting approach to division
  inner_product := shift_right(input_backward_elim.row_i(i)*r_j_i, v.best_approx.number_of_shifts);
  shifted_down_inner_product := resize(shift_right(inner_product, PRECISION_SHIFT), shifted_down_inner_product'length);
  To matrix A
  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));

  inner_product := shift_right(input_backward_elim.inv_row_i(i)*r_j_i, v.best_approx.number_of_shifts);
  shifted_down_inner_product := resize(shift_right(inner_product, PRECISION_SHIFT), shifted_down_inner_product'length);
  To matrix A_inv
  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));
C.2. ELIMINATION CORE

290    end if;
291    end loop;

292    -- Control signals --
293    v.write_address_odd := input_backward_elim.
294                              write_address_odd;
295    v.write_address_even := input_backward_elim.
296                              write_address_even;
297    v.flag_write_to_odd_row := input_backward_elim.
298                              flag_write_to_odd_row;
299    v.flag_write_to_even_row := input_backward_elim.
300                              flag_write_to_even_row;
301    v.state_reg := input_backward_elim.
302                              state_reg;
303    v.valid_data := input_backward_elim.
304                              valid_data;
305    v.forward_elimination_write_state := input_backward_elim.
306                              forward_elimination_write_state;
307    v.valid_data := '1';
308    end if;
309    if (reset_n = '0') then
310        v.index_i := P_BANDS−1;
311        v.index_j := P_BANDS−2;
312        v.valid_data := '0';
313    end if;
314    r_in <= v;

315    -- data
316    output_backward_elim.new_row_j <= r.
317                           row_j;
318    output_backward_elim.new_inv_row_j <= r.
319                           inv_row_j;

320    -- control
321    output_backward_elim.state_reg <= r.
322                             state_reg;
323    output_backward_elim.valid_data <= r.
324                               valid_data;
325    output_backward_elim.write_address_even <= r.
326                               write_address_even;
327    output_backward_elim.write_address_odd <= r.
328                               write_address_odd;
329    output_backward_elim.flag_write_to_even_row <= r.
330                              flag_write_to_even_row;
331    output_backward_elim.flag_write_to_odd_row <= r.
332                              flag_write_to_odd_row;
333    output_backward_elim.forward_elimination_write_state <= r.
334                        forward_elimination_write_state;
335    end process;

336    sequential_process : process (clk)
337      begin
if rising_edge(clk) then
    if clk_en = '1' then
        r <= r_in;
    end if;
end if;
end process;
end Behavioral;

C.3 BRAM SDP 18kbit

Listing C.3: 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;
          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;
beg
process (clk)
beg
    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;

C.4 Package Common types and functions

Listing C.4: Common types and functions

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;
library work;

package Common_types_and_functions is
    -- N_PIXELS is the number of pixels in the hyperspectral image
    constant N_PIXELS : integer range 0 to 628864 := 628864;  -- 578 pixels per row * 1088 rows
P_BANDS is the number of spectral bands

constant P_BANDS : integer range 0 to 100 := 12;

K is size of the kernel used in LRX.

constant K : integer;

PIXEL_DATA_WIDTH is the width of the raw input data from the HSI.

constant PIXEL_DATA_WIDTH : integer range 0 to 16 := 16;

constant NUMBER_OF_WRITES_PER_CYCLE : integer range 0 to 2 := 2;

constant BRAM_TDP_ADDRESS_WIDTH : integer range 0 to 10 := 10;

component generics

constant B_RAM_SIZE : integer := 100;

Need to be 33 bit due to updating (adding) of two 32 bit variables. Is 33 bit necessary? Precision question.

constant B_RAM_BIT_WIDTH : integer := 32;

Time from issuing write in top-level inverse to data is possible to read from BRAM:

constant B_RAM_WAIT_CLK_CYCLES : integer range 0 to 3 := 3;

constant ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA : integer range 0 to 6 := 4;

component generics

type matrix is array (natural range <>, natural range <>) of std_logic_vector(15 downto 0);

for correlation results

type matrix_32 is array (natural range <>, natural range <>) of std_logic_vector(31 downto 0);

type row_array is array (0 to P_BANDS−1) of signed(PIXEL_DATA_WIDTH∗2 −1 downto 0);

component generics

STATE_IDLE_DRIVE : std_logic_vector(2 downto 0) := "000";

STATE_FORWARD_ELIM_TRIANGULAR_FINISHED : std_logic_vector(2 downto 0) := "001";

STATE_FORWARD_ELIMINATION_FINISHED : std_logic_vector(2 downto 0) := "010";

STATE_BACKWARD_ELIMINATION_FINISHED : std_logic_vector(2 downto 0) := "011";

STATE_LAST_DIVISION_FINISHED : std_logic_vector(2 downto 0) := "111";
C.4. PACKAGE COMMON TYPES AND FUNCTIONS

constant IDLING : std_logic_vector(1 downto 0) := "00";
constant START_FORWARD_ELIMINATION : std_logic_vector(1 downto 0) := "01";
constant START_BACKWARD_ELIMINATION : std_logic_vector(1 downto 0) := "10";
constant START_IDENTITY_MATRIX_BUILDING : std_logic_vector(1 downto 0) := "11";

constant START_FORWARD_ELIM_TRIANGULAR : std_logic_vector (1 downto 0) := "10";
constant START_FORWARD_ELIM_CORE : std_logic_vector(1 downto 0) := "11";
constant STATE_FORWARD_TRIANGULAR : std_logic_vector(1 downto 0) := "10";
constant STATE_FORWARD_ELIM : std_logic_vector(1 downto 0) := "11";

type state_type is (STATE_IDLE, STATE_STORE_CORRELATION_MATRIX, STATE_FORWARD_ELIMINATION, STATE_BACKWARD_ELIMINATION, STATE_LASTDIVISION, STATE_OUTPUT_INVERSE_MATRIX);

type elimination_write_state is (STATE_IDLE, FIRST_ELIMINATION, ODD j WRITE, EVEN j WRITE, EVEN i START, ODD i START);

type forward_elimination_write_state_type is (STATE_IDLE, CHECK_DIAGONAL_ELEMENT_IS_ZERO, SWAP_ROWS, EVEN j WRITE, ODD j WRITE);

type last_division_write_state_type is (STATE_IDLE, EVEN i WRITE, ODD i WRITE);

type remainder_after_approximation_record is record
remainder : std_logic_vector(PIXEL_DATA_WIDTH∗2−1 downto 0); — For PIXEL_DATA_WIDTH of 16
number_of_shifts : integer range 0 to 31;
remainder_valid : std_logic;
end record;

type reg_state_type is record
state : state_type;
— drive : std_logic_vector(2 downto 0);
— fsm_start_signal : std_logic_vector(1 downto 0);
— inner_loop_iter_finished : std_logic;
— inner_loop_last_iter_finished : std_logic;
— start_inner_loop : std_logic;
APPENDIX C. VHDL CODE

−−−−\texttt{forward\_elim\_ctrl\_signal} : \texttt{std\_logic\_vector}\,(1\ \texttt{downto}\ 0);
−−−−\texttt{forward\_elim\_state\_signal} : \texttt{std\_logic\_vector}\,(1\ \texttt{downto}\ 0);
−−−−\texttt{flag\_forward\_core\_started} : \texttt{std\_logic};
−−−−\texttt{flag\_forward\_triangular\_started} : \texttt{std\_logic};
\texttt{end}\ \texttt{record};

\texttt{type}\ \texttt{input\_elimination\_reg\_type}\ \texttt{is}\ \texttt{record}
\begin{align*}
\texttt{row\_j} & : \texttt{row\_array}; \\
\texttt{row\_i} & : \texttt{row\_array}; \\
\texttt{row\_even} & : \texttt{row\_array}; \\
\texttt{row\_odd} & : \texttt{row\_array}; \\
\texttt{inv\_row\_even} & : \texttt{row\_array}; \\
\texttt{inv\_row\_odd} & : \texttt{row\_array}; \\
\texttt{inv\_row\_j} & : \texttt{row\_array}; \\
\texttt{inv\_row\_i} & : \texttt{row\_array}; \\
\texttt{state\_reg} & : \texttt{reg\_state\_type}; \\
\texttt{index\_i} & : \texttt{integer}\ \texttt{range}\ 0\ \text{to}\ \texttt{P\_BANDS}\ −\ 1; \\
\texttt{index\_j} & : \texttt{integer}\ \texttt{range}\ 0\ \text{to}\ \texttt{P\_BANDS}\ −\ 1; \\
\texttt{valid\_data} & : \texttt{std\_logic}; \\
\texttt{write\_address\_even} & : \texttt{integer}\ \texttt{range}\ 0\ \text{to}\ \texttt{P\_BANDS}/2\ −\ 1; \\
\texttt{write\_address\_odd} & : \texttt{integer}\ \texttt{range}\ 0\ \text{to}\ \texttt{P\_BANDS}/2\ −\ 1; \\
\texttt{read\_address} & : \texttt{integer}\ \texttt{range}\ 0\ \text{to}\ \texttt{P\_BANDS}/2\ −\ 1; \\
\texttt{flag\_write\_to\_even\_row} & : \texttt{std\_logic}; \\
\texttt{flag\_write\_to\_odd\_row} & : \texttt{std\_logic}; \\
\texttt{write\_enable\_odd} & : \texttt{std\_logic}; \\
\texttt{write\_enable\_even} & : \texttt{std\_logic}; \\
\texttt{forward\_elimination\_write\_state} & : \texttt{forward\_elimination\_write\_state\_type}; \\
\texttt{address\_row\_i} & : \texttt{integer}\ \texttt{range}\ 0\ \text{to}\ \texttt{P\_BANDS}/2\ −\ 1; \\
\texttt{address\_row\_j} & : \texttt{integer}\ \texttt{range}\ 0\ \texttt{to}\ \texttt{P\_BANDS}/2\ −\ 1; \\
\texttt{flag\_prev\_row\_i\_at\_odd\_row} & : \texttt{std\_logic}; \quad \texttt{two} \texttt{cycles} \texttt{ahead} \\
\texttt{flag\_prev\_row\_j\_at\_odd\_row} & : \texttt{std\_logic}; \quad \texttt{needed} \quad \texttt{for} \ \texttt{flip} \ \texttt{rows} \\
\texttt{flag\_start\_swapping\_rows} & : \texttt{std\_logic}; \quad \texttt{used} \ \texttt{in} \ \texttt{forward} \ \texttt{elimination} \\
\texttt{flag\_started\_swapping\_rows} & : \texttt{std\_logic}; \quad \texttt{used} \ \texttt{in} \ \texttt{flip} \ \texttt{rows} \ \texttt{and} \ \texttt{forward} \ \texttt{elimination} \\
\texttt{flag\_wrote\_swapped\_rows\_to\_BRAM} & : \texttt{std\_logic}; \\
\texttt{flag\_first\_data\_elimination} & : \texttt{std\_logic};
\end{align*}
C.4. PACKAGE COMMON TYPES AND FUNCTIONS

read_address_even : integer range 0 to P_BANDS/2 – 1;
read_address_odd : integer range 0 to P_BANDS/2 – 1;
read_enable : std_logic;
best_approx : remainder_after_approximation_record;
msb_index : integer range 0 to 31;
index_i_two_cycles_ahead : integer range 0 to P_BANDS – 1;
index_j_two_cycles_ahead : integer range 0 to P_BANDS – 1;
read_address_row_i_two_cycles_ahead : integer range 0 to P_BANDS/2 – 1;
wait_counter : integer range 0 to 3;
flag_waiting_for_bram_update : std_logic;
end record;
type inverse_top_level_reg_type is record
  row_j : row_array;
  row_i : row_array;
  inv_row_j : row_array;
  inv_row_i : row_array;
  state_reg : reg_state_type;
  index_i_two_cycles_ahead : integer range 0 to P_BANDS – 1;
  index_j_two_cycles_ahead : integer range 0 to P_BANDS – 1;
  index_i : integer range 0 to P_BANDS – 1;
  index_j : integer range 0 to P_BANDS – 1;
  valid_data : std_logic;
  write_address_even : integer range 0 to P_BANDS/2 – 1;
  write_address_odd : integer range 0 to P_BANDS/2 – 1;
  read_address_even : integer range 0 to P_BANDS/2 – 1;
  read_address_odd : integer range 0 to P_BANDS/2 – 1;
  bram_write_data_M : std_logic_vector(P_BANDS * PIXEL_DATA_WIDTH * 2 * 2 – 1 downto 0);
  bram_write_data_M_inv : 

