NTNU
Norwegian University of
Science and Technology

[SmallSat] Onboard Data Processing Pipeline and Software/Hardware Implementation of Extended Multiplicative Signal Correction

Espen Moen

Master of Science in Electronics

Submission date:	June 2018
Supervisor:	Kjetil Svarstad, IES
Co-supervisor:	Milica Orlandic, IES

Norwegian University of Science and Technology
Department of Electronic Systems

1 Abstract

This thesis concerns the data processing pipeline from an *CMV2000v3* Image Sensor to memory in a Hyper-spectral imaging Application. A *Low Voltage Differential Signal* (LVDS) Receiver was implemented, consisting of a *Deserializer Module* and a *Pixel Order Alignment Module* to receive and organize the pixel data from the Image Sensor to the rest of the pipeline. Further a *Binning Module* was implemented to make simple binning operations on the pixel data to reduce minor observation errors as well as reducing the data size. Additionally the thesis presents an algorithm called *Extended Multiplicative Signal Correction* [8], and a proposal to how this can be implemented in a Software/Hardware Co-design executing on the Zynq-7000 platform. A pure software implementation was made to map which parts of the algorithm would benefit the most from being executed in hardware based on time consumption. Then a SW/HW co-design was implemented with the chosen parts accelerated in hardware. A speedup of 4.36 was achieved by the combined SW/HW implementation to the pure software version.

Denne masteroppgaven omhandler en pipeline for dataflyten fra en *CMV2000v3* bildesensor til bildedataen lagres i minnet i en hyperspektral bilde applikasjon. Dette innebærer en implementasjon av en LVDS mottaker bestående av to moduler, en *Deserializer* og en *Pixel Order Alignment Module*. Førstnevnte er en modul for å motta og parallellisere data over *LVDS* og sistnevnte er en modul for organisere denne dataen i riktig rekkefølge før den sendes videre. Videre ble en *Binning Module* implementert for å redusere små unøyaktigheter og datastørrelse. I tillegg til dette presenterer denne masteroppgaven en algoritme som heter *Extended Multiplicative Signal Correction* [8], og et forslag til hvordan denne kan implementeres i et *Software/Hardware Co-design* som skal kjøres på en Zynq-7000 platform. En ren *software* versjon ble først implementert for å strukturer hvilke deler av algorithmen som ville tjene mest på å bli akselerert i maskinvare for å senke kjøretiden. Basert på dette ble en *Software/Hardware* implementasjon designet som utførte algoritmen opptil 4.36 ganger raskere enn *software* versjonen.

2 Preface

I would like to thank my supervisor Milica Orlandic for her ability to motivate, as well as giving good advices. Additionally, i would like to thank my co-students in this project, Johan, Lars, Aysel and Martin for good cooperation through this year.

Contents

List of Figures

List of Tables

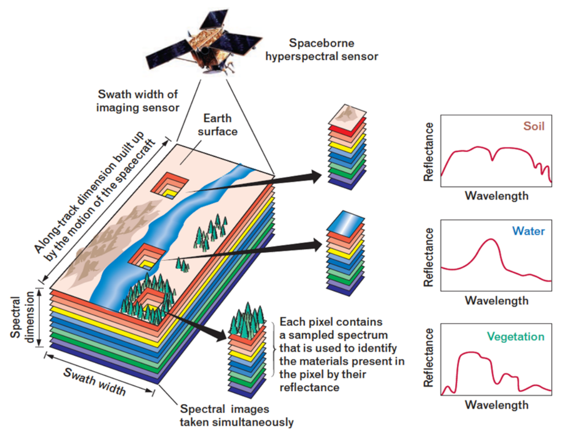
3 Introduction

3.1 NTNU SMALLSAT

The design problem for this thesis was issued by SmallSat - MASSIVE, a project collaboration between Department of Technical Cybernetics(ITK) and Department of Electronic Systems at the Norwegian University of Science and Technology. This project concerns development of a small satellite equipped with a hyperspectral camera. The information captured by this camera are planned to be used for supporting oceanographic applications. This could for example mean that the satellite captures images around a salmon farm, analyses these images to find indicators of water pollution, algae bloom and other metrics that may inflict the well being and quality of the salmons.

3.2 Hyperspectral Imaging

Hyperspectral imaging are a technique where you capture, for each pixel, hundreds or thousands of components spread across the electromagnetic spectrum. Different materials gives unique fingerprints in different spectrums which can be used to analyse and identify the materials captured in the image. This technique where you capture the frequency spectra using a camera is called spectrography [16]. Figure 1 illustrates how these images are captured. As the satellite moves over the scene it starts capturing rows of pixels where each pixel contains multiple components across the electromagnetic spectrum, referenced as *Spectral dimension* in the figure (figure 1). This will form a cube consisting of the number of pixels in one row, times the number of spectral components in one pixel, times the number of rows in the satellite motion dimension. This is illustrated in figure 2. As one probably would understand, this results in large amount of data which will be discussed further in this section.



Spaceborne hyperspectral sensor
Swath width of imaging sensor
Earth surface
Along-track dimension built up by the motion of the spacecraft
Spectral dimension
Swath width
Spectral images taken simultaneously
Each pixel contains a sampled spectrum that is used to identify the materials present in the pixel by their reflectance
Soil
Water
Vegetation
Reflectance
Wavelength

Figure 1: Illustration of Hyperspectral Image capture. [13]

As mentioned above the satellite that is constructed in the SmallSat project contains a hyperspectral camera able to capture hyperspectral images. In regular photography meant for the human eye, each pixel consists of pixel components in the visible light spectrum, usually long wavelengths (red), medium wavelengths (green) and short wavelengths (blue). This means that if you capture a picture with all channels (RGB), with a resolution of 1920x1080 you will have 2,073,600 pixels, each with 3 components, giving a total of 6,220,800 components. Each component is represented with a given number of bits called bit depth, giving the resolution of how many possible values for each component. For example, using a bit depth of 12 bits gives $2^{12} - 1 = 4095$ different representations, giving the example above a size of 6,220,800 components x 12 bit, or around 9.3 MB.

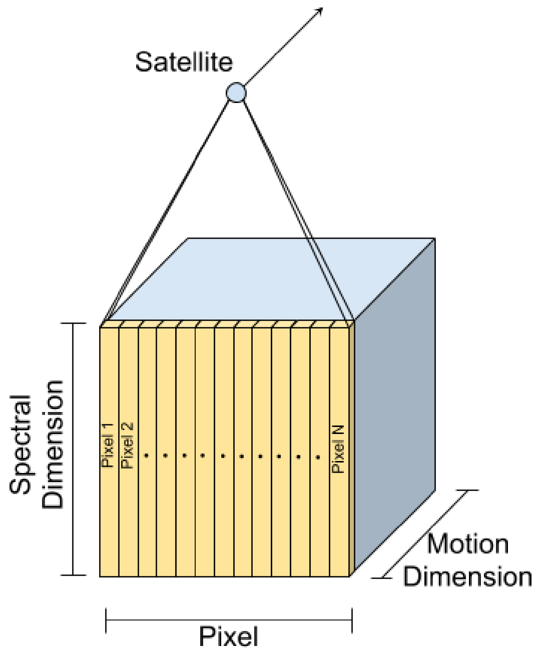


Figure 2: Cube representation of hyperspectral image data.

The example above demonstrates how large a digital image from regular photography becomes based on different choices or settings that are used. Even using only 3 components for each pixel, one get large amount of data if you increase the resolution and bit depths. For a Hyperspectral image this impacts the size even more due to hundreds or thousands of components per pixel. Adapting the RGB example above for a hyperspectral image containing 1000 components we get. 1920x1080x1000 = 2,073,600,000 components across all spectras, further giving 2,073,600,000 x 12 bit = 24,883,200,000 bits, or around

3.1 GB. For a modern computer where one usually have several TB of storage and powerful CPUs and GPUs available, this amount of data would be handled with minimal effort. However, for a small satellite where the data is to be sent back to earth through radio links, a energy expensive operation, it should be as minimal as possible to save both time and energy. Therefore a lot of data processing will have to be executed on-board the satellite. This thesis will look at this on-board data processing pipeline from the capture of image data from the image sensor and storing this data directly to memory or via compression algorithms or other data processing algorithms. In addition, this thesis will present a data processing algorithm called *Extended Multiplicative Signal Correction*, and how this algorithm may be implemented in hardware.