APPENDIX C. VHDL CODE

std_logic_vector(P_BANDS*PIXEL_DATA_WIDTH*2*2 −1 downto 0);

write_enable_even :
  std_logic; — Remove?
write_enable_odd :
  std_logic; — Remove?
read_enable :
  std_logic;
writes_done_on_column :
  std_logic_vector(0 downto 0); —should
— be size log2(P_BANDS/2)downto 0. Need to edit the size
manually if
— changing P_BANDS.
flag_first_data_elimination :
  std_logic;
flag_waited_one_clk :
  std_logic;
flag_first_memory_request :
  std_logic; — between each state shift
flag_write_to_odd_row :
  std_logic; — row_j might be on both odd and
— even rows.
flag_write_to_even_row :
  std_logic; — sometimes its necessary to write
— both rows.
—^ Needed for forward elimination
elimination_write_state :
  elimination_write_state;
read_address_row_i_two_cycles_ahead :
  integer range 0 to P_BANDS/2−1;
— read address of the row i
address_row_i :
  integer range 0 to P_BANDS/2−1;
flag_prev_row_i_at_odd_row :
  std_logic; —two cycles ahead
flag_wr_row_i_at_odd_row :
  std_logic;
—*
flag_finished_sending_data_to_BRAM_one_cycle_ago :
  std_logic;
flag_finished_sending_data_to_BRAM_two_cycles_ago :
  std_logic;
flag_finished_sending_data_to_BRAM_three_cycles_ago :
  std_logic;
flag_last_read_backward_elimination :
  std_logic;
flag_first_iter_backward_elim : std_logic;
wait_counter :
  integer range 0 to 3;
C.4. PACKAGE COMMON TYPES AND FUNCTIONS

flag_waiting_for_bram_update : std_logic;
−− Needed for last division:
last_division_write_state :
  last_division_write_state_type;
counter_output_inverse_matrix : integer range 0 to P_BANDS/2−1;
end record;
type inverse_output_reg_type is record
  −− outputting two rows of the inverse matrix per cycle:
two_inverse_rows : std_logic_vector(P_BANDS∗PIXEL_DATA_WIDTH∗2−1 downto 0);
valid_data : std_logic;
address : integer range 0 to P_BANDS/2−1;
end record;
type output_forward_elimination_reg_type is record
  row_j : row_array;
  row_i : row_array;
  inv_row_j : row_array;
  inv_row_i : row_array;
  index_i : integer range 0 to P_BANDS−1;
  index_j : integer range 0 to P_BANDS−1;
  state_reg : reg_state_type;
  r_addr_next : integer range 0 to P_BANDS/2−1;
  write_address_even : integer range 0 to P_BANDS/2−1;
  write_address_odd : integer range 0 to P_BANDS/2−1;
  valid_data : std_logic;
  flag_write_to_odd_row : std_logic;  −− row_j might be on both odd and
  write_enable_even : std_logic;
  write_enable_odd : std_logic;
  flag_prev_row_i_at_odd_row : std_logic;  −− two cycles ahead
  read_address_even : integer range 0 to P_BANDS/2−1;
  read_address_odd : integer range 0 to P_BANDS/2−1;
end record;
read_enable : std_logic;
forward_elimination_write_state : forward_elimination_write_state_type;
flag_started_swapping_rows : std_logic; -- used in flip rows and forward elimination
wait_counter : integer range 0 to 3;
index_i_two_cycles_ahead : integer range 0 to P_BANDS−1;
index_j_two_cycles_ahead : integer range 0 to P_BANDS−1;
read_address_row_i_two_cycles_ahead : integer range 0 to P_BANDS−1;
end record;


type output_backward_elimination_reg_type is record
new_row_j : row_array;
new_inv_row_j : row_array;
r_addr_next : integer range 0 to P_BANDS/2−1;
write_address_even : integer range 0 to P_BANDS/2−1;
write_address_odd : integer range 0 to P_BANDS/2−1;
valid_data : std_logic;
state_reg : reg_state_type;
flag_write_to_odd_row : std_logic; -- row_j might be on both odd and
-- even rows.
flag_write_to_even_row : std_logic; -- sometimes its necessary to write -- both rows.
write_enable_even : std_logic;
write_enable_odd : std_logic;
forward_elimination_write_state : forward_elimination_write_state_type;
end record;


type input_last_division_reg_type is record
row_i : row_array;
inv_row_i : row_array;
state_reg : reg_state_type;
index_i : integer range 0 to P_BANDS−1;
flag_write_to_even_row : std_logic; -- Maximum need to write one row at the
-- time in STATE LAST DIVISION
valid_data : std_logic;
write_address_even : integer range 0 to P_BANDS/2−1;
write_address_odd : integer range 0 to P_BANDS/2−1;
--
best_approx : ;
C.4. PACKAGE COMMON TYPES AND FUNCTIONS

```plaintext
remainder_after_approximation_record;
msb_index : integer range 0 to 31;
end record;

type output_last_division_reg_type is record
new_inv_row_i : row_array;
valid_data : std_logic;
index_i : integer range 0 to P_BANDS − 1;
write_address_even : integer range 0 to P_BANDS/2 − 1;
write_address_odd : integer range 0 to P_BANDS/2 − 1;
flag_write_to_even_row : std_logic;
state_reg : reg_state_type;
end record;

constant INITIAL_BEST_APPROX :
  remainder_after_approximation_record := (
    remainder => (PIXEL_DATA_WIDTH*2−1 => '0', others => '1'),
    number_of_shifts => 0,
    remainder_valid => '0'
  );
constant DIV_PRECISION : integer range 0 to 31 := 17;

function log2( i : natural) return integer;
function sel (n : natural) return integer;
function create_identity_matrix (n : natural) return matrix_32;

end Common_types_and_functions;

package body Common_types_and_functions is
  −− Found in SmallSat project description:
  −− constant P_BANDS : integer := 100;
  constant K : integer := 0;

function log2(i : natural) return integer is
  variable temp : integer := 1;
  variable ret_val : integer := 0;
begin
  while (temp > 1) loop
    ret_val := ret_val + 1;
    temp := temp / 2;
  end loop;
  return ret_val;
end function;

function create_identity_matrix(n : natural) return matrix_32
```
is

variable M_identity_matrix : matrix_32(0 to P_BANDS−1, 0 to P_BANDS−1);

begin

M_identity_matrix := (others => (others => (others => '0')));

for i in 0 to n−1 loop

M_identity_matrix(i, i) := std_logic_vector(to_unsigned(1, 32));

end loop;

return M_identity_matrix;

end function;

function sel(n : natural) return integer is

begin

return n;

end function;

end Common_types_and_functions;

C.5 Swap rows

Listing C.5: Swap rows

library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.Common_types_and_functions.all;

-- This module is used in the forward elimination
-- It flips rows i and j if a zero is detected in row i, and row j does not
-- contain a 0 at index j

entity swap_rows_module is

port(clk : in std_logic;
reset_n : in std_logic;
clk_en : in std_logic;
input_swap_rows : in input_elimination_reg_type;
output_swap_rows : out
output_forward_elimination_reg_type
);

end swap_rows_module;

architecture Behavioral of swap_rows_module is
C.5. SWAP ROWS

153

signal r, r_in : input_elimination_reg_type;

begin

comb_process : process(input_swap_rows, r, reset_n)

variable v : input_elimination_reg_type;

begin

v := r;

case input_swap_rows.forward_elimination_write_state is

when SWAP_ROWS =>

if input_swap_rows.flag_start_swapping_rows = '1' then

v.flag_started_swapping_rows := '1';

v.flag_wrote_swapped_rows_to_BRAM := '0';

v.index_j := input_swap_rows.

index_j;

v.index_i := input_swap_rows.

index_i;

v.row_i := input_swap_rows.

row_i;

v.row_j := input_swap_rows.

row_j;

v.address_row_i := input_swap_rows.

address_row_i;

v.address_row_j := input_swap_rows.

address_row_j;

v.flag_write_to_even_row := input_swap_rows.

flag_write_to_even_row;

v.flag_write_to_odd_row := input_swap_rows.

flag_write_to_odd_row;

v.flag_prev_row_i_at_odd_row := input_swap_rows.

flag_prev_row_i_at_odd_row;

v.flag_prev_row_j_at_odd_row := not(input_swap_rows.

flag_prev_row_j_at_odd_row);

if v.row_j(v.index_j) /= 0 then

--- flip the rows, write_to_BRAM

v.row_i := v.row_j;

v.row_j := input_swap_rows.

row_i;

v.flag_wrote_swapped_rows_to_BRAM := '1';

v.flag_write_to_even_row := '1';

v.flag_write_to_odd_row := '1';

if v.flag_prev_row_i_at_odd_row = '1' then

v.write_address_odd := v.address_row_i;

v.write_address_even := v.address_row_j;

else

v.write_address_even := v.address_row_i;

v.write_address_odd := v.address_row_j;

end if;

else

--- need to check next row_j. Issue reads for two
cycles ahead

if v.index_j <= P_BANDS-3 then
    if input_swap_rows.flag_write_to_even_row = '1'
        then
            -- need to read an odd row, has already read the
                even row with the
            -- same address as this odd row
            v.read_enable := '1';
            v.read_address_even := input_swap_rows.read_address_even;
            v.read_address_odd := input_swap_rows.read_address_odd;
        else
            -- need to read an even row
            v.read_enable := '1';
            v.read_address_even := input_swap_rows.read_address_even+1;
            v.read_address_odd := input_swap_rows.read_address_odd+1;
        end if;
    end if;
else
    -- all reads has been issued. Need to wait to see
        if some of the latest
    -- rows can be swapped
    v.read_enable := '0';
end if;

if r.flag_started_swapping_rows = '1' and r.
    flag_wrote_swapped_rows_to_BRAM = '0' then
    -- need to check if index_i and index_j is at two
        even or two odd
    -- indexes. If so, the writes two BRAM will have to
        continue in two cycles.
    v.index_j := r.index_j +1;
    v.flag_prev_row_j_at_odd_row := not(r.
        flag_prev_row_j_at_odd_row);
    v.flag_write_to_even_row := not(r.
        flag_write_to_even_row);
    v.flag_write_to_odd_row := not(r.
        flag_write_to_odd_row);
    v.row_j := input_swap_rows.
        row_j; -- this is outputted directly from BRAMS.
if v.flag_prev_row_j_at_odd_row = '0' then
    -- current row_j is at an even index, need to
        update row_j address
    v.address_row_j := r.address_row_j+1;
end if;
if v.row_j(v.index_j) /= 0 then
C.5. SWAP ROWS

---flip the rows, write to BRAM

\( v\text{.}row_i := v\text{.}row_j \);
\( v\text{.}row_j := \) input_swap_rows.row.i;  ---this correct?
\( v\text{.}flag_wrote_swapped_rows_to_BRAM := '1'; \)
\( v\text{.}flag_write_to_even_row := '1'; \)
\( v\text{.}flag_write_to_odd_row := '1'; \)
if v.flag_prev_row_i_at_odd_row = '1' then
  \( v\text{.}write_address_odd := v\text{.}address_row_i; \)
  \( v\text{.}write_address_even := v\text{.}address_row_j; \)
else
  \( v\text{.}write_address_even := v\text{.}address_row_i; \)
  \( v\text{.}write_address_odd := v\text{.}address_row_j; \)
end if;
else
  --- need to read new data
  if v.index_j <= P_BANDS-3 then
    \( v\text{.}valid_data := '0'; \)
    if v.flag_write_to_even_row = '1' then
      ---need to read an odd row, has already read the even row with the
      --- same address as this odd row
      \( v\text{.}read_enable := '1'; \)
      \( v\text{.}read_address_even := v\text{.}read_address_even; \)
      \( v\text{.}read_address_odd := v\text{.}read_address_odd; \)
    else
      --- need to read an even row
      \( v\text{.}read_enable := '1'; \)
      \( v\text{.}read_address_even := v\text{.}read_address_even+1; \)
      \( v\text{.}read_address_odd := v\text{.}read_address_odd +1; \)
    end if;
  end if;
  elsif v.index_j = 0 then
    --- The loop has continued without any swap of rows
    --- The matrix is singular.
    \( v\text{.}valid_data := '1'; \)
  else
    --- all reads has been issued. Need to wait to see if some of the latest
    --- rows can be swapped
    \( v\text{.}read_enable := '0'; \)
  end if;
else
  if r.flag_wrote_swapped_rows_to_BRAM = '1' then
    --- valid_data is used to signal that the swapping is done
    --- and that the data is finished written to BRAM
    \( v\text{.}valid_data := '1'; \)
    \( v\text{.}flag_write_to_even_row := '0'; \)
v.flag_write_to_odd_row := '0';
v.read_enable := '0';
-- Start issuing reads
if r.flag_prev_row_i_at_odd_row = '0' then
  -- This means that row j two cycles ahead is at an odd index
  -- because the first row j is at an odd index
  v.read_address_odd := r.address_row_i;
  v.read_address_even := r.address_row_i;
else
  -- This means that row j two cycles ahead is at an even index
  -- because the first row j is at an even index
  v.read_address_odd := r.address_row_i;
  v.read_address_even := r.address_row_i+1;
end if;
end if;
when others =>
  v.index_i := 0;
  v.index_j := 1;
  v.valid_data := '0';
  v.address_row_i := 0;
  v.address_row_j := 1;
end case;
if (reset_n = '0') then
  v.index_i := 0;
  v.index_j := 1;
  v.valid_data := '0';
  v.address_row_i := 0;
  v.address_row_j := 1;
  v.flag_write_to_even_row := '0';
  v.flag_write_to_odd_row := '0';
  v.flag_started_swapping_rows := '0';
end if;

r_in <= v;
-- This module needs to write
-- output_forward_elimination
output_swap_rows.row_j <= r.row_j;
output_swap_rows.row_i <= r.row_i;
output_swap_rows.read_address_odd <= r.read_address_odd;
output_swap_rows.read_address_even <= r.read_address_even;
output_swap_rows.flag_write_to_odd_row <= r.flag_write_to_odd_row;
output_swap_rows.flag_write_to_even_row <= r.flag_write_to_even_row;
output_swap_rows.read_enable <= r.read_enable;
output_swap_rows.valid_data <= r.valid_data;
output_swap_rows.write_address_odd <= r.write_address_odd;
output_swap_rows.write_address_even <= r.write_address_even;
output_swap_rows.state_reg <= r.state_reg;
output_swap_rows.flag_prev_row_i_at_odd_row <= r.flag_prev_row_i_at_odd_row;

end process;

sequential_process : process (clk, clk_en)
begin
if (rising_edge(clk) and clk_en = '1') then
  r <= r_in;
end if;
end process;

end Behavioral;

C.6 ACAD inverse

Listing C.6: ACAD inverse
library IEEE;
use IEEE.STD_LOGIC_1164.all;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;

library work;
use work.Common_types_and_functions.all;

— Uncomment the following library declaration if using
— arithmetic functions with Signed or Unsigned values
— use IEEE.NUMERIC_STD.ALL;

— Uncomment the following library declaration if instantiating
— any Xilinx leaf cells in this code.
— library UNISIM;
— use UNISIM.VComponents.all;

— This entity is the top-level for computing the inverse of a
— matrix
— It is written using the two-step method, as described by Jiri
Gaisler,
entity inverse_matrix is
port (reset_n : in std_logic;
  clk_en : in std_logic;
  clk : in std_logic;
  valid : in std_logic;  — connect this
to valid_out from
— correlation module
— assumes that data are inputted row-wise, two rows at
the time
din : in std_logic_vector(P_BANDS*PIXEL_DATA_WIDTH*2*2−1 downto 0);
— increases by one for every two write to BRAM:
writes_done_on_column : in std_logic_vector(log2(P_BANDS/2) downto 0);
— outputting two and two rows of the inverse matrix
inverse_rows : out inverse_output_reg_type);
end inverse_matrix;

architecture Behavioral of inverse_matrix is
signal r, r_in : inverse_top_level_reg_type;
signal output_backward_elim : output_backward_elimination_reg_type;
C.6. ACAD INVERSE

signal output_forward_elim : output_forward_elimination_reg_type;
signal output_last_division : output_last_division_reg_type;
signal data_out_brams_M : std_logic_vector(P_BANDS * PIXEL_DATA_WIDTH*2*2-1 downto 0);
signal data_out_brams_M_inv : std_logic_vector(P_BANDS * PIXEL_DATA_WIDTH*2*2-1 downto 0);

— write address for 18kbit BRAMs storing even indexes of the matrices
signal write_address_even : integer range 0 to B_RAM_SIZE -1;
— write address for 18kbit BRAMs storing odd row indexes of the matrices
signal write_address_odd : integer range 0 to B_RAM_SIZE -1;
signal read_address_even : integer range 0 to B_RAM_SIZE -1;
signal read_address_odd : integer range 0 to B_RAM_SIZE -1;
signal write_enable_odd : std_logic := '0';
signal write_enable_even : std_logic := '0';
— for BRAMs containing the inverse matrix:
signal write_enable_inv_odd : std_logic := '0';
signal write_enable_inv_even : std_logic := '0';

— input record to the forward elimination module
signal input_forward_elimination : input_elimination_reg_type;
— input record to the elimination core
signal input_elimination : input_elimination_reg_type;
signal input_last_division : input_last_division_reg_type;
— index of the top bit of the even rows in data out from the BRAMs:
constant EVEN_ROW_TOP_INDEX : integer range 0 to P_BANDS* PIXEL_DATA_WIDTH*2-1 := P_BANDS*PIXEL_DATA_WIDTH*2-1;
— index of the topper bit of the odd rows in data out from the BRAMs:
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;

begin

gen_BRAM_18_for_storing_correlation_matrix : for i in 0 to P_BANDS-1 generate
— Generating N_BRAMS = P_BANDS BRAM 36 kbits.
— Storing matrix M in the Gauss Jordan elimination
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);

begin
APPENDIX C. VHDL CODE

-- Block ram row for even row indexes of the correlation
matrix
block_ram_even : entity work.block_ram
 generic map (
   B_RAM_SIZE => B_RAM_SIZE,
   B_RAM_BIT_WIDTH => B_RAM_BIT_WIDTH)
 port map (
   clk => clk,
   aresetn => reset_n,
   data_in => data_in_even_i,
   write_enable => write_enable_even,
   read_enable => r.read_enable,
   read_address => read_address_even,
   write_address => write_address_even,
   data_out => data_out_even_i);

-- Block ram row for odd row indexes of the correlation
matrix
block_ram_odd : entity work.block_ram
 generic map (
   B_RAM_SIZE => B_RAM_SIZE,
   B_RAM_BIT_WIDTH => B_RAM_BIT_WIDTH)
 port map (
   clk => clk,
   aresetn => reset_n,
   data_in => data_in_odd_i,
   write_enable => write_enable_odd,
   read_enable => r.read_enable,
   read_address => read_address_odd,
   write_address => write_address_odd,
   data_out => data_out_odd_i);

-- Process to control data input to BRAMs.
process (valid, r, output_last_division, output_forward_elim,
 output_backward_elim)
begin
  if (r.state_reg.state = STATE_STORE_CORRELATION_MATRIX) then
    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);
    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);
  elsif (r.state_reg.state = STATE_FORWARD_ELIMINATION) then
    if output_forward_elim.forward_elimination_write_state =
      SWAP_ROWS then
      if output_forward_elim.flag_started_swapping_rows =
        '1' then
        if output_forward_elim.flag_prev_row_i_at_odd_row =
          -- Further code...
      end if;
    end if;
  end if;
end process;
'1' then
    -- row i at odd index
    data_in_even_i <= std_logic_vector(output_forward_elim.row_j(i));
data_in_odd_i <= std_logic_vector(output_forward_elim.row_i(i));
else
    data_in_even_i <= std_logic_vector(output_forward_elim.row_i(i));
data_in_odd_i <= std_logic_vector(output_forward_elim.row_j(i));
end if;
else
    data_in_odd_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
data_in_even_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
end if;
-- output_backward_elim in state EVEN_j or ODD_j
elsif output_backward_elim.flag_write_to_even_row = '1' and output_backward_elim.valid_data = '1' then
    -- row_j is at even row
    data_in_even_i <= std_logic_vector(output_backward_elim.new_row_j(i));
data_in_odd_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
elsif output_backward_elim.flag_write_to_odd_row = '1' and output_backward_elim.valid_data = '1' then
    -- row_j is at odd row
    data_in_odd_i <= std_logic_vector(output_backward_elim.new_row_j(i));
data_in_even_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
else
    data_in_odd_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
data_in_even_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
end if;
elsif (r.state_reg.state = STATE_BACKWARD_ELIMINATION) then
    if (output_backward_elim.valid_data = '1') then
        -- Received data from backward elimination.
        if (output_backward_elim.flag_write_to_odd_row = '1') then
            -- the j-indexed row is an odd row of the matrix
            data_in_odd_i <= std_logic_vector(output_backward_elim.new_row_j(i));
data_in_even_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
elsif (output_backward_elim.flag_write_to_even_row = '1') then
    — the j-indexed row is an even row of the matrix
    data_in_even_i <= std_logic_vector(output_backward_elim.new_row_j(i));
    data_in_odd_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
else
    data_in_odd_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
    data_in_even_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
end if;
else
    data_in_odd_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
    data_in_even_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
end if;
elsif (r.state_reg.state = STATE_LAST_DIVISION) then
    data_in_odd_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
    data_in_even_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
else
    data_in_odd_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
    data_in_even_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
end if;
end process;
data_out_brams_M(PIXEL_DATA_WIDTH*2−1 + i∗PIXEL_DATA_WIDTH*2 downto i∗PIXEL_DATA_WIDTH*2) <= data_out_even_i;
— even row
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;
— odd row
end generate;
gen_BRAM_18_for_storing_inv_correlation_matrix : for i in 0 to P_BANDS−1 generate
    — Generating N_BRAMS = P_BANDS BRAM 36 kbits.
    — Storing the inverse matrix in the Gauss–Jordan Elimination
    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);
begin

-- Block ram row for even indexes of the inverse matrix
block_ram_even : entity work.block_ram
  generic map (  
    B_RAM_SIZE    => B_RAM_SIZE,  
    B_RAM_BIT_WIDTH => B_RAM_BIT_WIDTH)
  port map (    
    clk => clk ,  
    aresetn => reset_n ,  
    data_in => inv_data_in_even_i ,  
    write_enable => write_enable_inv_even ,  
    read_enable => r.read_enable ,  
    read_address => read_address_even ,  
    write_address => write_address_even ,  
    data_out => inv_data_out_even_i) ;

-- Block ram row for odd indexes of the inverse matrix
block_ram_odd : entity work.block_ram
  generic map (  
    B_RAM_SIZE    => B_RAM_SIZE,  
    B_RAM_BIT_WIDTH => B_RAM_BIT_WIDTH)
  port map (    
    clk => clk ,  
    aresetn => reset_n ,  
    data_in => inv_data_in_odd_i ,  
    write_enable => write_enable_inv_odd ,  
    read_enable => r.read_enable ,  
    read_address => read_address_odd ,  
    write_address => read_address_odd ,  
    data_out => inv_data_out_odd_i) ;

-- Process to control data input to BRAMs.
process(valid , r , output_forward_elim , output_backward_elim ,
        output_last_division)
begin
  if (r.state_reg.state = STATE_STORE_CORRELATION_MATRIX)
    then
      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) ;
      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) ;
    elsif r.state_reg.state = STATE_FORWARD_ELIMINATION then
      if output_forward_elim.forward_elimination_write_state = 
        SWAP_ROWS and output_forward_elim validoa = '1'
        then
        -- do nothing actually
        -- Set data in to zero. Should not overwrite data
        anyway, write enable
    end if;
end process;
APPENDIX C. VHDL CODE

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
239      elsif output_backward_elim.flag_write_to_even_row = '1' and output_backward_elim.valid_data = '1' then
240        -- row_j is at even row
241        inv_data_in_even_i <= std_logic_vector(output_backward_elim.new_inv_row_j(i));
242
243      elsif (r.state_reg.state = STATE_BACKWARD_ELIMINATION)
244                  then
245                      if (output_backward_elim.valid_data = '1') then
246                          -- Received data from backward elimination.
247                          if (output_backward_elim.flag_write_to_odd_row = '1') then
248                              -- the j-indexed row is an odd row of the matrix
249                              inv_data_in_odd_i <= std_logic_vector(output_backward_elim.new_inv_row_j(i));
250                              inv_data_in_even_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
251                              elsif(output_backward_elim.flag_write_to_even_row = '1') then
252                                  -- the j-indexed row is an even row of the matrix
253                                  inv_data_in_even_i <= std_logic_vector(output_backward_elim.new_inv_row_j(i));
254                                  inv_data_in_odd_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
255                              else
256
257                      end if;
258                      elsif(output_backward_elim.flag_write_to_odd_row = '1') then
259                          -- row_j is at odd row
260                          inv_data_in_odd_i <= std_logic_vector(output_backward_elim.new_inv_row_j(i));
261
262                      elsif(output_backward_elim.flag_write_to_even_row = '1') then
263                          -- row_j is at even row
264                          inv_data_in_even_i <= std_logic_vector(output_backward_elim.new_inv_row_j(i));
265                          inv_data_in_odd_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
266                      else
267
268                      end if;
269
270                else
271
272            end if;
273
274            elsif(output_backward_elim.flag_write_to_odd_row = '1') then
275                -- row_j is at odd row
276                inv_data_in_odd_i <= std_logic_vector(output_backward_elim.new_inv_row_j(i));
277                inv_data_in_even_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
278
279                elsif(output_backward_elim.flag_write_to_even_row = '1') then
280                    -- row_j is at even row
281                    inv_data_in_even_i <= std_logic_vector(output_backward_elim.new_inv_row_j(i));
282                    inv_data_in_odd_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
283                else
284
285                end if;
286
287                elsif(output_backward_elim.flag_write_to_odd_row = '1') then
288                    -- row_j is at odd row
289                    inv_data_in_odd_i <= std_logic_vector(output_backward_elim.new_inv_row_j(i));
290                    inv_data_in_even_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
291
292                else
293
294            end if;
295
296            elsif(output_backward_elim.flag_write_to_even_row = '1') then
297                -- row_j is at even row
298                inv_data_in_even_i <= std_logic_vector(output_backward_elim.new_inv_row_j(i));
299                inv_data_in_odd_i <= std_logic_vector(to_signed(0, PIXEL_DATA_WIDTH*2));
300
301            else
302
303        end if;
C.6. ACAD INVERSE

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

<= inv_data_out_even_i;

data_out_brms_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) <=

end generate;

top_forward_elimination_1 : entity work.
top_forward_elimination

port map (clk => clk ,
reset_n => reset_n ,
clk_en => clk_en ,
input_forward_elimination => input_forward_elimination ,
output_forward_elimination => output_forward_elimination);

backward_elim_core is used by both forward_elimination and
backward_elimination.

elimination_core_1 : entity work.backward_elim_core

port map (clk => clk ,
reset_n => reset_n ,
clk_en => clk_en ,
input_backward_elim => input_backward_elim ,
output_backward_elim => output_backward_elim);

top_last_division_1 : entity work.top_last_division

port map (clk => clk ,
reset_n => reset_n ,
clk_en => clk_en ,
ininput_last_division => input_last_division ,
output_last_division => output_last_division);

just_for_test : process(data_out_brms_M, r)

variable row_even, row_odd, inv_row_even, inv_row_odd : row_array;

begin

for i in 0 to P_BANDS−1 loop

row_odd(i) := signed(data_out_brms_M(i*PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2−1 downto i*PIXEL_DATA_WIDTH*2));

row_even(i) := signed(data_out_brms_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));

the odd row

inv_row_even(i) := signed(data_out_brms_M_inv(i*PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2−1 downto i*PIXEL_DATA_WIDTH*2));