3.3 Main contributions

The main contributions of this thesis are achieved in communication between different parts of the hyperspectral payload, in particular building communication modules establishing data streaming between image sensor and memory. This consists of a a hardware implementation of a LVDS receiver consisting of a *Deserializer* and a *Pixel Order Alignment Module*, the former a serial-to-parallel converter for the *LVDS* data from the image sensor and the latter a module to organise the order of this data before it is processed or stored in memory. Further a *Binning Module* that merges either 4, 8 or 16 data samples into one sample with an average value of the merged samples. Additionally the thesis presents an algorithm, *Extended Multiplicative Signal Correction* and proposals to how this may be implemented to execute on a Zynq-7000 platform. This includes a pure software implementation, and a SW/HW co-design implementation. The pure software implementation was analysed to find what parts would benefit the most being accelerated in hardware, which resulted in the SW/HW co-design. The thesis presents the process of transforming a software into a software/hardware co-design which may be perceived as a complicated process. Further the SW/HW co-design was analysed and compared to the software version both in terms of speed and precision. Both implementations was analysed and compared to a MATLAB script which was considered the solution of a correct EMSC operation. Additionally a proposal to increase parallelism of the SW/HW co-design was implemented which in theory would increase speedup by four times. Tutorials of how to use both the software and software/hardware co-design was created and can be found in the Appendix together with all design files and testbenches used in the project. To be able to achieve these implementations knowledge had to be built around the design tools, Vivado and Xilinx SDK together with the *AXI-protocol*, Cube DMA and the Image Sensor which is used in these implementations.

4 Data Processing Pipeline

Figure 3 shows the main parts of the data processing pipeline mentioned in the previous section. The goal is to implement a Low Voltage Differential Signal (LVDS) receiver capable of capturing the data transmitted from the image sensor. On the output of the LVDS reciever there will be a stream of pixel components that either can be stored directly to a SD-card or sent further through a pipeline consisting of *Binning*, some *On-the-fly Application* and then stored on the on-chip memory by a Direct-Memory Access (DMA). *Binning* is a pre-processing technique and will be introduced later in this section (section 4.8). *On-the-fly Applications* simply means different algorithms that is necessary for the application. An example of this is Extended Multiplicative Signal Correction (EMSC) which will be presented further on in this paper (section 5).

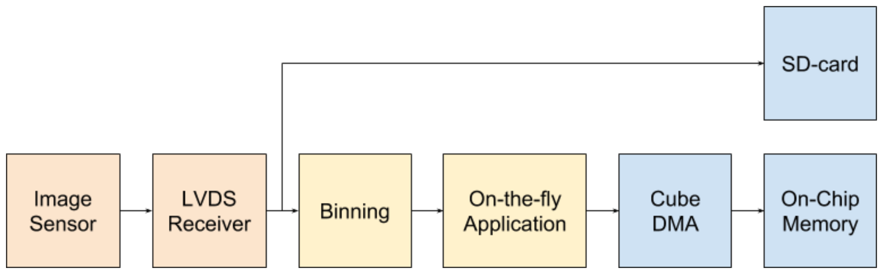


Figure 3: Data processing pipeline

4.1 Prototype

The prototype produced for this project contains 3 major parts. A carrier board, a PicoZed and an image sensor with associated optics. Shown in figure 4 is a rendering from a 3D model of the prototype. The green board on the left side is the PicoZed, the microscope looking part on the right side is the optics mounted on top of image sensor and the black bottom layer board which is connected to both the optics and the PicoZed is the carrier board.

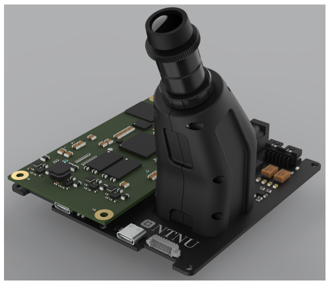


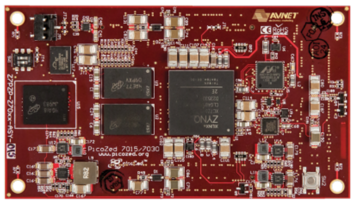
Figure 4: 3D Render of the prototype. [15]

4.2 Carrier Board

The carrier boards main task, is to connect the image sensor to the correct pins of the PicoZed, distribute the main power input into different voltage levels required by the different parts of the system, as well as providing a JTAG, USB-UART and SD card interface. The carrier board is powered through a 2.1mm/5.5mm barrel jack connector and this input is rated at 5-14.5V at 6A, meaning that any AC/DC adapter which fulfils these requirements may be used. The board contains 5 regulators, 1x 5V SMPS(switching mode power supply), 2x 1500mA LDO(Low-dropout) regulator and 2x 500mA LDO regulator providing the required voltages.

4.3 PicoZed

The PicoZed from AVNET is a so called SOM (System-On-Module) based around the *Xilinx Zynq-7000 All Programmable (AP) SoC*. Figures 5 a-b show the PicoZed and a block diagram of the architecture within. As can be seen from the Processing System (PS) of the block diagram there are multiple versions of the PicoZed containing different versions of the Xilinx Zynq. The PicoZed used in this prototype is equipped with the Xilinx Zynq-7030 containing a Dual-core Arm Cortex-A9, and a Programmable logic (PL) with 125 K programmable logic cells and 78,600 LUTs for briefly mentioning some of the specs.


(a) PicoZed. [2]

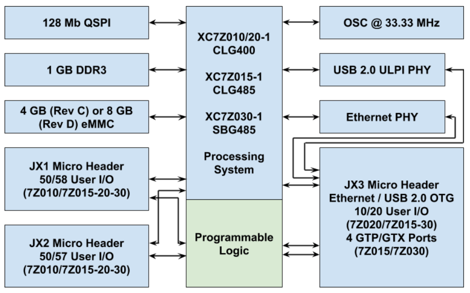

128 Mb QSPI
1 GB DDR3
4 GB (Rev C) or 8 GB (Rev D) eMMC
JX1 Micro Header 50/58 User I/O (7Z010/7Z015-20-30)
JX2 Micro Header 50/57 User I/O (7Z010/7Z015-20-30)
XC7Z010/20-1 CLG400
XC7Z015-1 CLG485
XC7Z030-1 SBG485
Processing System
Programmable Logic
OSC @ 33.33 MHz
USB 2.0 ULPI PHY
Ethernet PHY
JX3 Micro Header Ethernet / USB 2.0 OTG 10/20 User I/O (7Z020/7Z015-30) 4 GTP/GTX Ports (7Z015/7Z030)
(b) PicoZed Block Diagram. [2]

Figure 5: PicoZed Overview

4.4 Image Sensor

The image sensor used is the CMV2000v3 from CMOSIS. A CMOS image sensor with a resolution of 2048x1088 , global shutter and a maximum frame rate of 340fps at a bit width of 10 bits. The image sensor is aimed for applications such as 3D-imaging, Machine Vision, Bar and 2D code, Motion capture and more, and should also be applicable for Hyperspectral Imaging given the necessary camera lens. The sensor is soldered directly on the carrier board and will be connected to the PicoZed through a LVDS interface for reading out data from the image sensor, and a Serial Peripheral Interface (SPI) for programming the control registers of the image sensor. Figure 6 shows how the image sensor corresponds to the hyperspectral application. Each row of the image sensor correspond to one pixel and each column a spectral component. Using the numbers from the resolution gives, for each frame, a max of 1088 pixels, each with 2048 spectral components or samples.

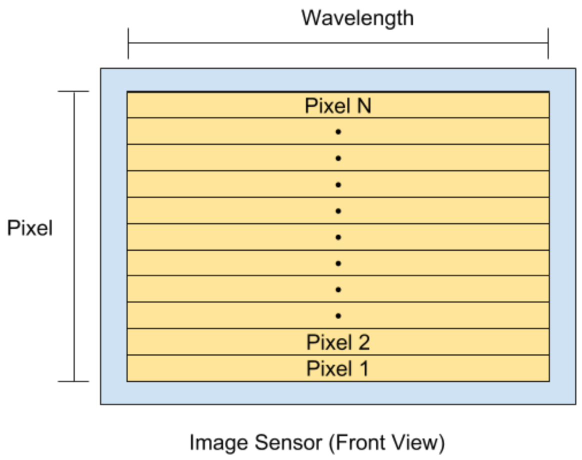


Figure 6: Relation between image sensor and hyperspectral image

4.4.1 Image Sensor Architecture

The architecture of the Image Sensor is illustrated in the block diagram shown in Figure 7. In the upper right corner of the figure we see the *Active Pixel Area*, this is a square consisting of 1088 rows and 2048 columns. In the hyperspectral application each row correspond to a pixel and each column to a spectral component. For clarification, the term *sample* is used with the same meaning as a pixel component. The image sensor has a global shutter meaning that all of the samples are exposed at once. When the samples have been exposed the specific amount of time defined by the *exposure time*, the analog values collected are written out row by row to the *Analog front end* shown in the figure. This block amplifies the sample values with a user defined gain and converts this amplified analog signal to a digital value, either 10-bit or 12-bit. Further on this digital representation of the samples is sent to the *LVDS block*. In this block the samples are converted into standard LVDS, defined by the *TIA/EIA-644A* [10], which can be transferred out of the image sensor using a LVDS interface, consisting of 18 LVDS channels; 1 clock channel, 1 control channel and 16 data channels. On the upper left side of the figure is the *Sequencer*. This block generates the required control signals to operate the sensor, based on the external input signals and the values programmed in its registers.