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));

end loop;
end process;

control addresses and control inputs to BRAMs
control_addresses_and_control_BRAM : process (r, output_backward_elim, output_last_division, output_forward_elim)
begin
if (r.state_reg.state = STATE_STORE_CORRELATION_MATRIX) then
write_address_even <= r.write_address_even;
write_address_odd <= r.write_address_odd;
read_address_even <= r.read_address_even;
read_address_odd <= r.read_address_odd;
write_enable_even <= '1';
write_enable_odd <= '1';
write_enable_inv_even <= '1';
write_enable_inv_odd <= '1';
elsif (r.state_reg.state = STATE_FORWARD_ELIMINATION) then
-- Set read addresses to output from top elimination
read_address_even <= output_forward_elim.read_address_even
read_address_odd <= output_forward_elim.read_address_odd;
if output_forward_elim.forward_elimination_write_state = SWAP_ROWS then
write_address_even <= output_forward_elim.
write_address_even;
write_address_odd <= output_forward_elim.
write_address_odd;
write_enable_even <= output_forward_elim.
flag_write_to_even_row;
write_enable_odd <= output_forward_elim.
flag_write_to_odd_row;
write_enable_inv_even <= '0';
write_enable_inv_odd <= '0';
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
write_address_even <= output_backward_elim.
write_address_even;
write_address_odd <= output_backward_elim.
write_address_odd;
write_enable_even <= output_backward_elim.
flag_write_to_even_row;
write_enable_odd <= output_backward_elim.
flag_write_to_odd_row;
else
write_address_even <= output_backward_elim.
write_address_odd <= output_backward_elim.
write_enable_even <= output_backward_elim.
write_enable_odd <= output_backward_elim.
end if;
else
end if;
end if;
end process;
write_enable_inv_even <= output_backward_elim.flag_write_to_even_row;
write_enable_inv_odd <= output_backward_elim.flag_write_to_odd_row;

else
write_enable_even <= '0';
write_enable_odd <= '0';
write_enable_inv_even <= '0';
write_enable_inv_odd <= '0';
write_address_even <= 0;
write_address_odd <= 0;
end if;

elsif (r.state_reg.state = STATE_BACKWARD_ELIMINATION) then
read_address_even <= r.read_address_even;
read_address_odd <= r.read_address_odd;
if (output_backward_elim.valid_data = '1') then
— Received data from backward elimination.
if (output_backward_elim.flag_write_to_odd_row = '1') then
— the j-indexed row is an odd row of the matrix
write_enable_inv_odd <= '1';
write_enable_inv_even <= '0';
write_enable_odd <= '1';
write_enable_even <= '0';
write_address_even <= 0;
write_address_odd <= output_backward_elim.write_address_odd;
elif (output_backward_elim.flag_write_to_even_row = '1') then
— the j-indexed row is an even row of the matrix
write_enable_inv_odd <= '0';
write_enable_inv_even <= '1';
write_enable_odd <= '0';
write_enable_even <= '1';
write_address_even <= output_backward_elim.write_address_even;
write_address_odd <= 0; — To avoid latches
else
write_enable_inv_odd <= '0';
write_enable_inv_even <= '0';
write_enable_odd <= '0';
write_enable_even <= '0';
write_address_even <= 0;
write_address_odd <= 0;
end if;
else
write_enable_inv_odd <= '0';
write_enable_inv_even <= '0';
write_enable_even <= '0';
write_enable_odd <= '0';
write_address_even <= 0;
write_address_odd <= 0;  — To avoid latches
end if;
elsif r.state_reg.state = STATE_LAST_DIVISION then
read_address_odd <= r.read_address_odd;
read_address_even <= r.read_address_even;
if output_last_division.valid_data = '1' then
write_enable_inv_odd <= not(output_last_division.
  flag_write_to_even_row);
write_enable_inv_even <= output_last_division.
  flag_write_to_even_row;
write_enable_odd <= not(output_last_division.
  flag_write_to_even_row);
write_enable_even <= output_last_division.
  flag_write_to_even_row;
write_address_even <= output_last_division.
  write_address_even;
write_address_odd <= output_last_division.
else
write_enable_inv_even <= '0';
write_enable_inv_odd <= '0';
write_enable_even <= '0';
write_enable_odd <= '0';
write_address_even <= 0;
write_address_odd <= 0;
end if;
elsif r.state_reg.state = STATE_OUTPUT_INVERSE_MATRIX then
read_address_even <= r.read_address_even;
read_address_odd <= r.read_address_odd;
write_enable_inv_even <= '0';
write_enable_inv_odd <= '0';
write_enable_even <= '0';
write_enable_odd <= '0';
write_address_even <= 0;
write_address_odd <= 0;
else
read_address_even <= 0;
read_address_odd <= 0;
write_enable_inv_even <= '0';
write_enable_inv_odd <= '0';
write_enable_even <= '0';
write_enable_odd <= '0';
write_address_even <= 0;
write_address_odd <= 0;
end if;
end process;
−− control inputs to elimination processes and last division
APPENDIX C. VHDL CODE

control_input_to_elimination : process (r, output_backward_elim, output_last_division, output_forward_elim, data_out_brams_M_inv, data_out_brams_M)
begin
if (r.valid_data = '1') then
  if (r.state_reg.state = STATE_FORWARD_ELIMINATION) then
    -- In state forward elimination the reads and writes are issued from
    -- top_forward_elimination, not from top-inverse
    input_forward_elimination.state_reg <= r.state_reg;
    input_forward_elimination.valid_data <= '1';
    input_forward_elimination.flag_first_data_elimination <= r.flag_first_data_elimination;
    input_forward_elimination[row_i] <= ((others => (others => '0')));
    input_forward_elimination.inv_row_i <= ((others => (others => '0')));
    input_forward_elimination.state_reg.state <= STATE_IDLE;
  end if;
  input_last_division.row_i <= 0;
  input_last_division.valid_data <= '0';
  input_last_division.flag_write_to_even_row <= '0';
  input_last_division.write_address_even <= 0;
  input_last_division.write_address_odd <= 0;
  for i in 0 to P_BANDS-1 loop
    input_forward_elimination[row_even(i)] <= signed(
data_out_brams_M(i*PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 downto i*PIXEL_DATA_WIDTH+1));
    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+1));
    input_forward_elimination.inv_row_even(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+1));
    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+1));
  end loop;
end 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
APPENDIX C. VHDL CODE

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

r . index_i ;
input_elimination . index_j <=
r . index_j ;
input_elimination . inv_row_j <=
r . inv_row_j ;
input_elimination . inv_row_i <=
r . inv_row_i ;
input_elimination . valid_data <=
'0 ';  
input_elimination . state_reg <=
r . state_reg ;
input_elimination . write_address_even <=
r . write_address_even ;
input_elimination . write_address_odd <=
r . write_address_odd ;
input_elimination . flag_write_to_even_row <=
'0 ';  
input_elimination . flag_write_to_odd_row <=
'0 ';  
input_elimination . forward_elimination_write_state <=
output_forward_elim.forward_elimination_write_state ;
end for ;

else

for i in 0 to P_BANDS−1 loop

input_forward_elimination . row_even( i ) <= to_signed
( 0 , PIXEL_DATA_WIDTH+2 ) ;
input_forward_elimination . row_odd( i ) <= to_signed
( 0 , PIXEL_DATA_WIDTH+2 ) ;
input_forward_elimination . inv_row_even( i ) <= to_signed
( 0 , PIXEL_DATA_WIDTH+2 ) ;
input_forward_elimination . inv_row_odd( i ) <= to_signed
( 0 , PIXEL_DATA_WIDTH+2 ) ;
end loop ;

else

input_last_division . row_i <=
(( others => ( others => '0 ') )) ;
input_last_division . inv_row_i <=
(( others => ( others => '0 ') )) ;
input_last_division . state_reg . state <=
STATE_IDLE;
input_last_division . index_i <=
0 ;
input_last_division . valid_data <=
'0 ';  
input_last_division . flag_write_to_even_row <=
C.6. ACAD INVERSE

175

'0';
input_last_division.write_address_even <= 0;
input_last_division.write_address_odd <= 0;

-- set input to elimination core
input_elimination.row_j <= r.row_j;
input_elimination.row_i <= r.row_i;
input_elimination.index_i <= r.index_i;
input_elimination.index_j <= r.index_j;
input_elimination.inv_row_j <= r.inv_row_j;
input_elimination.inv_row_i <= r.inv_row_i;
input_elimination.valid_data <= '0';
input_elimination.state_reg.state <= STATE_IDLE;
input_elimination.write_address_even <= r.write_address_even;
input_elimination.write_address_odd <= r.write_address_odd;
input_elimination.flag_write_to_even_row <= '0';
input_elimination.flag_write_to_odd_row <= '0';
input_elimination.forward_elimination_write_state <= output_forward_elim.forward_elimination_write_state;

-- set input to forward_elimination
input_forward_elimination.valid_data <= '0';
input_forward_elimination.state_reg <= r.state_reg;
input_forward_elimination.flag_first_data_elimination <= '0';
for i in 0 to P_BANDS-1 loop
  input_forward_elimination.row_even(i) <= to_signed (0, PIXEL_DATA_WIDTH*2);
  input_forward_elimination.row_odd(i) <= to_signed (0, PIXEL_DATA_WIDTH*2);
  input_forward_elimination.inv_row_even(i) <= to_signed (0, PIXEL_DATA_WIDTH*2);
  input_forward_elimination.inv_row_odd(i) <= to_signed (0, PIXEL_DATA_WIDTH*2);
end loop;
end if;
```vhdl
else
  -- set input to forward_elimination
  input_forward_elimination.valid_data <= '0';
  input_forward_elimination.state_reg <= r.state_reg;
  input_forward_elimination.flag_first_data_elimination <= '0';
  for i in 0 to P_BANDS-1 loop
    input_forward_elimination.row_even(i) <= to_signed(0, PIXEL_DATA_WIDTH*2);
    input_forward_elimination.row_odd(i) <= to_signed(0, PIXEL_DATA_WIDTH*2);
    input_forward_elimination.inv_row_even(i) <= to_signed(0, PIXEL_DATA_WIDTH*2);
    input_forward_elimination.inv_row_odd(i) <= to_signed(0, PIXEL_DATA_WIDTH*2);
  end loop;
  -- set input to elimination core
  input_elimination.row_j <= r.row_j;
  input_elimination.row_i <= r.row_i;
  input_elimination.index_i <= r.index_i;
  input_elimination.index_j <= r.index_j;
  input_elimination.inv_row_j <= r.inv_row_j;
  input_elimination.inv_row_i <= r.inv_row_i;
  input_elimination.valid_data <= '0';
  input_elimination.state_reg.state <= STATE_IDLE;
  input_elimination.write_address_even <= r.write_address_even;
  input_elimination.write_address_odd <= r.write_address_odd;
  input_elimination.flag_write_to_even_row <= '0';
  input_elimination.flag_write_to_odd_row <= '0';
  input_elimination.forward_elimination_write_state <= output_forward_elim.forward_elimination_write_state;
  -- Input to last division
  input_last_division.row_i <= ((others => (others => '0')));
  input_last_division.inv_row_i <= ((others => (others => '0')));
  input_last_division.state_reg.state <= STATE_IDLE;
  input_last_division.index_i <= 0;
```

C.6. ACAD INVERSE

```
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   comb : process (reset_n, valid, r, data_out_brams_M_inv,
667               data_out_brams_M, output_forward_elim, output_backward_elim
668               , output_last_division, din, writes_done_on_column,
669               write_address_odd, write_address_even) −− combinatorial
670   process
671     variable v : inverse_top_level_reg_type;
672   begin
673     v := r;
674     case v.state_reg.state is
675       when STATE_IDLE =>
676         v.read_enable := '0';
677         v.flag_write_to_even_row := '0';
678         v.flag_write_to_odd_row := '0';
679         v.valid_data := '0';
680         if (valid = '1') then
681           v.valid_data
682             := '1';
683           v.state_reg.state
684             := STATE_STORE_CORRELATION_MATRIX;  −− Set write address to
685             BRAMS
686         v.write_address_even
687             := 0;
688         v.read_address_odd
```
APPENDIX C. VHDL CODE

:= 0;
v.read_address_even := 0;

:= '1';
v.flag_write_to_odd_row := '1';
v.flag_write_to_even_row := '1';
v.read_enable := '1';
v.bram_write_data_M := din;

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

:= (others => '0');
v.bram_write_data_M_inv := (others => '0');
v.bram_write_data_M_inv((to_integer(unsigned(r.
writes_done_on_column))*2)*PIXEL_DATA_WIDTH*2) := '1';

identity matrix

:= 0;
v.flag_waiting_for_bram_update := '0';

end if;

— need to wait until valid data on all

when STATE_STORE_CORRELATION_MATRIX =>

— SET BRAM to write

input data

:= writes_done_on_column;
v.write_address_even := r.write_address_even +1;
v.write_address_odd := r.write_address_odd +1;
v.read_address_odd := 0;
v.read_address_even := 0;
v.read_enable := '1';
v.bram_write_data_M := din;
v.bram_write_data_M_inv := (others => '0');

v.bram_write_data_M_inv((to_integer(unsigned(writes_done_on_column)) * 2) * PIXEL_DATA_WIDTH * 2) := '1';

v.bram_write_data_M_inv((to_integer(unsigned(writes_done_on_column)) * 2 + 1) * PIXEL_DATA_WIDTH * 2) := '1';

v.flag_waiting_for_bram_update := '0';

if (to_integer(unsigned(r.writes_done_on_column)) +1 < P_BANDS/2) then
  v.bram_write_data_M_inv((to_integer(unsigned(writes_done_on_column)) * 2 + 1) * PIXEL_DATA_WIDTH * 2) := '1';
  the identity matrix
  v.bram_write_data_M_inv((to_integer(unsigned(writes_done_on_column)) * 2) * PIXEL_DATA_WIDTH * 2) := '1';
end if;

if to_integer(unsigned(r.writes_done_on_column)) = P_BANDS/2-1 then
180 APPENDIX C. VHDL CODE

--- in BRAM before starting to edit it.

v.read_enable := '1';
v.read_address_even := 0;
v.read_address_odd := 0;
v.state_reg.state := STATE_FORWARD_ELIMINATION;
v.write_enable_even := '0';
v.write_enable_odd := '0';
v.wait_counter := 0;
v.flag_last_read_backward_elimination := '0';
v.flag_first_data_elimination := '1';
v.valid_data := '1';

end if;

if valid = '0' then
  v.state_reg.state := STATE_IDLE;
  v.state_reg.drive := STATE_IDLE_DRIVE;
end if;

when STATE_FORWARD_ELIMINATION =>
  -- Set first memory_request?
  -- Set write_state?
  v.flag_first_data_elimination := '0';
  if output_forward_elim.index_j = P_BANDS-1 and
     output_forward_elim.index_i = P_BANDS-2 then
    -- finished forward elimination
    v.state_reg.state := STATE_BACKWARD_ELIMINATION;
    v.flag_first_iter_backward_elim := '1';
    -- Request data for BACKWARD elimination
    v.read_address_even := P_BANDS/2-1;  --
    read toppermost address, contains
    v.read_address_odd := P_BANDS/2-1;  --row P_BANDS-1 and
    P_BANDS-2

end if;
end if;
when STATE_BACKWARD_ELIMINATION \Rightarrow
if (r.flag_first_iter_backward_elim = '1') then
    -- Read first data from BRAMs
    v.write_address_even := P_BANDS/2−1; -- first write will happen
    v.write_address_odd := P_BANDS/2−1;
    v.read_enable := '1';
    v.write_enable_even := '0';
    v.write_enable_odd := '0';
    v.flag_first_data_elimination := '0';
    -- v.flag_waited_one_clk := '0';
    v.flag_first_memory_request := '1';
    v.index_j_two_cycles_ahead := P_BANDS−2;
    v.index_i_two_cycles_ahead := P_BANDS−1;
    v.read_address_row_i_two_cycles_ahead := P_BANDS/2−1;
    v.read_address_even := P_BANDS/2−1;
    v.read_address_odd := P_BANDS/2−1;
    v.address_row_i := P_BANDS/2−1;
    v.flag_finished_sending_data_to_BRAM_one_cycle_ago := '0';
    v.flag_finished_sending_data_to_BRAM_two_cycles_ago := '0';
    v.flag_wr_row_i_at_odd_row := '1';
    v.flag_prev_row_i_at_odd_row := '1';
    v.flag_first_iter_backward_elim := '0';
    v.wait_counter := 0;
    v.flag_waiting_for_bram_update := '0';
    v.flag_last_read_backward_elimination := '0';
end if;

if (r.flag_first_memory_request = '1') then
    v.flag_first_memory_request := '0';
    -- v.flag_waited_one_clk := '1';
    v.index_j_two_cycles_ahead := r.
        index_j_two_cycles_ahead - 1;
    v.read_address_even := r.read_address_even - 1;
    -- need to read an odd row
    v.read_address_odd := r.read_address_odd - 1;
    v.flag_first_data_elimination := '1';
end if;

if (r.flag_waited_one_clk = '1') then
    -- v.flag_first_data_elimination := '1'; — the next
    clock cycle the BRAM
    — will have the
    correct output, for
    — — the first input of the inverse matrix
    — v.flag_waited_one_clk := '0';
    — if(r.index_j_two_cycles_ahead-1 >= 0) then
    — — need to read an even row, do not change read
    address
    — v.index_j_two_cycles_ahead := r.
        index_j_two_cycles_ahead - 1;
    — end if;
    — end if;
    if (r.flag_first_data_elimination = '1') then — —
        received the first
        input_data to backward elimination from BRAM
    v.state_reg fsm_start_signal :=
        START_BACKWARD_ELIMINATION;
    — must set the flag low again
    v.flag_first_data_elimination := '0';
    for i in 0 to P_BANDS-1 loop
        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));
        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));
        — the odd row
        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));
        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 +
C.6. ACAD INVERSE

```vhdl
EVEN_ROW_TOP_INDEX+1));
end loop;
v.index_i := P_BANDS−1;
v.index_j := P_BANDS−2;
v.address_row_i := P_BANDS/2−1;
v.valid_data := '1';
-- The first written j-row will always be at an even row.
v.flag_write_to_even_row := '1';
v.flag_write_to_odd_row := '0';
v.write_enable_even := '1';
v.write_enable_odd := '0';
v.write_address_even := P_BANDS/2−1;
v.write_address_odd := P_BANDS/2−1;
v.flag_wr_row_i_at_odd_row := '1';
v.elimation_write_state := ODD_j_WRITE;
-- read new data
if (r.read_address_odd >= 0 and r.
   index_j_two_cycles_ahead >= 1) then
   -- need to read an even row
   v.read_address_odd := r.read_address_odd;
   v.read_address_even := r.read_address_even;
   v.index_j_two_cycles_ahead := r.
      index_j_two_cycles_ahead−1;
elsif r.index_j_two_cycles_ahead < 1 then
   -- new i or finished, update
   if r.index_i_two_cycles_ahead >= 2 then
      v.index_i_two_cycles_ahead := r.
         index_i_two_cycles_ahead−1;
      v.index_j_two_cycles_ahead := r.
         index_i_two_cycles_ahead−1;
      if r.flag_prev_row_i_at_odd_row = '1' then
         -- next row_i will be located in an even indexed row
         v.read_address_even := r.
            read_address_row_i_two_cycles_ahead;
         v.read_address_odd := r.
            read_address_row_i_two_cycles_ahead−1;
         v.read_address_row_i_two_cycles_ahead := r.
            read_address_row_i_two_cycles_ahead;
         v.flag_prev_row_i_at_odd_row := '0';
      else
         -- next row_i will be located in an odd indexed row
         v.read_address_odd := r.
            read_address_row_i_two_cycles_ahead−1;
         v.read_address_even := r.
            read_address_row_i_two_cycles_ahead−1;
         v.read_address_row_i_two_cycles_ahead := r.
            read_address_row_i_two_cycles_ahead−1;
      end if;
   else
      -- next row_i will be located in an even indexed row
      v.read_address_even := r.
         read_address_row_i_two_cycles_ahead;
      v.read_address_odd := r.
         read_address_row_i_two_cycles_ahead−1;
      v.read_address_row_i_two_cycles_ahead := r.
         read_address_row_i_two_cycles_ahead;
      v.flag_prev_row_i_at_odd_row := '0';
   end if;
end if;
```

APPENDIX C. VHDL CODE

v.flag_prev_row_i_at_odd_row := '1';
end if;
end if;
end if;
end if;
end if;
case r.elimination_write_state is
when ODD_j_WRITE =>
if r.flag_waiting_for_bram_update = '0' then
v.flag_write_to_even_row := '0';
v.flag_write_to_odd_row := '1';
/* row_j is outputted from odd BRAMs (located at higher end of output)
for i in 0 to P_BANDS-1 loop
v.row_j(i) := signed(data_out_brams_M(i*PIXEL_DATA_WIDTH +
EVEN_ROW_TOP_INDEX downto i*PIXEL_DATA_WIDTH*
2 + EVEN_ROW_TOP_INDEX+1));
v.inv_row_j(i) := signed(data_out_brams_M_inv(i*PIXEL_DATA_WIDTH +
EVEN_ROW_TOP_INDEX downto i*PIXEL_DATA_WIDTH*
2 + EVEN_ROW_TOP_INDEX+1));
end loop;
v.write_enable_even := '0';
v.write_enable_odd := '1';
v.index_j := r.index_j -1;
if r.index_j >= 1 then
v.write_address_odd := r.write_address_odd -1;
v.write_address_even := r.write_address_even -1;
end if;
end if;
-- do not really understand how the -2 got in this if... check
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
/* Need to wait for the row to update before reading it.
v.wait_counter := r.wait_counter +1;
v.flag_waiting_for_bram_update := '1';
else
v.flag_waiting_for_bram_update := '0';
v.wait_counter := 0;
if (v.index_j >= 1) then
v.index_j := r.index_j -1;
v.elimination_write_state := EVEN_j_WRITE;
elsif v.index_j < 1 then
v.index_i := r.index_i -1;
--v.index_j := r.index_i -2;
if (r.flag_wr_row_i_at_odd_row = '0') then
  v.elimination_write_state := ODD_i_START;
else
  v.elimination_write_state := EVEN_i_START;
end if;
end if;

-- read new data. Data need to be read two clock cycles in advance
if (r.read_address_odd >= 1 and r.
  index_j_two_cycles_ahead >= 2) then
  -- need to read an odd row
  v.read_address_odd := r.read_address_odd
  - 1;
  v.read_address_even := r.
  read_address_even - 1;
  v.index_j_two_cycles_ahead := r.
  index_j_two_cycles_ahead - 1;
elsif v.index_j < 2 then
  -- new i, update
  if r.index_i_two_cycles_ahead >= 2 then
    v.index_i_two_cycles_ahead := r.
    index_i_two_cycles_ahead - 1;
    v.index_j_two_cycles_ahead := r.
    index_i_two_cycles_ahead - 2;
    if r.flag_prev_row_i_at_odd_row = '1' then
      -- next row i will be located in an even indexed row
      v.read_address_even := r.
      address_row_i;
      v.read_address_odd := r.
      address_row_i - 1;
      v.read_address_row_i_two_cycles_ahead := r.
      read_address_row_i_two_cycles_ahead;
      v.flag_prev_row_i_at_odd_row := '0';
    else
      -- next row i will be located in an odd indexed row
      v.read_address_odd := r.
      read_address_row_i_two_cycles_ahead - 1;
      v.read_address_even := r.
      read_address_row_i_two_cycles_ahead - 1;
      v.read_address_row_i_two_cycles_ahead := r.
      read_address_row_i_two_cycles_ahead - 1;
      v.flag_prev_row_i_at_odd_row := '1';
    end if;
  end if;
  end if;
end if;
when EVEN_j_WRITE =>
  if r.flag_waiting_for_bram_update = '0' then
    for i in 0 to P_BANDS-1 loop
      -- data is located in the even part of the output from BRAM
      v.row_j(i) := signed(data_out_brams_M(i * PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1 downto i*PIXEL_DATA_WIDTH*2));
      v.inv_row_j(i) := signed(data_out_brams_M_inv(i * PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2-1 downto i*PIXEL_DATA_WIDTH*2));
    end loop;
    v.flag_write_to_even_row := '1';
    v.flag_write_to_odd_row := '0';
    v.write_address_even := r.write_address_even;
    v.write_address_odd := r.write_address_odd;
    v.write_enable_even := '1';
    v.write_enable_odd := '0';
    v.index_j := r.index_j - 1;
    v.index_i := r.index_i;
  end if;
  if v.index_j <= 1 and r.index_j_two_cycles_ahead >= 1 and v.index_j < B_RAM_WAIT_CLK_CYCLES and r.wait_counter < B_RAM_WAIT_CLK_CYCLES-(r.index_j_two_cycles_ahead - v.index_j) then
    v.wait_counter := r.wait_counter + 1;
    v.flag_waiting_for_bram_update := '1';
  else
    v.wait_counter := 0;
    v.flag_waiting_for_bram_update := '0';
    if (v.index_j >= 2) then
      v.elimination_write_state := ODD_j_WRITE;
    elsif v.index_j < 2 then
      if (r.flag_wr_row_i_at_odd_row = '0') then
        v.elimination_write_state := ODD_i_START;
      else
        v.elimination_write_state := EVEN_i_START;
      end if;
    end if;
  end if;
  -- read new data
  if (r.read_address_odd >= 1 and v.index_j >= 1) then
    if r.flag_last_read_backward_elimination = '0' then
      if (r.read_address_odd >= 0 and r.index_j_two_cycles_ahead >= 1) then
        -- need to read an even row("two clock cycles ahead")
--- Even and odd read addresses will be equal in backward elimination except for when reading the first read for an even indexed i.

```vhdl
v.read_address_odd := r.read_address_odd;
```

```vhdl
v.read_address_even := r.read_address_even;
```

```vhdl
v.read_address_even := r.read_address_odd;
```

```vhdl
v.index_j_two_cycles_ahead := r.index_j_two_cycles_ahead - 1;
```

```vhdl
elsif v.index_j < 2 then
--- new i or finished all necessary reads, update
if v.index_i >= 2 then
```vhdl
v.index_i_two_cycles_ahead := r.index_i_two_cycles_ahead - 1;
```