The *SPI* is used to read/write the registers in the *Sequencer* and to read the values from the Temperature sensor (*Temp sensor*).

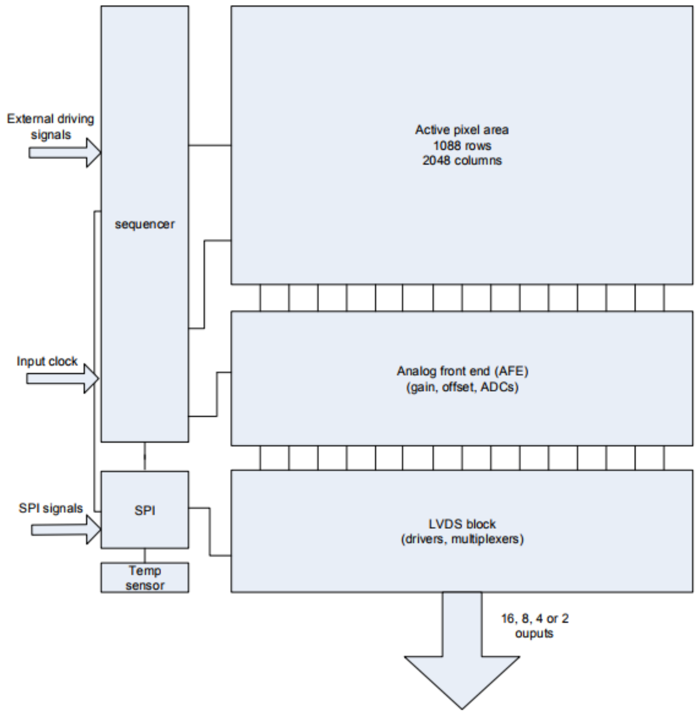


Figure 7: Block diagram of the architecture of CMV2000. [4]

4.4.2 Interfacing the Image Sensor

The Image sensor contains 10 external I/O pins which are used to control and configure the sensor from an external source. The I/O pins are as shown in table 1. *CLK_IN* is the master clock input with a frequency equal to 10 or 12 times lower than the rate of the output data, depending on the bit width that is chosen. As the maximum output data rate is 480 Mbps the CLK_IN may at a maximum be 48MHz at 10-bit and 40MHz at 12-bit. The minimum frequency is 5 MHz and all frequencies in between may be used. *LVDS_CLK_IN_N/P* is a LVDS input clock that can be used to define the output rate of the LVDS interface of the image sensor. This signal is optional as the image sensor contains an internal PLL (Phase Locked Loop) which is able to generate this clock signal internally. The PLL is set to generate this clock signal by default, also the PLL has to be disabled to use the external LVDS input clock signal. *SYS_RES_N* is the system reset and is active low. This resets the sequencer and must be active during start-up. The *FRAME_REQ* initiates the capture of a frame and can be considered the trigger if we compare the sensor to a handheld camera.

SPI_IN and *SPI_OUT* is the data input/output pins for the SPI and is used to program and read values from the image sensors internal registers. *SPI_EN* and *SPI_CLK* is the enable pin and clock pin respectively for the SPI. *T_EXP1* and *T_EXP2* are pins used in external exposure mode which will be described later in this section.

Signal name	Description
CLK_IN	Master input clock 5-48MHz
LVDS_CLK_N/P	LVDS clock, 50-480MHz.
SYS_RES_N	System reset, active low
FRAME_REQ	Frame request
SPI_IN	SPI data input
SPI_OUT	SPI data output
SPI_EN	SPI enable
SPI_CLK	SPI clock
T_EXP1	External exposure
T_EXP2	External exposure in HDR

Table 1: External inputs/outputs to Image Sensor

The datasheet for the image sensor describes some sequences in the signal flow which are recommended to prevent incorrect behaviour. These sequences are associated with the start-up of the sensor or when performing a reset. Figure 8 shows the start-up sequence. The master clock *CLK_IN* should be started after the input supply is stable. After 1μs the *SYS_RES_N* is set high, forcing the system out of reset state. As start-up may be a suited point to configure the image sensor, the SPI uploads should take place 1μs after the system exits the reset state. The system then requires some settling time before any frames may be requested. The duration of this settling time varies and according to the datasheet the main factor for this is changes to the ADC gain. For example changing the ADC gain from default value to the maximum value, which is the worst case scenario, may increase the settling time to $20ms$.

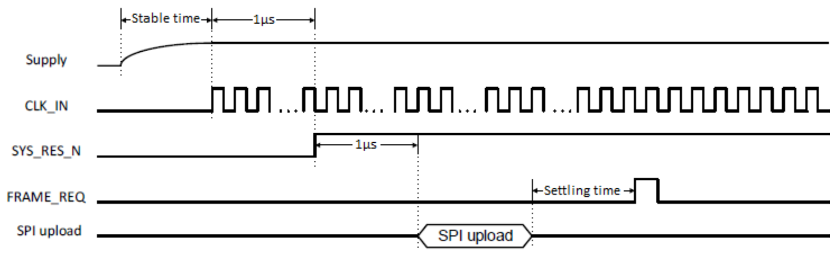


Figure 8: Start-up sequence. [4]

The reset sequence is similar to the start-up sequence except the power supply is stable and that the clock is already running. Figure 9 illustrates this situation. The *SYS_RES_N* is sampled at rising edge of the *CLK_IN* and should therefore be at least one clock period to make sure of detection.

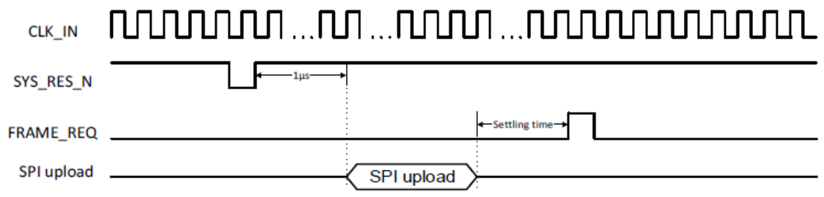


Figure 9: Reset sequence. [4]

4.4.3 SPI

As mentioned in the architecture of the image sensor, SPI is used to read and write the registers of the sequencer as well as the temperature sensor. A brief overview of what registers can be programmed and a description of these can be found in Appendix C. Writing to a register in the image sensor over SPI is shown in figure 10.

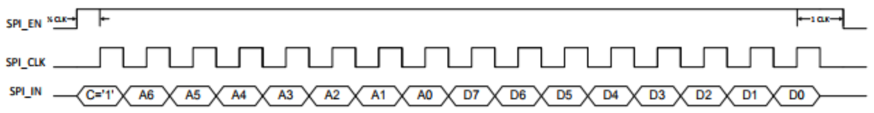


Figure 10: Example of SPI write operation. [4]

To initiate a write, the *SPI_EN* is set high, half a period before sampling the first data bit on *SPI_IN*, and stays high until one clock period after sampling the last data bit as shown in the figure. The 16 bits forming the write consists of one control bit, which is the first bit in the sequence. It tells if it is a write ('1') or a read ('0'). Then there is 7 bits forming the register address. The last 8 bits contains the data that is to be written to this register. For a read, a sequence of 8 bits has to be transferred. The control bit indicating a read ('0') and 7 bits forming the address of the register to read. Then, the clock cycle after the last address bit is sampled, the data contained in this register is outputted on the *SPI_OUT*. The read operation is shown in Figure 11.

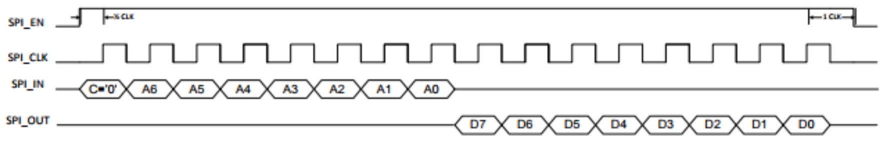


Figure 11: Example of SPI read operation. [4]

In case of multiple registers required to be written or read, this can be done in burst transfers, meaning that after the last data bit of a read or write has been sampled, a new operation may start right away. Some of the registers may not be updated, unless the camera is currently in IDLE time, meaning when the sensor is not capturing or reading out frames. Changing values in the register while a frame is captured may create unwanted effects on the image, so this should be avoided.

4.4.4 Requesting Frames

Requesting frames can be initiated by sending a pulse at *FRAME_REQ*. By default the image sensor runs through the process of exposing the samples and read out 1 frame, this is illustrated in figure 12. The exposure process consist of the exposure time where the samples are exposed followed by a frame overhead time (FOT). When the FOT is completed the sensor is ready to initiate the next frame request. This means that 1 pulse on *FRAME_REQ* produces 1 frame.