```vhdl
v.index_j_two_cycles_ahead := r.index_i_two_cycles_ahead - 2;
```

```vhdl
if r.flag_prev_row_i_at_odd_row = '1' then
--- next row i will be located in an even indexed row
```vhdl
v.read_address_even := r.read_address_row_i_two_cycles_ahead;
```

```vhdl
v.read_address_odd := r.read_address_row_i_two_cycles_ahead - 1;
```

```vhdl
v.read_address_row_i_two_cycles_ahead := r.read_address_row_i_two_cycles_ahead;
```

```vhdl
v.flag_prev_row_i_at_odd_row := '0';
```

```vhdl
else
--- next row i will be located in an odd indexed row
```

```vhdl
v.read_address_odd := r.read_address_row_i_two_cycles_ahead - 1;
```

```vhdl
v.read_address_even := r.read_address_row_i_two_cycles_ahead - 1;
```

```vhdl
v.read_address_row_i_two_cycles_ahead := r.read_address_row_i_two_cycles_ahead - 1;
```

```vhdl
v.flag_prev_row_i_at_odd_row := '1';
```

```vhdl
end if;
```

```vhdl
end if;
```

```vhdl
end if;
```

```vhdl
end if;
```
when ODD_i_START =>
  if (r.
    flag_finished_sending_data_to_BRAM_one_cycle_ago
    = '0') then
    for i in 0 to P_BANDS−1 loop
      v.row_j(i) := signed(data_out_brRAMs_M(i∗
        PIXEL_DATA_WIDTH+2 + PIXEL_DATA_WIDTH+2−1
        downto i∗PIXEL_DATA_WIDTH+2));
      v.row_i(i) := signed(data_out_brRAMs_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));
    end loop;
    v.flag_write_to_even_row := '1';
    v.flag_wr_row_i_at_odd_row := '1';
    v.flag_write_to_odd_row := '0';
    v.index_i := r.index_i−1;
    v.index_j := r.index_i−2;
    v.address_row_i := r.address_row_i−1;
    v.write_address_even := r.address_row_i−1;
    v.write_address_odd := r.address_row_i−1;
    v.write_enable_even := '1';
    v.write_enable_odd := '0';
    if (v.index_j > 1) then — the first two indexes
      are contained
      — within address 0
      — v.index_j := r.index_j−1;
      v.elimination_write_state := ODD_j_WRITE;
      elsif v.index_j = 0 and v.index_i = 1 then
        — In two clock cycles the data will be written
        to B_RAM.
        — and it is possible to change state to
        TOP_LAST_DIVISON.
        v.
        flag_finished_sending_data_to_BRAM_one_cycle_ago
        := '1';
      end if;

  end if;

if r.flag_last_read_backward_elimination = '0'
then
  if (r.read_address_odd >= 0 and r.
    index_j_two_cycles_ahead >= 1) then
if (r.read_address_odd >= 1 and r.index_j_two_cycles_ahead >= 2) then
    v.read_address_odd := r.read_address_odd;
    v.read_address_even := r.read_address_even;
    v.index_j_two_cycles_ahead := r.index_j_two_cycles_ahead - 1;
elsif r.index_j_two_cycles_ahead < 1 then
    if r.index_i_two_cycles_ahead >= 2 then
        v.index_i_two_cycles_ahead := r.index_i_two_cycles_ahead - 1;
        v.index_j_two_cycles_ahead := r.index_i_two_cycles_ahead - 2;
        if r.flag_prev_row_i_at_odd_row = '1' then
            v.read_address_even := r.read_address_row_i_two_cycles_ahead - 1;
            v.read_address_odd := r.read_address_row_i_two_cycles_ahead - 1;
            v.read_address_row_i_two_cycles_ahead := r.read_address_row_i_two_cycles_ahead;
            v.flag_prev_row_i_at_odd_row := '0';
        else
            v.read_address_even := r.read_address_row_i_two_cycles_ahead - 1;
            v.read_address_odd := r.read_address_row_i_two_cycles_ahead - 1;
            v.read_address_row_i_two_cycles_ahead := r.read_address_row_i_two_cycles_ahead - 1;
            v.flag_prev_row_i_at_odd_row := '1';
        end if;
    end if;
else
    v.read_address_even := 0;
    v.read_address_odd := 0;
end if;
else
    v.read_address_even := 0;
    v.read_address_odd := 0;
end if;
flag_finished_sending_data_to_BRAM_two_cycles_ago := '1';
end if;

if (r.
    flag_finished_sending_data_to_BRAM_two_cycles_ago
    = '1') then
  v.read_address_even := 0;
  v.read_address_odd := 0;
  v.
    flag_finished_sending_data_to_BRAM_three_cycles_ago
    := '1';
end if;

if r.
  flag_finished_sending_data_to_BRAM_three_cycles_ago
  = '1' then
  v.state_reg.state := STATE_LAST.Division;
  v.last_division_write_state := EVEN_i_WRITE;
  v.valid_data := '0';
  v.index_i_two_cycles_ahead := 2;
  v.flag_first_memory_request := '1'; -- used to indicate that the next cycle the first write will happen from
  v.read_address_even := 0;
  v.read_address_odd := 0;
  v.flag_finished_sending_data_to_BRAM_one_cycle_ago
  := '0';
  v.
    flag_finished_sending_data_to_BRAM_two_cycles_ago
    := '0';
  v.
    flag_finished_sending_data_to_BRAM_three_cycles_ago
    := '0';
end if;

when EVEN_i_START =>
for i in 0 to P_BANDS-1 loop
    v.row_i(i) := signed(data_out_brams_M(i*PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 downto i*PIXEL_DATA_WIDTH*2));
    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));
    v.inv_row_i(i) := signed(data_out_brams_M_inv(i*PIXEL_DATA_WIDTH*2 + PIXEL_DATA_WIDTH*2 downto i*PIXEL_DATA_WIDTH*2));
    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));
end loop;

v.flag_wr_row_i_at_odd_row := '0';
v.flag_write_to_even_row := '0';
v.flag_write_to_odd_row := '1';
v.index_i := r.index_i-1;
v.index_j := r.index_i-2;
v.address_row_i := r.address_row_i;
v.write_address_even := r.address_row_i-1;
v.write_address_odd := r.address_row_i-1;
v.write_enable_even := '0';
v.write_enable_odd := '1';
if (v.index_j >= 1) then
    v.elimination_write_state := EVEN_j_WRITE;
end if;

-- read new data.
if r.flag_last_read_backward_elimination = '0' then
    if (r.read_address_odd >= 1 and r.
        index_j_two_cycles_ahead >= 2) then
        need to read an odd row
    v.read_address_odd := r.read_address_odd -1;
    v.read_address_even := r.
        read_address_even-1;
    v.index_j_two_cycles_ahead := r.
        index_j_two_cycles_ahead-1;
elsif r.index_j_two_cycles_ahead < 1 then
    new i, update
    if r.index_i_two_cycles_ahead >= 2 then
        v.index_i_two_cycles_ahead := r.
            index_i_two_cycles_ahead-1;
        v.index_j_two_cycles_ahead := r.
            index_i_two_cycles_ahead-2;
    if r.flag_prev_row_i_at_odd_row = '1' then
        -- next row i will be located in an even
APPENDIX C. VHDL CODE

indexed row

v.read_address_even := r.read_address_row_i_two_cycles_ahead;

v.read_address_odd := r.read_address_row_i_two_cycles_ahead - 1;

v.read_address_row_i_two_cycles_ahead := r.read_address_row_i_two_cycles_ahead;

v.flag_prev_row_i_at_odd_row := '0';

else

next row_i will be located in an odd indexed row

v.read_address_odd := r.read_address_row_i_two_cycles_ahead - 1;

v.read_address_even := r.read_address_row_i_two_cycles_ahead - 1;

v.read_address_row_i_two_cycles_ahead := r.read_address_row_i_two_cycles_ahead - 1;

v.flag_prev_row_i_at_odd_row := '1';

end if;

end if;

if v.index_i_two_cycles_ahead = 1 then

Finished reading data for backward elimination

v.flag_last_read_backward_elimination := '1';

end if;

end if;

when others =>

end case;

when STATE_LAST_DIVISION =>

case r.last_division_write_state is

when EVEN_i_WRITE =>

if (r.flag_first_memory_request = '1') then

First write is to a even row

v.index_i := 0;

v.flag_first_memory_request := '0';

v.write_address_even := 0;

v.write_address_odd := 0;

v.write_address_even := 0;

v.write_address_odd := 0;

v.valid_data := '1';

else

v.index_i := r.index_i+1;

v.write_address_even := write_address_even + 1;

v.write_address_odd := write_address_odd+1;

end if;

v.flag_write_to_even_row := '1';

v.flag_write_to_odd_row := '0';
if \( r \text{. read_address\_even} < \frac{P\_BANDS}{2} - 1 \) then
    \( v \text{. read_address\_even} := r \text{. read_address\_even} + 1; \)
    \( v \text{. read_address\_odd} := r \text{. read_address\_even} + 1; \)
end if;

for i in 0 to P\_BANDS–1 loop
    -- data is located in the even part of the output from BRAM
    v.row\_i(i) := \text{signed}(data\_out\_brams\_M}(i*PIXEL\_DATA\_WIDTH*2 + \text{PIXEL\_DATA\_WIDTH*2} - 1 downto i*\text{PIXEL\_DATA\_WIDTH*2});
    v.inv_row\_i(i) := \text{signed}(data\_out\_brams\_M\_inv}(i*PIXEL\_DATA\_WIDTH*2 + \text{PIXEL\_DATA\_WIDTH*2} - 1 downto i*\text{PIXEL\_DATA\_WIDTH*2});
end loop;

if v.index\_i <= P\_BANDS–2 then
    v.last\_division\_write\_state := ODD\_i\_WRITE;
end if;

when ODD\_i\_WRITE =>
    v.index\_i := r.index\_i + 1;
    if v.index\_i >= P\_BANDS–1 then
        -- top_last\_division is finished written
        v.state\_reg\_state := STATE\_OUTPUT\_INVERSE\_MATRIX;
        v.read\_address\_odd := 0;
        v.read\_address\_even := 0;
        v.read\_enable := ’1’;
        v.counter\_output\_inverse\_matrix := 0;
    else
        -- row\_i is outputted from odd BRAMs (located at higher end of output)
        for i in 0 to P\_BANDS–1 loop
            v.row\_i(i) := \text{signed}(data\_out\_brams\_M}(i*PIXEL\_DATA\_WIDTH*2 + \text{PIXEL\_DATA\_WIDTH*2} + \text{EVEN\_ROW\_TOP\_INDEX} downto i*\text{PIXEL\_DATA\_WIDTH} + \text{EVEN\_ROW\_TOP\_INDEX} + 1);
            v.inv_row\_i(i) := \text{signed}(data\_out\_brams\_M\_inv}(i*PIXEL\_DATA\_WIDTH*2 + \text{PIXEL\_DATA\_WIDTH*2} + \text{EVEN\_ROW\_TOP\_INDEX} downto i*\text{PIXEL\_DATA\_WIDTH} + \text{EVEN\_ROW\_TOP\_INDEX} + 1);
        end loop;
        v.read\_address\_even := r.read\_address\_even;
        v.read\_address\_odd := r.read\_address\_odd;
        v.flag\_write\_to\_even\_row := ’0’;
        v.flag\_write\_to\_odd\_row := ’1’;
    end if;
    if v.index\_i <= P\_BANDS–3 then
        v.last\_division\_write\_state := EVEN\_i\_WRITE;
    end if;
when others =>
    v.read\_address\_even := 0;
APPENDIX C. VHDL CODE

```
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  when others =>
1190    v.read_enable := '0';
1191    v.write_enable_even := '0';
1192    v.write_enable_odd := '0';
1193    v.elimination_write_state := STATE_IDLE;
1194    v.state_reg.state := STATE_IDLE;
1195    v.last_division_write_state := STATE_IDLE;
1196    v.valid_data := '0';
1197  end case;
1198  if (reset_n = '0') then
1199    v.read_enable := '0';
1200    v.write_enable_even := '0';
1201    v.write_enable_odd := '0';
1202    v.elimination_write_state := STATE_IDLE;
1203    v.state_reg.state := STATE_IDLE;
1204    v.last_division_write_state := STATE_IDLE;
1205    v.valid_data := '0';
1206  end if;
1207  r_in <= v;
1208 end process;
1209
reps : process (clk, reset_n, clk_en)
1210 begin
1211 if rising_edge(clk) and clk_en = '1' then
1212 if (reset_n = '0') then
1213 else
1214 r <= r_in;
1215 end if;
1216 end if;
1217 end process;
```

C.7 Shiftregister

Listing C.7: Shiftregister

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.Common_types_and_functions.all;

-- A serial (n bit at a time) in parallel out shift register
-- Inputs four bands at a time, until a whole pixel is shifted in
entity shiftregister_four_pixels is
  port (din : in std_logic_vector (PIXEL_DATA_WIDTH * ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA − 1 downto 0);
       valid : in std_logic;
       clk : in std_logic;
       clk_en : in std_logic;
       reset_n : in std_logic;
   shift_counter : out std_logic_vector(log2(P_BANDS * PIXEL_DATA_WIDTH/(PIXEL_DATA_WIDTH * ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA)) downto 0);
       valid_out : out std_logic;
       dout : inout std_logic_vector(P_BANDS * PIXEL_DATA_WIDTH − 1 downto 0)) ;
end shiftregister_four_pixels;