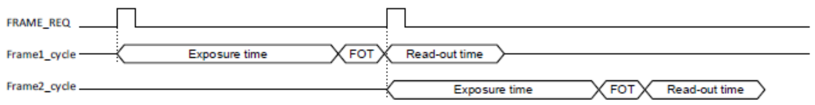


Figure 12: Default frame request. [4]

However, this process may be configured in several ways. Firstly, the user can program the number of frames the image sensor should produce when *FRAME_REG* is pulsed. This is done by programming the *Number_frames* (Figure 97) register using the SPI interface. Figure 13 shows an example of how this works when the image sensor is programmed to produce 2 frames when receiving a request.

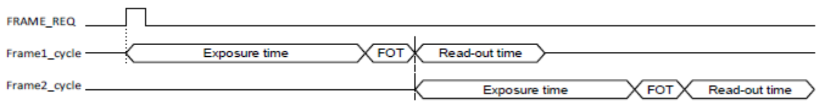


Figure 13: A frame request which produce 2 frames. [4]

It should be noticed that the read-out of the previous frame is executed in parallel with the exposure of the current frame. This means that as long as the read-out time is less than the exposure time the total time to produce a frame, which directly influences the frame rate is only affected by the exposure time.

Both of the examples presented above utilise the internal exposure mode of the image sensor. This simply means that the duration of the exposure time is programmed as a value by the user within the image sensor. However, the image sensor also contains another exposure mode called external exposure which lets the user externally program the exposure time, changing the role of the *FRAME_REQ*. In this exposure mode a new pin is introduced, *T_EXP1*. This pin is used to start the exposure time of the frame, and *FRAME_REQ* is now used to end the exposure of the frame and start the read-out of the frame. This is illustrated in figure 14.

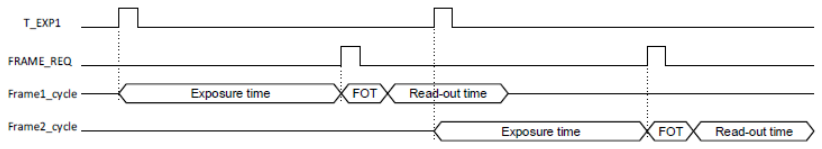


Figure 14: Frame request with external exposure. [4]

4.4.5 Reading data from the sensor

Transfer of the image data from the sensor is done through the 16 data channels of the LVDS. In addition to these channels the LVDS got a control channel and a clock channel. The clock channel outputs a clock which are synchronous to the data outputted on the data channels. The clock is DDR (double data rate), meaning that sampling of data needs to be done on both rising and falling edge of the clock. An example of this is shown in Figure 15 where a 12-bit image data is sent over one of the LVDS data channels. The least significant bit (LSB) is sent first.

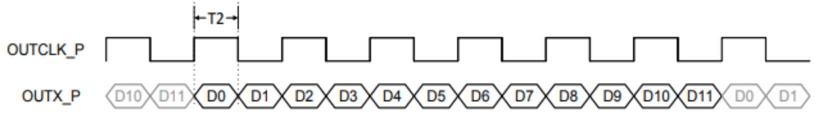


Figure 15: LVDS data channel with 12-bit image data. [4]

Data is read out from the sensor in bursts of 128 samples per channel. Between these bursts there are an overhead period that equals one period of the master input clock. If 16 of the data channels are utilised, one row will be read out for each burst. (16 channels x 128 samples = 2048). This correspond to one pixel in terms of the hyperspectral image. This is the maximum output rate and results in a frame rate of 340 fps. Figure 16 shows this behaviour for one channel. Here one can see the overhead (OH) between bursts and that one row is transferred each burst.

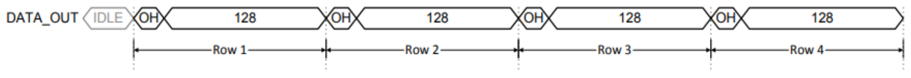


Figure 16: Timing in 10-bit mode utilising 16-channels. [4]

Only the 10-bit mode is compatible with transferring across 16 channels. If 12-bit mode is desired, a maximum of 4 channels may be used, giving a frame rate of 70 fps. This is due to a restriction from the ADC, because the conversion takes 4 times longer to complete. This means that in 12-bit mode, 4 bursts is required to output a whole row (pixel). (4 channels x 128 samples x 4 burst = 2048). This behaviour is shown in figure 17, 4 burst is required to transfer one row.

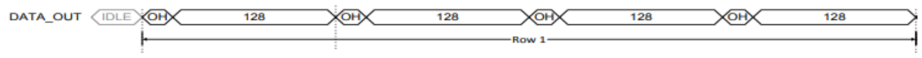
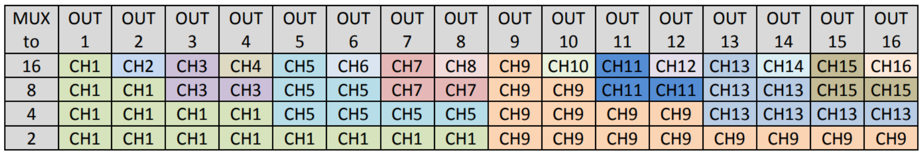


Figure 17: Timing in 12-bit mode utilising 4-channels. [4]

Figure 18 shows how the outputs behave depending on the chosen number of channels. In 16 channel mode, each output represent one specific channel. However, if less than 16 channels are utilised the outputs are multiplexed and the same channel is represented over multiple outputs as illustrated in the figure (Figure 18). Operating in the 4 channel mode, it can be seen that all 16 channels is divided in four groups, where each group outputs the same channel. As multiple outputs for the same channel may be excessive and active outputs are power consuming, the image sensor contains a feature where the user may disable outputs that are not used. The datasheet for the image sensor states that disabling unused channels is the main source for power saving in the image sensor. Reducing from 16 to 4 outputs, may save up to 33% of the power, or 216mW. Each output consumes approximately 18 mW when enabled.



MUX to	OUT 1	OUT 2	OUT 3	OUT 4	OUT 5	OUT 6	OUT 7	OUT 8	OUT 9	OUT 10	OUT 11	OUT 12	OUT 13	OUT 14	OUT 15	OUT 16
16	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8	CH9	CH10	CH11	CH12	CH13	CH14	CH15	CH16
8	CH1	CH1	CH3	CH3	CH5	CH5	CH7	CH7	CH9	CH9	CH11	CH11	CH13	CH13	CH15	CH15
4	CH1	CH1	CH1	CH1	CH5	CH5	CH5	CH5	CH9	CH9	CH9	CH9	CH13	CH13	CH13	CH13
2	CH1	CH1	CH1	CH1	CH1	CH1	CH1	CH1	CH9	CH9	CH9	CH9	CH9	CH9	CH9	CH9

Figure 18: Overview of channel outputs. [4]

Figure 19 shows the ordering of the samples at the output when utilising 4 channels. It can be seen that one pixel, consisting of 2048 samples are divided in 4 sub-rows, the first sub row is outputted at channel 1 through 4 bursts, the second sub row on channel 2 and so on.

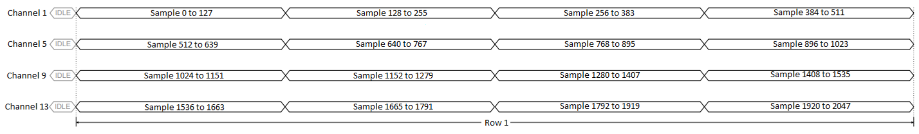


Figure 19: Sample order for 4-channel mode output. [4]

As mentioned the LVDS has a control channel. This channel is mainly used to achieve timing and synchronisation at the receiver side. Table 2 shows an description of the 12 bits forming the control word transferred on this channel. Only the three first signals are required to know when the data is valid, the rest of the signal is pure informational. The $DVAL$ signal is always high during a burst of valid data and low between bursts. The $LVAL$ is high during the read-out of a whole row, meaning that it is only low between bursts that separates the read-out of a row. Finally the $FVAL$ is high during the read-out of a whole frame and is set low between bursts that separates frames. This means that $DVAL$ can be used to identify when there is a valid burst of pixel data on the output. The $LVAL$ is used to identify when a row has completed read-out and a new row is initiated and the $FVAL$ the same for frames.

Bit	Function	Description
[0]	DVAL	Indicates valid pixel data on the outputs
[1]	LVAL	Indicates validity of the read-out of a row
[2]	FVAL	Indicates validity of the read-out of a frame
[3]	SLOT	Indicates overhead period before 128-sample burst
[4]	ROW	Indicates overhead period before the read-out of a row
[5]	FOT	Indicates when the sensor in in FOT
[6]	INTE1	Indicates when samples of integration block 1 are integrating
[7]	INTE2	Indicates when samples of integration block 2 are integrating
[8]	'0'	Constant zero
[9]	'1'	Constant one
[10]	'0'	Constant zero
[11]	'0'	Constant zero

Table 2: Bits of the LVDS control channel [4]

4.5 LVDS receiver

4.5.1 Design Criterias

According to the requirements from the NTNU Smallsat the Image Sensor is desired to be used in 12-bit mode utilising 4 of the LVDS data channels. Therefore the focus will be on this mode of operation when presenting the existing solution in this section, but also when presenting the implementations that was done in this thesis in the *Method* section.