architecture Behavioral of shiftregister_four_pixels is
  signal r, r_in : std_logic_vector (P_BANDS*PIXEL_DATA_WIDTH − 1 downto 0);
  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');
  signal r_in_valid_out : std_logic;
  variable v_shift_counter : integer := to_integer (unsigned(r_shift_counter));
begin
comb_proc : process(din, valid, dout, r_shift_counter)
  variable v_shift_counter : integer := to_integer (unsigned(r_shift_counter));
```
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);
variable v : std_logic_vector(P_BANDS*PIXEL_DATA_WIDTH-1 downto 0);
variable v_valid_out : std_logic := '0';
begin
  if (valid = '1') then
    v_shift_counter := to_integer(unsigned(r_shift_counter)) + 1;
    v := dout;
    v_temp_shift_data_in := v(P_BANDS*PIXEL_DATA_WIDTH-1 downto ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA*PIXEL_DATA_WIDTH downto 0);
    v(P_BANDS*PIXEL_DATA_WIDTH-1 downto P_BANDS*PIXEL_DATA_WIDTH-1 ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA*PIXEL_DATA_WIDTH downto 0) := v_temp_shift_data_in;
    v(P_BANDS*PIXEL_DATA_WIDTH-1 downto P_BANDS*PIXEL_DATA_WIDTH-1 ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA*PIXEL_DATA_WIDTH downto 0) := din;
    if v_shift_counter = P_BANDS/ELEMENTS_SHIFTED_IN_FROM_CUBE_DMA then
      v_valid_out := '1';
    else
      v_valid_out := '0';
    end if;
  else
    v_shift_counter := 0;
    v := (others => '0');
    v_valid_out := '0';
  end if;
  if (reset_n = '0') then
    v := (others => '0');
    v_shift_counter := 0;
    v_valid_out := '0';
  end if;
  r_shift_counter_in <= std_logic_vector(to_unsigned(
    v_shift_counter, r_shift_counter_in'length));
  shift_counter <= r_shift_counter;
  r_in_valid_out <= v_valid_out;
  r_in <= v;
  dout <= r;
end process;
C.8. FORWARD ELIMINATION

Listing C.8: Forward elimination

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.Common_types_and_functions.all;

-- This module controls the forward elimination stage. It issues reads and writes to BRAM
entity top_forward_elimination is
  port( clk : in std_logic;
        reset_n : in std_logic;
        clk_en : in std_logic;
        input_forward_elimination : in
          input_elimination_reg_type;
        output_forward_elimination : out
          output_forward_elimination_reg_type
          output_top_level : output_forward_elimination_reg_type
    );
end top_forward_elimination;

architecture Behavioral of top_forward_elimination is
begin
  -- Instance to swap the rows if needed.
  swap_rows_1 : entity work.swap_rows_module
    port map ( clk => clk,
```
APPENDIX C. VHDL CODE

```vhdl
reset_n  => reset_n ,
clk_en   => clk_en ,
input_swap_rows => input_swap_rows ,
output_swap_rows => output_swap_rows);

-- process to set output to inverse top level
set_outputs : process (r , output_swap_rows , output_top_level)
begin
  case r . forward_elimination_write_state is
    when STATE_IDLE =>
      output_forward_elimination <= output_top_level ;
    when CHECK_DIAGONAL_ELEMENT_IS_ZERO =>
      output_forward_elimination <= output_top_level ;
    when SWAP_ROWS =>
      output_forward_elimination <= output_swap_rows ;
    when EVEN_j_WRITE =>
      output_forward_elimination <= output_top_level ;
    when ODD_j_WRITE =>
      output_forward_elimination <= output_top_level ;
    when others =>
      output_forward_elimination <= output_top_level ;
  end case ;
end process ;

set_inputs_to_swap_rows : process (input_forward_elimination , r )
begin
  case r . forward_elimination_write_state is
    when STATE_IDLE =>
      input_swap_rows . forward_elimination_write_state <=
        STATE_IDLE ;
      input_swap_rows . row_i <= ((
        others => (others => '0'))) ;
      input_swap_rows . row_j <= ((
        others => (others => '0'))) ;
      input_swap_rows . index_i <= 0 ;
      input_swap_rows . index_j <= 0 ;
      input_swap_rows . address_row_i <= 0 ;
      input_swap_rows . address_row_j <= 0 ;
      input_swap_rows . flag_write_to_even_row <= '0' ;
      input_swap_rows . flag_write_to_odd_row <= '0' ;
      input_swap_rows . flag_prev_row_i_at_odd_row <= '0' ;
    when SWAP_ROWS =>
      input_swap_rows . forward_elimination_write_state <= r .
        forward_elimination_write_state ;
      if r . flag_start_swapping_rows = '1' then
        -- input from top level
        input_swap_rows . row_i <= r . row_i ;
        input_swap_rows . row_j <= r . row_j ;
        input_swap_rows . index_i <= r .
```
index_i;
input_swap_rows.index_j <= r.
index_j;
input_swap_rows.address_row_i <= r.
    address_row_i;
input_swap_rows.address_row_j <= r.
    address_row_j;
input_swap_rows.flag_write_to_even_row <= r.
    flag_write_to_even_row;
input_swap_rows.flag_write_to_odd_row <= r.
    flag_write_to_odd_row;
input_swap_rows.flag_prev_row_i_at_odd_row <= r.
    flag_prev_row_i_at_odd_row;
else
    -- receive row i and row j from BRAM directly
    -- Not been simulated and tested that this works.
    input_swap_rows.row_i <=
        input_forward_elimination.row_i;
    input_swap_rows.row_j <=
        input_forward_elimination.row_j;
    input_swap_rows.index_i <=
        input_forward_elimination.index_i;
    input_swap_rows.index_j <=
        input_forward_elimination.index_j;
    input_swap_rows.address_row_i <=
        input_forward_elimination.address_row_i;
    input_swap_rows.address_row_j <=
        input_forward_elimination.address_row_j;
    input_swap_rows.flag_write_to_even_row <=
        input_forward_elimination.flag_write_to_even_row;
    input_swap_rows.flag_write_to_odd_row <=
        input_forward_elimination.flag_write_to_odd_row;
    input_swap_rows.flag_prev_row_i_at_odd_row <=
        input_forward_elimination.
        flag_prev_row_i_at_odd_row;
end if;
when others =>
    input_swap_rows.forward_elimination_write_state <=
        STATE_IDLE;
    input_swap_rows.row_i <= ((
        others => (others => '0')));
    input_swap_rows.row_j <= ((
        others => (others => '0')));
    input_swap_rows.index_i <= 0;
    input_swap_rows.index_j <= 0;
    input_swap_rows.address_row_i <= 0;
    input_swap_rows.address_row_j <= 0;
    input_swap_rows.flag_write_to_even_row <= '0';
    input_swap_rows.flag_write_to_odd_row <= '0';
    input_swap_rows.flag_prev_row_i_at_odd_row <= '0';
comb_process : process (output_swap_rows, input_forward_elimination, r, reset_n)
begin
variable v : input_elimination_reg_type;
begin
v := r;
v.state_reg := input_forward_elimination.state_reg;
if (input_forward_elimination.state_reg.state = STATE_FORWARD_ELIMINATION and input_forward_elimination.valid_data = '1') then
case r.forward_elimination_write_state is
when STATE_IDLE =>
v.valid_data := '0';
-- input_elimination.flag_first_data_elimination is to be sent only
-- once, by the top level inverse
if input_forward_elimination.flag_first_data_elimination = '1' then
v.forward_elimination_write_state := CHECK_DIAGONAL_ELEMENT_IS_ZERO;
v.flag_first_data_elimination := '1';
end if;
when CHECK_DIAGONAL_ELEMENT_IS_ZERO =>
if r.flag_first_data_elimination = '1' then
-- First iteration of the forward-elimination
-- for the current processed pixel
v.index_i := 0;
v.index_j := 0;
-- Has already read the first j
v.index_j_two_cycles_ahead := 2;
v.read_address_row_i_two_cycles_ahead := 0;
v.flag_write_to_even_row := '0';
v.flag_write_to_odd_row := '1';
v.write_address_even := 0;
v.write_address_odd := 0;
v.valid_data := '0';
v.flag_first_data_elimination := '0';
-- First iteration row_i is located at even index=0
v.row_i := input_forward_elimination.row_even;
v.row_j := input_forward_elimination.row_odd;
v.inv_row_i :=
end case;
end process;
```cpp
C.8. FORWARD ELIMINATION

201

input_forward_elimination.inv_row_even;

v.inv_row_j :=
input_forward_elimination.inv_row_odd;

v.address_row_i := 0;

v.flag_prev_row_i_at_odd_row := '0';

v.wait_counter := 0;

v.flag_waiting_for_bram_update := '0';

elsif r.index_i >= P_BANDS−2 and r.index_j >= P_BANDS−3 then

--- Forward elimination is finished.

v.valid_data := '0';

v.flag_write_to_odd_row := '0';

v.flag_write_to_even_row := '0';

v.read_address_even := P_BANDS/2−1;

v.read_address_even := P_BANDS/2−1;

else

v.valid_data := '0';

v.index_i := r.index_i + 1;

--- 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

v.index_j := r.index_i + 1;

--- flag_prev_row_i_at_odd_row set by EVEN_j_WRITE

--- or if previous index_j = P_BANDS−1 and index_i=
P_BANDS−3, then

--- it was set by ODD_j_WRITE

if r.flag_prev_row_i_at_odd_row = '1' then

v.flag_write_to_odd_row := '0';

v.flag_write_to_even_row := '1';

else

v.flag_write_to_odd_row := '1';

v.flag_write_to_even_row := '0';

end if;

v.wait_counter := 0;

v.flag_waiting_for_bram_update := '0';

if v.flag_write_to_even_row = '1' then

--- index_i at odd row i

--- address row i?

v.address_row_i := r.address_row_i;

v.address_row_j := r.address_row_i+1;

--- write address is changed in EVEN_j_WRITE before

writing

v.write_address_even := r.address_row_i;

v.write_address_odd := r.address.row_i;

v.row_i := input_forward_elimination.row_odd;

v.row_j := input_forward_elimination.row_even;

v.inv_row_i := input_forward_elimination.inv_row_odd;

```
v.inv_row_j := input_forward_elimination.inv_row_even;

else
    -- index i at even row
    v.address_row_i := r.address_row_i+1;
    v.write_address_odd := r.write_address_odd +1;
    v.address_row_j := r.address_row_i+1;
    v.write_address_even := r.write_address_even;
    v.row_i := input_forward_elimination.row_even;
    v.row_j := input_forward_elimination.row_odd;
    v.inv_row_i := input_forward_elimination.inv_row_even;
    v.inv_row_j := input_forward_elimination.inv_row_odd;
end if;
end if;

if v.row_i(v.index_i) = 0 then
    v.forward_elimination_write_state := SWAP_ROWS;
    v.flag_start_swapping_rows := '1';
    -- insecure about the reading process here...
    v.read_address_even := r.read_address_even +1;
    v.read_address_odd := r.read_address_odd +1;
else
    if v.flag_write_to_even_row = '1' then -- and data is ready
        v.forward_elimination_write_state := EVEN_j_WRITE;
        v.read_address_even := r.read_address_even;
        v.read_address_odd := r.read_address_odd+1;
    else
        v.forward_elimination_write_state := ODD_j_WRITE;
        v.read_address_even := r.read_address_even+1;
        v.read_address_odd := r.read_address_odd+1;
    end if;
end if;

when SWAP_ROWS =>
    -- wait until received new swapped rows from swapped row module
    v.flag_start_swapping_rows := '0';
    if output_swap_rows.valid_data = '1' then
        -- A swap of rows have happened. The forward elimination can continue
        if output_swap_rows.flag_prev_row_i_at_odd_row = '1'
        end if;
    end if;
then
  v.forward_elimination_write_state := EVEN_j_WRITE;
  −− read data. Need to read an odd row
  v.read_address_odd :=
    output_swap_rows.read_address_odd;
  v.read_address_even :=
    output_swap_rows.read_address_even;
else
  v.forward_elimination_write_state := ODD_j_WRITE;
  −− read data. Need to read an even row
  v.read_address_odd :=
    output_swap_rows.read_address_odd +1;
  v.read_address_even :=
    output_swap_rows.read_address_even +1;
end if;
end if;
when EVEN_j_WRITE =>
  −− Need to check if i two cycles forward is at new place..
  if r.flag_waiting_for_bram_update = '0' then
    v.valid_data := '1';
    v.flag_write_to_even_row := '1';
    v.flag_write_to_odd_row := '0';
    v.valid_data := '1';
    v.index_j := r.index_j+1;
    v.row_j :=
      input_forward_elimination.row_even;
    v.inv_row_j :=
      input_forward_elimination.inv_row_even;
    if r.index_j <= P_BANDS−2 then
      v.write_address_even := r.write_address_even+1;
      v.write_address_odd := r.write_address_odd +1;
    end if;
    if v.index_j >= P_BANDS−2 and v.index_j − v.
      index_j_two_cycles_ahead < B_RAM_WAIT_CLK_CYCLES
      and r.wait_counter < B_RAM_WAIT_CLK_CYCLES−(v.
      index_j − v.index_j_two_cycles_ahead) then
      −− Need to wait for the row to update before reading
      it
      v.wait_counter := r.wait_counter+1;
      v.flag_waiting_for_bram_update := '1';
    else
      v.valid_data := '0';
      v.flag_write_to_even_row := '0';
      v.flag_write_to_odd_row := '0';
      v.valid_data := '1';
      v.index_j := r.index_j+1;
      v.row_j :=
        input_forward_elimination.row_even;
      v.inv_row_j :=
        input_forward_elimination.inv_row_even;
      v.write_address_even := r.write_address_even+1;
      v.write_address_odd := r.write_address_odd +1;
    end if;
  end if;
end when;
when ODD_j_WRITE =>
  −− read data. Need to read an odd row
  v.read_address_odd :=
    output_swap_rows.read_address_odd +1;
  v.read_address_even :=
    output_swap_rows.read_address_even +1;
end if;
end case;
v.wait_counter := 0;
v.flag_waiting_for_bram_update := '0';
if v.index_j <= P_BANDS-2 then
  v.forward_elimination_write_state := ODD_j_WRITE;
end if;
— read new data. Data need to be read two clock cycles in advance
if(r.read_address_even <= P_BANDS/2−1 and r.
  index_j_two_cycles_ahead <= P_BANDS−3) then
  — need to read an even row
  v.read_address_even := r.read_address_even +1;
  v.read_address_odd := r.read_address_odd +1;
  v.index_j_two_cycles_ahead := r.
    index_j_two_cycles_ahead +1;
elsif v.index_j >= P_BANDS−3 then
  — new i, update
    if r.index_i_two_cycles_ahead <= P_BANDS−3 then
      — v.index_j_two_cycles_ahead := r.
        index_i_two_cycles_ahead+1;
      — v.index_j_two_cycles_ahead := r.
        index_j_two_cycles_ahead +2;
      if r.flag_prev_row_i_at_odd_row = '1' then
        — next row i will be located in an even indexed row
        v.read_address_even := r.
          address_row_i+1;
        v.read_address_odd := r.
          address_row_i+1;
        v.read_address_row_i_two_cycles_ahead := r.
          address_row_i_two_cycles_ahead+1;
        v.flag_prev_row_i_at_odd_row := '0';
      else
        — next row i will be located in an odd indexed row
        — Row even will be located at an address one increment ahead
        v.read_address_odd := r.
          read_address_row_i_two_cycles_ahead;
        v.read_address_even := r.
          read_address_row_i_two_cycles_ahead+1;
        v.read_address_row_i_two_cycles_ahead := r.
          read_address_row_i_two_cycles_ahead+1;
        v.flag_prev_row_i_at_odd_row := '1';
      end if;
    end if;
  end if;
end if;
when ODD_j_WRITE =>
  -- Need to check if i two cycles forward is at new place..
  if r.flag_waiting_for_bram_update = '0' then
    v.valid_data := '1';
    v.flag_write_to_even_row := '0';
    v.flag_write_to_odd_row := '1';
    v.valid_data := '1';
    v.row_j := input_forward_elimination.row_odd;
    v.inv_row_j := input_forward_elimination.inv_row_odd;
    v.index_j := r.index_j+1;
    v.address_row_i := r.address_row_i;
    v.write_address_even := r.write_address_even;
    v.write_address_odd := r.write_address_odd;
  end if;
  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
    -- Need to wait for the row to update before reading it
    v.wait_counter := r.wait_counter+1;
    v.flag_waiting_for_bram_update := '1';
  else
    v.wait_counter := 0;
    v.flag_waiting_for_bram_update := '0';
    -- Set next state
    if v.index_j <= P_BANDS-3 then
      v.forward_elimination_write_state := EVEN_j_WRITE;
    else
      -- New iteration of the outermost loop
      if v.index_j = P_BANDS-1 and v.index_i = P_BANDS-3
        then
        -- The last row_i of the forward elimination is located at an
        -- even indexed row.
        v.flag_prev_row_i_at_odd_row := '0';
      end if;
      v.forward_elimination_write_state := CHECK_DIAGONAL_ELEMENT_IS_ZERO;
    end if;
    -- read new data. Data need to be read two clock cycles in advance
    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
APPENDIX C. VHDL CODE