4.5.2 Simple Deserialiser

In the paper *Understanding Serial LVDS Capture in High-Speed ADCs* [9] a scheme for capturing a LVDS signal on a receiver is presented. It is shown in figure 20 and consists of a double data rate (DDR) flip flop which is fed into a shift register.

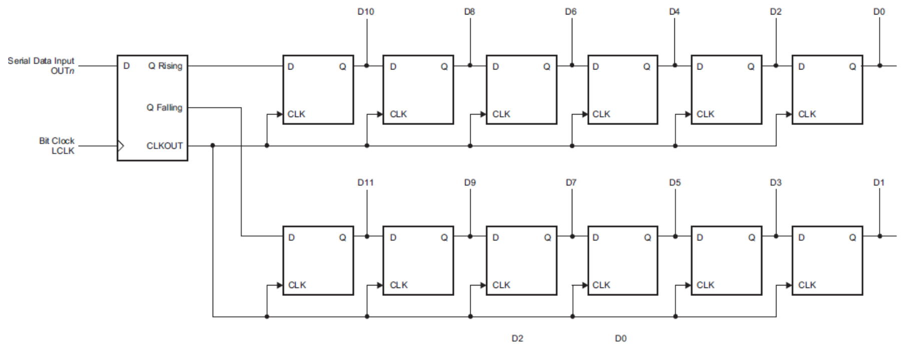


Figure 20: Scheme for capturing LVDS. [9]

At the *Q Rising* and *Q Falling* in Figure 20, the values captured on the rising and falling edge of LVDS clock (*Bit Clock*) is outputted. Because both the registers connected to the *Q Rising* and *Q Falling* are connected to the same clock source *CLKOUT* they are updated at the same time. This means that the output *Q Falling* is changing half a clock period before it is captured into the register. This re-latches the falling edge values and synchronises the values captured at rising and falling edge. Figure 21 illustrates the mechanism of this design in a waveform. The *LVDS* signal is sampled at rising and falling edge of the *DDR_Clock*. The inputs to the registers *reg_rising* and *reg_falling* samples the *Q Rising* and *Q Falling* at the same rising edge, making these two inputs synchronised.

In *Xilinx user guide7 Series FPGAs SelectIO Resources* [17] a DDR register called Input DDR (IDDR) was found to be available on all 7 series FPGAs. According to the user guide this register has three modes of operation. *OPPOSITE_EDGE*, *SAME_EDGE* and *SAME_EDGE_PIPELINED*. The *OPPOSITE_EDGE* mode outputs the values of *Q1* and *Q2* as they are sampled. This

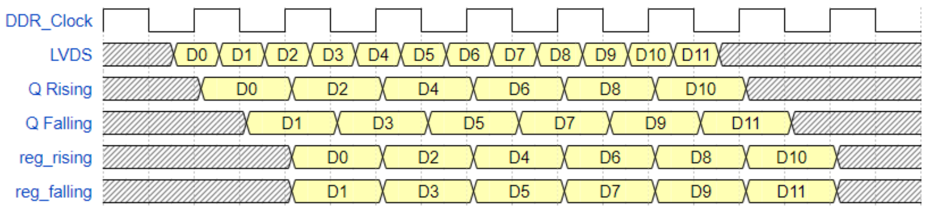


Figure 21: Waveform of LVDS capture.

means that *Q1* is set at rising edge and *Q2* at falling edge. This is illustrated in figure 22.

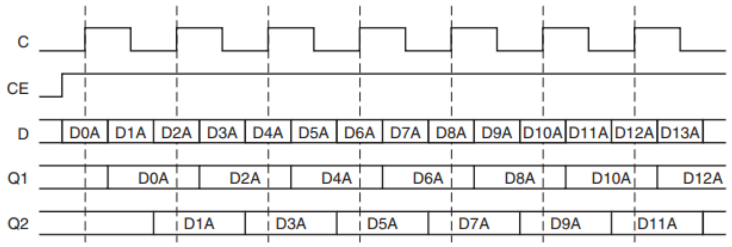


Figure 22: OPPOSITE_EDGE mode [17]

SAME_EDGE sets *Q1* and *Q2* at rising edge starting from the first rising edge. However because the first falling edge has not yet been sampled at the first rising edge the values of the rising edge and falling edge will not be synchronised. This means that the value of the first falling edge will not be outputted on *Q2* before the second rising edge. This behaviour is shown in Figure 23.

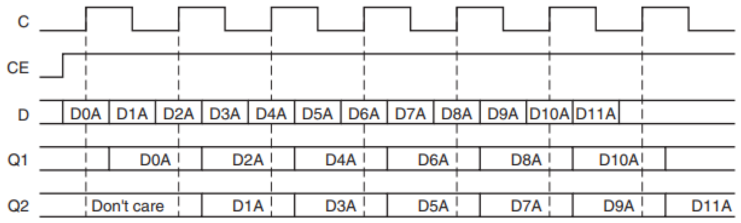


Figure 23: SAME_EDGE mode [17]

The last operation mode, *SAME_EDGE_PIPELINED*, was chosen for this implementation. In this mode the output of the first rising and falling edge is set at the next rising edge. In this case the outputs will be synchronised as pairs and this makes the rest of the design easy to implement. Figure 24 shows how this mode operates.

However, as LVDS signals are operating at a high speed, they are vulnerable to latencies. In *Serial LVDS high-speed ADC interface* [5] the author explains

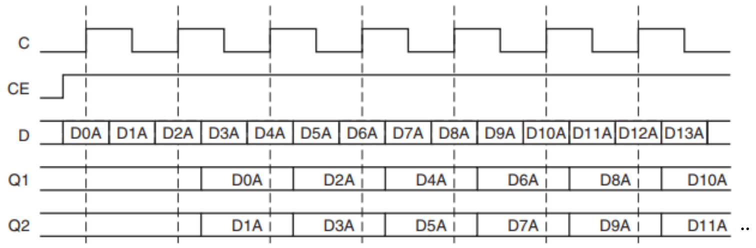


Figure 24: SAME_EDGE_PIPELINED mode [17]

why this is a problem and why the simple scheme for capturing LVDS shown in Figure 20 in many cases is not sufficient enough. As both the LVDS data and clock are sent over independent LVDS channels it is important that these arrive synchronised enough at the receiver side. This is important so the receiver does not sample the data during a transition. However, as the data and clock channels typically experience different amount of delays this is not as simple as one should hope. In an ideal world the data transitions on the LVDS would occur instantly and the risk of sampling the data during a transitions is not present. However, in the real world this is not the case and Figure 25 illustrates this. The moment when data is stable on the channel and can be read is called the *Data Eye*, and is the desired place to sample the data, as shown with red dotted lines in the figure. If sampling is done inside a *bit transition* the result is unpredictable and may cause a corrupt sample.

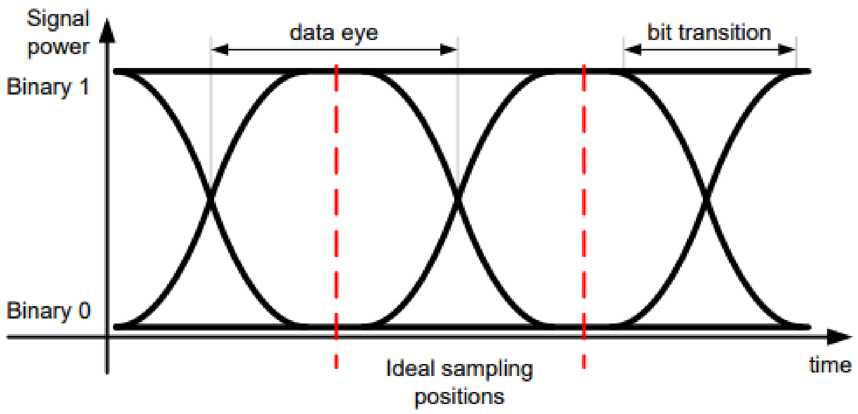


Figure 25: Data eye sample diagram. [7]

Figure 25 shows why the LVDS capturing method shown in figure 20 may not work. It is not able to synchronise the data and clock channels if they arrive with a skew. Another approach utilising primitives *IDELAYE2* and *ISERDESE2* from Xilinx is found. Starting with a closer look at these primitives, this approach will now be presented.

4.5.3 IDELAYE2

Figure 26 shows the block diagram of the *IDELAYE2* primitive which is a programmable delay block. It has 4 different operation modes, *FIXED*, *VARIABLE*, *VAR_LOAD* and *VAR_LOAD_PIPE*. Modes *FIXED* and *VARIABLE* will be used and therefore these operations will be presented.