-- need to read an odd row
v.read_address_even := r.read_address_even;
v.read_address_odd := r.read_address_odd;
v.index_j_two_cycles_ahead := r.
    index_j_two_cycles_ahead +1;
elif v.index_j >= P_BANDS-1 then
    -- In the previous clock cycle a new index i was read
    v.read_address_even := r.read_address_even;
v.read_address_odd := r.read_address_odd;
v.index_j_two_cycles_ahead := r.
    index_j_two_cycles_ahead +1;
    elsif r.flag_prev_row_i_at_odd_row = '1' then
    -- if r.flag_prev_row_i_at_odd_row = '1' then
    -- v.read_address_even := r.
        read_address_even;
    -- v.read_address_odd := r.
        read_address_even;
    -- v.index_j_two_cycles_ahead := r.
        index_j_two_cycles_ahead +1;
    -- else
    -- v.read_address_even := r.
        read_address_even;
    -- v.read_address_odd := r.
        read_address_odd;
    -- v.index_j_two_cycles_ahead := r.
        index_j_two_cycles_ahead +1;
    -- end if;
end if;
when others =>
v.forward_elimination_write_state := STATE_IDLE;
v.flag_write_to_odd_row := '0';
v.flag_write_to_even_row := '0';
end case;
end if;
if (reset_n = '0') then
    v.index_i := 0;
v.index_j := 1;
v.valid_data := '0';
v.address_row_i := 0;
v.flag_write_to_even_row := '0';
v.flag_write_to_odd_row := '0';
v.forward_elimination_write_state := STATE_IDLE;
end if;
r_in <= v;
data
output_top_level.row_j <= r.row_j;
output_top_level.row_i <= r.row_i;
C.9. LAST DIVISION

Listing C.9: Last division

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
```

```vhdl
end Behavioral;
```
use work.Common_types_and_functions.all;

entity top_last_division is
    port ( clk : in std_logic;
            reset_n : in std_logic;
            clk_en : in std_logic;
            input_last_division : in input_last_division_reg_type;
            output_last_division : out output_last_division_reg_type)
    ;
end top_last_division;

architecture Behavioral of top_last_division is

    signal r, r_in : input_last_division_reg_type;
    signal divisor_is_negative : std_logic;
        number of shifts required to approximate the division
    signal divisor : std_logic_vector(PIXEL_DATA_WIDTH*2−1 downto 0);
        If the divisor is negative, we need to take two's complement of the divisor
    signal divisor_valid : std_logic
        := '0';
    signal remainder_valid : std_logic
        := '0';

    type remainders_array is array(0 to PIXEL_DATA_WIDTH*2−2) of std_logic_vector(PIXEL_DATA_WIDTH*2−1 downto 0);
    signal remainders : remainders_array;

    constant ONE : signed(PIXEL_DATA_WIDTH*2−1 downto 0) := (0 => '1', others => '0');

    signal msb_index : integer range 0 to 31;  — msb of the divisor(unsigned)
        msb of the divisor (unsigned)
    signal msb_valid : std_logic
        := '0';

    signal divisor_lut : unsighed(DIV_PRECISION−1 downto 0);

    signal divisor_inv : unsigned(DIV_PRECISION−1 downto 0);

begin

    division_lut_1 : entity work.division_lut
        port map ( 
            y => divisor_lut,  
            y_inv => divisor_inv);

    input_to_divisor_lut : process(msb_valid, msb_index)
        begin
            if msb_valid = '1' and msb_index<=DIV_PRECISION then
C.9. LAST DIVISION

```vhDL
divisor_lut <= to_unsigned(to_integer(unsigned(divisor)), DIV_PRECISION);
else
  divisor_lut <= to_unsigned(0, DIV_PRECISION);
end if;
end process;

check_if_divisor_is_negative : process(input_last_division.
  state_reg.state, input_last_division.row_i, input_last_division.valid_data, reset_n)
begin
  if reset_n = '0' or not(input_last_division.state_reg.state = STATE_LAST_DIVISION) then
    divisor_valid <= '0';
    divisor_is_negative <= '0';
    divisor <= std_logic_vector(to_signed(1, PIXEL_DATA_WIDTH*2));
  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
    -- row[i][i] is negative
    -- using the absolute value
    divisor_is_negative <= '1';
    divisor <= std_logic_vector(abs(signed( input_last_division.row_i(input_last_division.index_i))));
    divisor_valid <= '1';
  elsif input_last_division.valid_data = '1' then
    divisor_is_negative <= '0';
    divisor <= std_logic_vector(input_last_division.row_i(input_last_division.index_i));
    divisor_valid <= '1';
  else
    divisor_valid <= '0';
    divisor_is_negative <= '0';
    divisor <= std_logic_vector(to_signed(1, PIXEL_DATA_WIDTH*2));
  end if;
end process;

-- generate PIXEL_DATA_WIDTH*2−1 number of shifters that shifts
-- A[i][i] n places in order to see how many shifts yield the best
-- approximation to the division. Don't need to shift the
-- 31 bit as this is the sign bit.
generate_shifters : for i in 1 to PIXEL_DATA_WIDTH*2−1
  generate
    signal remainder_after_approximation_i :
```
begin
process(divisor, divisor_valid, reset_n, input_last_division, state_reg)
begin
if reset_n = '0' or not(input_last_division.state_reg.state = STATE_LAST_DIVISION) then
    remainder_after_approximation_i.remainder <=
    std_logic_vector(shift_right(signed(divisor), i));
    remainder_after_approximation_i.number_of_shifts <= i;
    remainder_after_approximation_i.remainder_valid <= '0';
elsif divisor_valid = '1' then
    remainder_after_approximation_i.remainder <=
    std_logic_vector(shift_right(signed(divisor), i));
    remainder_after_approximation_i.number_of_shifts <= i;
    remainder_after_approximation_i.remainder_valid <= '1';
else
    remainder_after_approximation_i.remainder <=
    std_logic_vector(shift_right(signed(divisor), i));
    remainder_after_approximation_i.number_of_shifts <= i;
    remainder_after_approximation_i.remainder_valid <= '0';
end if;
end process;
remainders(i-1) <= remainder_after_approximation_i.remainder;
remainder_valid <= remainder_after_approximation_i.remainder_valid;
end generate;

find_msb : process(divisor_valid, input_last_division, reset_n, divisor)
begin
if divisor_valid = '1' and reset_n = '1' then
    --For PIXEL_DATA_WIDTH = 16.
    if divisor(30) = '1' then
        msb_index <= 30;
        msb_valid <= '1';
    elsif divisor(29) = '1' then
        msb_index <= 29;
        msb_valid <= '1';
    elsif divisor(28) = '1' then
        msb_index <= 28;
        msb_valid <= '1';
    elsif divisor(27) = '1' then
        msb_index <= 27;
        msb_valid <= '1';
    elsif divisor(26) = '1' then
        msb_index <= 26;
        msb_valid <= '1';
    end if;
end process;
elsif divisor (25) = '1' then
  msb_index <= 25;
  msb_valid <= '1';
elsif divisor (24) = '1' then
  msb_index <= 24;
  msb_valid <= '1';
elsif divisor (23) = '1' then
  msb_index <= 23;
  msb_valid <= '1';
elsif divisor (22) = '1' then
  msb_index <= 22;
  msb_valid <= '1';
elsif divisor (21) = '1' then
  msb_index <= 21;
  msb_valid <= '1';
elsif divisor (20) = '1' then
  msb_index <= 20;
  msb_valid <= '1';
elsif divisor (19) = '1' then
  msb_index <= 19;
  msb_valid <= '1';
elsif divisor (18) = '1' then
  msb_index <= 18;
  msb_valid <= '1';
elsif divisor (17) = '1' then
  msb_index <= 17;
  msb_valid <= '1';
elsif divisor (16) = '1' then
  msb_index <= 16;
  msb_valid <= '1';
elsif divisor (15) = '1' then
  msb_index <= 15;
  msb_valid <= '1';
elsif divisor (14) = '1' then
  msb_index <= 14;
  msb_valid <= '1';
elsif divisor (13) = '1' then
  msb_index <= 13;
  msb_valid <= '1';
elsif divisor (12) = '1' then
  msb_index <= 12;
  msb_valid <= '1';
elsif divisor (11) = '1' then
  msb_index <= 11;
  msb_valid <= '1';
elsif divisor (10) = '1' then
  msb_index <= 10;
  msb_valid <= '1';
elsif divisor (9) = '1' then
  msb_index <= 9;
APPENDIX C. VHDL CODE

msb_valid <= '1';
elif divisor(8) = '1' then
  msb_index <= 8;
  msb_valid <= '1';
elif divisor(7) = '1' then
  msb_index <= 7;
  msb_valid <= '1';
elif divisor(6) = '1' then
  msb_index <= 6;
  msb_valid <= '1';
elif divisor(5) = '1' then
  msb_index <= 5;
  msb_valid <= '1';
elif divisor(4) = '1' then
  msb_index <= 4;
  msb_valid <= '1';
elif divisor(3) = '1' then
  msb_index <= 3;
  msb_valid <= '1';
elif divisor(2) = '1' then
  msb_index <= 2;
  msb_valid <= '1';
elif divisor(1) = '1' then
  msb_index <= 1;
  msb_valid <= '1';
elif divisor(0) = '1' then
  msb_index <= 0;
  msb_valid <= '1';
else
  msb_valid <= '0';
  msb_index <= 0;
end if;
else
  msb_index <= 0;
  msb_valid <= '0';
end if;

comb_process : process(input_last_division, r, reset_n,
  divisor_is_negative, divisor, remainder_valid, remainders,
  msb_valid, divisor, divisor_inv, msb_index)
  variable v : input_last_division_reg_type;
  variable divisor_inv_from_lut : integer range 0 to 2**DIV_PRECISION := 0;
begin
  v := r;
  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
  v := input_last_division;
  v.best_approx := INITIAL_BEST_APPROX;
  v.msb_index := msb_index;

if v.msb_index <= DIV_PRECISION then
divisor_inv_from_lut := to_integer(divisor_inv);
else
  -- Using shifting approach
  divisor_inv_from_lut := to_integer(divisor_inv);

  -- The best approximation may be either the msb–shifted division, or the
  -- msb+1 shifted division.
  v.best_approx.remainder := remainders(v.msb_index);
  v.best_approx.number_of_shifts := v.msb_index;

  -- The best approximation to the divisor may be larger
  -- than the divisor.
  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
    -- This is a better approximation
    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));
    number_of_shifts+1;
  end if;
end if;

-- Doing division
if divisor_is_negative = '1' then
  for i in 0 to P_BANDS–1 loop
    if v.msb_index <= DIV_PRECISION then
      v.inv_row_i(i) := shift_right(input_last_division.
        inv_row_i(i)*divisor_inv_from_lut, DIV_PRECISION)
      ;
      -- v.inv_row_i(i) := shift_right(input_last_division.
      inv_row_i(i), v.best_approx.number_of_shifts);
      -- Negating the number with two’s complement
APPENDIX C. VHDL CODE

```vhdl
v.inv_row_i(i) := not(v.inv_row_i(i)) + ONE;
else
  v.inv_row_i(i) := shift_right(input_last_division.inv_row_i(i), v.best_approx.number_of_shifts);
end if;
v.inv_row_i(i) := not(v.inv_row_i(i)) + ONE;
end loop;
else
  for i in 0 to P_BANDS-1 loop
    if v.msb_index <= DIV_PRECISION then
      v.inv_row_i(i) := shift_right(input_last_division.inv_row_i(i)*divisor_inv_from_lut, DIV_PRECISION);
    else
      v.inv_row_i(i) := shift_right(input_last_division.inv_row_i(i), v.best_approx.number_of_shifts);
    end if;
  end loop;
end if;
end if;
if (reset_n = '0' or input_last_division.state_reg.state /= STATE_LAST_DIVISION) then
  v.valid_data := '0';
  v.best_approx := INITIAL_BEST_APPROX;
  v.msb_index := 31;
end if;
r_in <= v;
end process;
output_last_division.new_inv_row_i <= r.inv_row_i;
output_last_division.valid_data <= r.valid_data;
output_last_division.index_i <= r.index_i;
output_last_division.write_address_even <= r.write_address_even;
output_last_division.write_address_odd <= r.write_address_odd;
output_last_division.flag_write_to_even_row <= r.flag_write_to_even_row;
output_last_division.state_reg <= r.state_reg;
sequential_process : process (clk)
begin
  if rising_edge(clk) then
    if clk_en = '1' then
      r <= r_in;
    end if;
  end if;
end process;
```
end Behavioral;