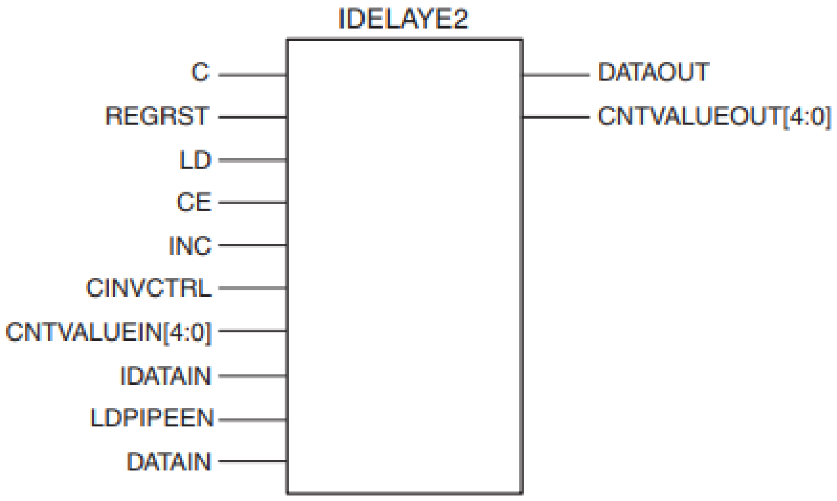


Figure 26: Block diagram IDELAYE2. [17]

In the *FIXED* mode the block is simply programmed with a fixed delay. This passes the input to the output with a delay corresponding to this fixed value. In the *VARIABLE* mode the delay can be varied. By assigning a '1' or '0' on the *INC* input together with a '1' on the *CE* (enable) the delay will be increased or decreased respectively. This increase or decrease has a unit called *taps*. There are in total 31 taps, and the resolution of these are decided by another module *IDELAYCTRL*. This is a independent module but has to be present in the same clock region as the *IDELAYE2* to calibrate the taps. This module has two inputs *REFCLK*, *RST* and one output *RDY*. The frequency of the clock signal inputted on the *REFCLK* decides the tap precision. Frequency of 200 MHz gives a tap precision of $\sim 75ps$.

4.5.4 ISERDESE2

Figure 27 shows a block diagram of the other Xilinx primitive, *ISERDESE2*. This is a serial-to-parallel converter which is suited to serialise the LVDS channels from the image sensor. The serial stream are inputted at either *D* or *DDLY* depending on if the stream comes from a FPGA IOB resource or an *IDELAYE2*.

In this case it will be connected to the delay module, *DDLY* will be used. The parallel data will be outputted at the Q1-Q8 outputs, with Q1 MSB.

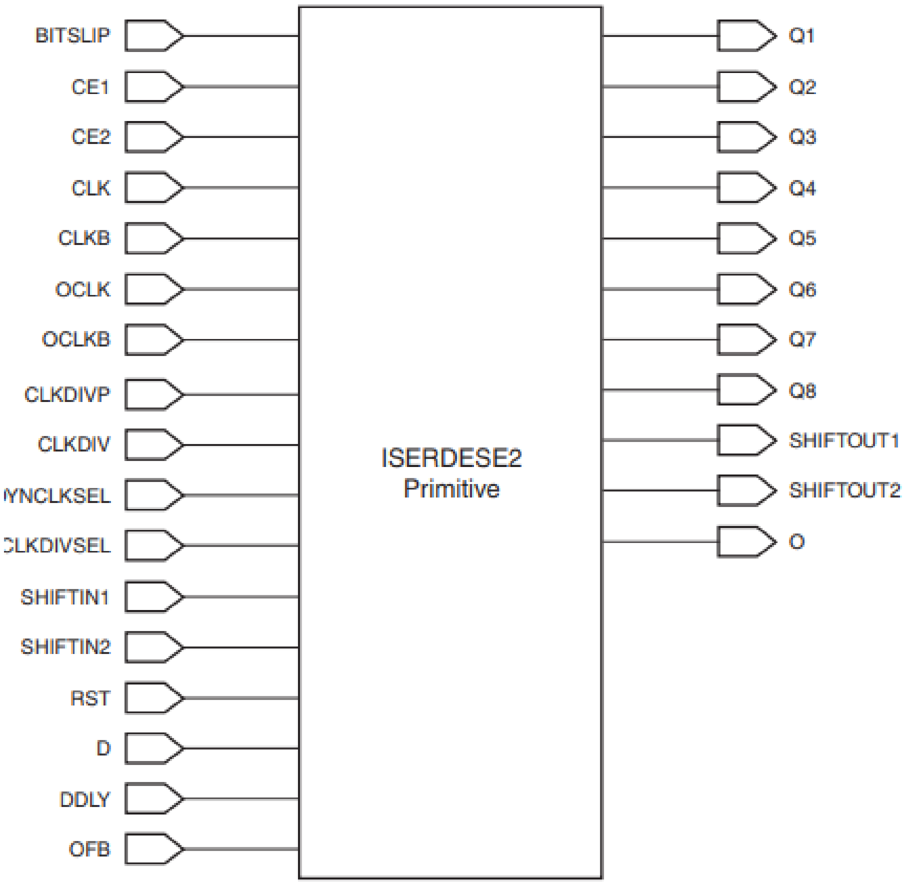


Figure 27: Block diagram ISERDESE2. [17]

This primitive can be operated in SDR and DDR mode and each module are able to deserialise up to 8 bits. However, two modules can be chained into outputting 10 or 14 bits. As the image sensor is capable of outputting 12 bits this it is not compatible with the way these primitives are meant to be used according to the Xilinx documentation [17], but a workaround was found in *Serial LVDS High-Speed ADC Interface* [5]. In this paper two *ISERDESE2* modules are used in SDR mode. One for capturing the rising edge values and one for capturing the falling edge values of the DDR clock. Two designs, one using the primitives according to the Xilinx documentation and the other one using the workaround mentioned above will be presented later in this section. It should also be mentioned that the *ISERDESE2* primitive has a feature to align the bits captured. As it can be hard to synchronise the sampling of the data words initially, *ISERDESE2* has a input called *BITSLIP* which helps with this. Lets assume that we have a training pattern of 2770 (b"1010 1101 0010") that are inputted to the primitive as shown in figure 28. However, the receiver may have captured another order of the bits than is expected. By using the *BITSLIP* the receiver *slips* bits to achieve the correct order. In SDR mode this is done by shifting the order to left by one, in DDR it is alternating between shift right by one and shift left by three.

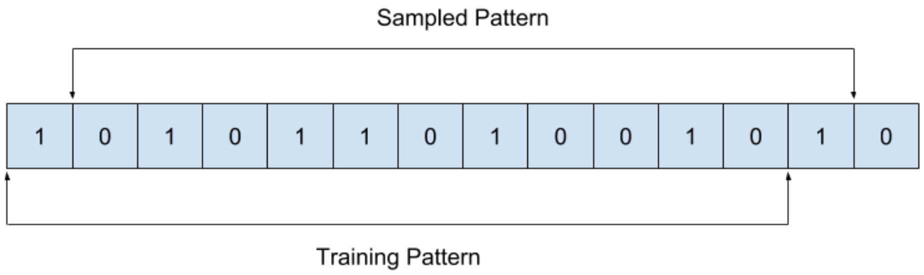


Figure 28: Example when bitslip are needed.

4.5.5 Existing LVDS receiver designs

Two different designs utilising the described primitives suitable for the application in this paper were found.

4.5.5.1 Design 1 First in "16-Channel, DDR LVDS Interface with Per-Channel Alignment" [3] an implementation for a 16 channel DDR LVDS receiver is presented. This implementation is designed for data words of 8-bit but could be adapted to a 10 bit version to be applicable with the image sensor. However, this design is not directly compatible if the image sensor is operating in the 12 bit mode. Figure 29 shows a block diagram of this implementation. The LVDS signals are inputted on the right side, through the *DATA_RX* and the clock through the *CLOCK_RX*. All LVDS inputs are converted to a single ended signal and fed into a *IODELAY* primitive.

The clock signal delay has a fixed value and the data signal delays are variable and are configured by the *Bit Align Machine* to make sure the sampling occurs within the data eye of the signals. From the delay modules the signals are converted from serial to parallel data in the *ISERDES* primitive. As the *ISERDES* blocks used in this design is restricted to 6-bit, two of this primitives has to be connected in a Master-Slave chain to be able to process 8 bits. The *ISERDESE2* available in the Zynq-7000 series, can process 8-bit data before a chain of two modules are needed. The *Bit Align Machine* monitor the outputs of the *ISERDES* modules to configure the delay blocks as well as bit ordering by using the *BITSLIP* functionality mentioned before. To avoid implementing a *Bit Align Machine* for every channel, a *Resource Sharing Control* block is

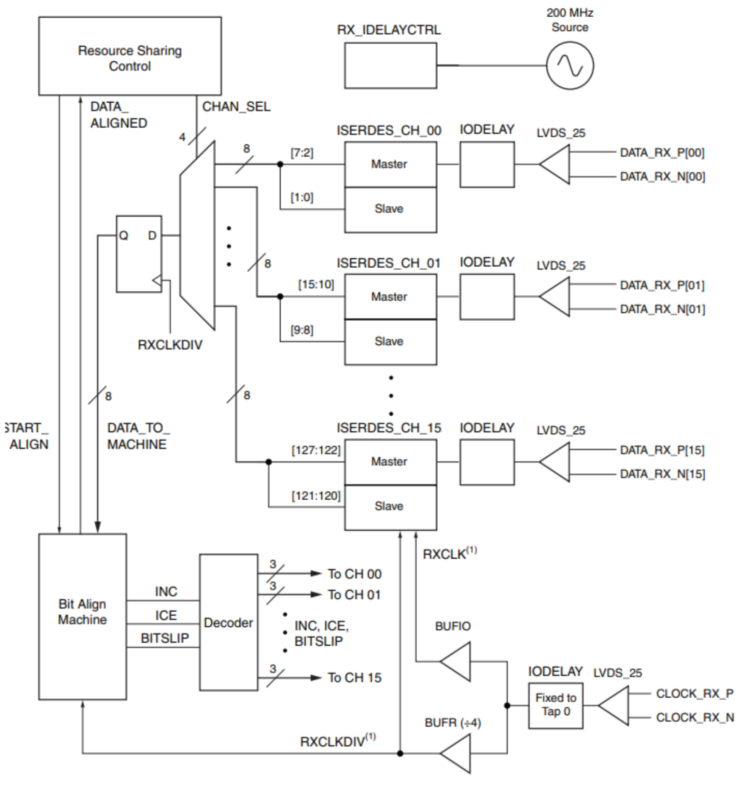


Figure 29: Block diagram of LVDS receiver. [3]

distributing access to the *Bit Align Machine*. At the top of the figure there is an independent *IDELAYCTRL* primitve which is used to calibrate the *taps* for all of the *IODELAY* modules. It should also be mentioned that the clock from the LVDS clock channel is divided into two clocks. One with the same frequency as the LVDS clock (*RXCLK*), and one with the same period as the bit width, which in this case is the LVDS clock divided with 4 (*RXCLKDIV*). *RXCLKDIV* is used by the *ISERDES* to output the parallel data as well as to sample the *BITSLIP* input.

4.5.5.2 Design 2 The next design is taken from *Serial LVDS High-Speed ADC Interface* [5] combined with the knowledge of users on Xilinxs own forum. This design was created as a workaround to achieve a 12-bit LVDS receiver as this is not supported by the primitives according to Xilinx documentations [17]. The interesting part of this design is how the *ISERDESE2* primitive is used. Figure 30 illustrates this for one LVDS data and clock channel.

The idea with this design is to have both *ISERDESE2* primitives in SDR and to use one for sampling rising edge values and one for sampling the falling

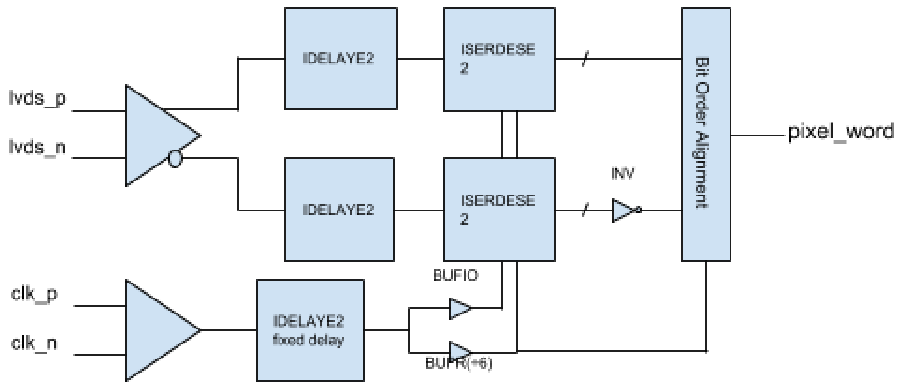


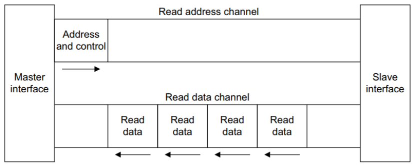
Figure 30: Block diagram of LVDS receiver.

edge values. It can be seen in the figure that instead of converting the signal into a single ended signal, it keeps the differential form where the p and n side are inputted directly to two different *IDELAYE2* primitives. From here they are sampled in the *ISERDESE2* blocks, the p side on rising edge and the n side on falling edge of the clock. It should be mentioned that the lower *ISERDESE2* block in the figure requires an inverted clock input. This way of sampling the data will produce an inverted result from the *ISERDESE2* on the n-side and it is therefore required to invert the output. In the end the even and odd bits of the words will have to be correctly ordered to form the *pixel word* on the output. Also the part calibrating the *IDELAYE2* to make sure sampling happens inside the data eye has been omitted but this is done in a similar way as the previous design. It can be seen that the clock divider in the figure, divides the clock by 6, this is an example for a 12-bit implementation and should be changed to 5 for an 10 bit version. Because this version is adaptable to both 10 and 12 bits it is suited to use in an implementation with the image sensor.

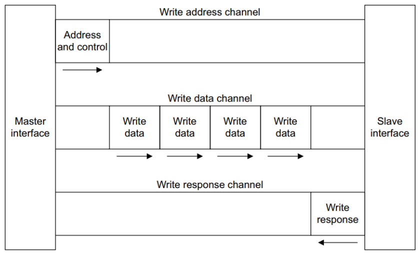
4.6 AXI protocols

A bus protocol that will be used in the implementations of this thesis is the Advanced eXstensible Interface (AXI) developed by ARM. This protocol defines a set of rules of how transactions of data between parts of a design should be executed.

This is a burst-based master-slave protocol and consists of five independent channels. **Read address**, **Read data**, **Write address**, **Write data**, **Write response**. During a read or a write the master provides the read or write address to the slave through the corresponding channels. The *Read data* and *Write data* channels are then used to transfer the data to read or write. After a write the *Write response* channel is used by the slave to inform the master about the status of this operation. Figure 31 show the mechanics of AXI read/write operations.



(a) AXI read operation. [1]



(b) AXI write operation. [1]

Figure 31: AXI read/write operations

The AXI protocol also use a simple but concise handshaking process. Using two signals, **VALID** and **READY**. The **READY** signal is asserted when the receiving part is ready to receive data and the **VALID** is asserted by the sender to indicate that valid data is present on the channel. When both signals are asserted it is called a handshake and the transfer is performed. In addition there are some rules for asserting the **VALID** and **READY**, these can be found in Appendix E.

There is three different AXI bus interfaces available, AXI, AXI-Lite and AXI-stream. These fulfil different use cases but consist of the same fundamentals. AXI is suited for memory mapped communication with high performance requirements. AXI-Lite is a simpler version of AXI where high performance is not crucial. It is fitted for low-throughput memory-mapped communication. AXI-stream is a interface for high speed streaming of data. It has been released multiple versions of AXI where the newest version is called AXI4 released in 2010. This is the version that will be used in this thesis.

4.7 Cube DMA

An important module for distributing data between the memory and different hardware designs is the Direct Memory Access (DMA). This module is commanded by the CPU to initiate data transfers from and to the memory, while the CPU executes something else. This is an important mechanism as the hyperspectral cube consists of large amount of data that will be transferred around to different hardware accelerators. These transfers will occupy the CPU if no DMA is present.

This concludes that a DMA will be necessary in the hyperspectral imaging application. However, there are some requirements that the DMA must be able to fulfil. In the paper *Direct Memory Access for Hyperspectral Imaging Applications* [6] the author looks into different DMA solutions for hyperspectral imaging applications. He starts by exploring the available solutions on the Zynq-7000 platform. He finds that the existing DMA solutions, *AXI DMA*, *AXI DMA* *in 2D mode* and *Video DMA* are not suited for this application and designs a custom DMA he calls *Cube DMA*. He emphasises that the DMA should be able to access the cube sequential, block-wise and plane-wise, a requirement none of the existing DMA solutions was able to meet. The *Cube DMA* is able to provide all of these transfers and will therefore be the preferred DMA solution in light of hyperspectral imaging. The Cube DMA transfer uses two independent channels to tranfer data over AXI stream, mm2s(memory to stream) and s2mm(stream to memory). Tables 19-20 in Appendix D shows the register map for the registers that was used in this thesis for both mm2s and s2mm channels.

4.7.1 Pixel Order Alignement Module

To store the LVDS data in memory it has to be passed to the *Cube DMA* through an AXI4-stream interface. Also the samples of the pixel will need to be organised in such a way that the *Cube DMA* is able to store them in the desired order in memory. The desired order is that the pixels are stored sequentially, meaning that all components from first pixel are stored first and then all components from the next pixel and so on. This means that the pixels will have to be passed to the *Cube DMA* in this order. From the previous figure 19 it can be seen that the LVDS data is not transferred in the correct order and therefore a mechanism to reorder and pass data through an AXI-stream interface will have to be implemented. Figure 32 shows the principle of the operation required from this module. In this figure, 16 samples from the LVDS Deserializer arrives. One sample from each *LVDS* channel arrive each clock cycle, meaning that 1, 5, 9 and 13 arrives in the first cycle. The right side of the arrow shows the desired ordering on the output. It is clear that to be able to output these samples in the correct order, one first has to wait 4 cycles to output all samples from *lvds_1* before starting on *lvds_2* and so on. This means that samples has to be buffered until the sample before has been outputted.

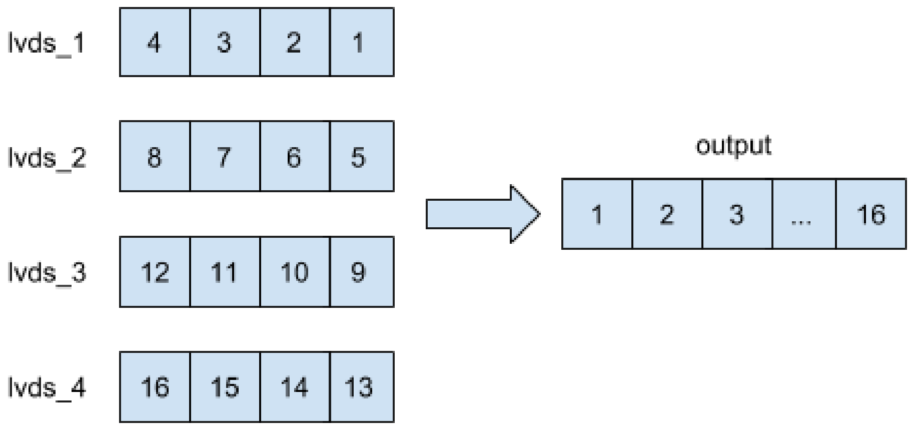


Figure 32: Operation required on the output of LVDS receiver.

4.8 Binning

A technique that is interesting in terms of hyperspectral imaging is binning. This is a technique where one try to reduce minor observation errors as well as reducing the amount of data without loosing too much information. The principle of binning is that multiple samples in a data set are merged together and given a new value based on the values of the individual samples. This value could be calculated as the summation, mean, median or other measures of the original values depending on the application. For the hyperspectral imaging application binning of the components of the pixels is useful, by merging multiple pixel components into a average of these samples. This because the lens that is used captures wavelengths in the interval 400-900 nm and 2048 components are not required to cover this. Therefore, by binning spectral components together reaching a point where there is enough components to represent the interesting data would be reasonable. The number of pixel components that will be merged together will be called the *Binning Factor*.

5 Extended Multiplicative Signal Correction

The Extended Multiplicative Signal Correction (EMSC) is a model-based pre-processing technique derived from the Multiplicative Signal Correction (MSC), and is used to reduce the impact of other phenomenons than the components that are of interest when capturing a spectra using spectroscopy. This could be phenomenons such as noise in form of light scattering or errors due to the instruments that are used. In the SmallSat Project the hyperspectral images are captured by a hyperspectral camera on a satellite. As the camera is capturing images of the earth from the space, light scattering may impact the images as the light captured travels through the clouds or other materials. Also the camera itself may contain sources inflicting the captured image in a erroneous way. This means that EMSC may be highly relevant for this application.

In the papers *Light Scattering and Light Absorbance Separated by Extended Multiplicative Signal Correction. Application to Near-Infrared Transmission Analysis of Powder Mixtures* [8] and *Extended multiplicative signal correction in vibrational spectroscopy, a tutorial* [11] the authors demonstrate the potential of EMSC in the Near Infrared Transmission spectra. In the former the authors explains that by using EMSC they are able to analyse different powder mixtures by separating the physical light scattering by the the chemical light absorption. In other words, when doing vibrational spectroscopy, each chemical bond in the material analysed emits unique vibrational energy levels, which is sampled and can be used as a fingerprint for this specific chemical bond [12]. However, in addition to these fingerprints, phenomenons as those mentioned above usually disrupts the information in these samples. It is here EMSC comes in, separating these fingerprints with light scattering and other sources of noise that are also captured.

5.1 Mathematical Model

In the paper *Light Scattering and Light Absorbance Separated by Extended Multiplicative Signal Correction. Application to Near-Infrared Transmission Analysis of Powder Mixtures* [8] the authors presents a way of calculating the EMSC for a given data set. Using Beer-Lambert's law as a starting point the authors states that the theoretical chemical absorbance spectrum of a sample can be considered as a sum of all contributions to this spectrum by all the constituents contained in the sample. This can be shown with equation 1, where $z_{i,chem}$ is the chemical absorbance spectrum for the sample i, c is the concentration and $\mathbf{k}$ the vector representing the absorptivity spectrum of the j'th constituent contained in the sample.

$$\mathbf{z}_{i,chem} = c_{i,i}\mathbf{k}_1' + ... + c_{i,j}\mathbf{k}_j' + ... + c_{i,J}\mathbf{k}_J' \tag{1}$$

The idea with the EMSC is to correct the measured absorbance spectrum by removing light scattering, path length and other wavelength dependent spectral effects, giving a result containing only the chemical absorbance information. The EMSC model is shown in equation 2 which is used to approximate the physical effects related to light scatter variations. The $\mathbf{z}_i$ is the measured absorbance spectra of sample i. a_i and b_i are coefficients representing baseline offset and path length relative to the baseline offset and path length in a reference spectrum. λ is the wavelength. Because light-scattering effects depends on the wavelength, both a linear and quadratic term of the wavelength is taken into the account together with coefficients d and e allowing for wavelength dependent spectral variations from sample to sample.

$$\mathbf{z}_i \approx a_i + b_i \mathbf{z}_{i,chem} + d_1\lambda + e_1\lambda^2 \tag{2}$$

Then by estimating the coefficients in equation 2 this can be used to find the EMSC correction necessary to subtract all except the chemical absorbance information from the test data. This is shown in equation 3.

$$\mathbf{z}_{i,corrected} = (\mathbf{z}_i - a_i - d_1\lambda - e_1\lambda^2)/b_i \tag{3}$$

Even though the calculations are straight forward as shown in the equations above. The estimations of the coefficients are not. The authors presents a versatile solution for calculating the coefficients and applying the correction on the measured absorbance spectra. The solution starts by constructing a matrix $\mathbf{M}$ as follows:

$$\mathbf{M} = [1; m; k'; \lambda; \lambda^2] \tag{4}$$

The first row consists of ones and is according to the authors, introduced because of matrix formalities. Then $\mathbf{M}$ contains the mean of the reference spectra m and the reference spectra itself k', the wavelength λ and wavelength squared λ^2. Now the EMSC coefficients can be estimated using least squares regression as shown:

$$\mathbf{z}_i = \mathbf{p}_i\mathbf{M} + \epsilon_i \tag{5}$$

where $\mathbf{p}_i$ is a vector containing the coefficients that we estimates as follows:

$$\mathbf{p}_i = \mathbf{z}_j\mathbf{V}\mathbf{M}'(\mathbf{M}\mathbf{V}\mathbf{M}\text{'})^-1 \tag{6}$$

This is weighted least squares estimator where $\mathbf{V}$ is weights that can be defined by the user based on different criteria. This finally gives the solution that is an approximation spectra containing only the chemical absorbance information. Rearranging the terms in equation 5 gives equation 7 where ϵ is the EMSC corrected spectra

$$\epsilon_i = \mathbf{z}_j - \mathbf{p}_i\mathbf{M} \tag{7}$$

6 Method

6.1 Exploring the prototype

Some testing of the prototype has been done. Starting with setting up the hardware in Vivado and mapping different peripherals such as SPI, UART, USB according to the schematics [14]. However, it was discovered that the clock signal from the image sensor is mounted to a pin on the high performance bank on the Zynq-7030 only able to operate at 1.8V. According to the datasheet for the Image Sensor [4] the digital I/O minimum voltage on the image sensor is 3V, which makes them incompatible to each other. Therefore a decision was made to make everything work on a ZedBoard. This is a development board that are similar to the *PicoZed*, containing a Zynq-7020 FPGA. The only disadvantage with this is that it wont be possible to get actual LVDS data from the Image Sensor, but as this is already incompatible in its current mapping to the Picozed on the Prototype this makes no difference.

6.2 LVDS Receiver

Figure 33 shows an overview of the pipeline for the implemented design. The LVDS receiver contsist of a *Deserialiser* and a *Pixel Order Alignment Module*. Then the output of the *Pixel Order Alignment Module* is connected to the *Binning Module* if binning is desired, if not the *Pixel Order Alignment Module* should be connected directly to the *Cube DMA*. The *Cube DMA* then stores the values in memory.

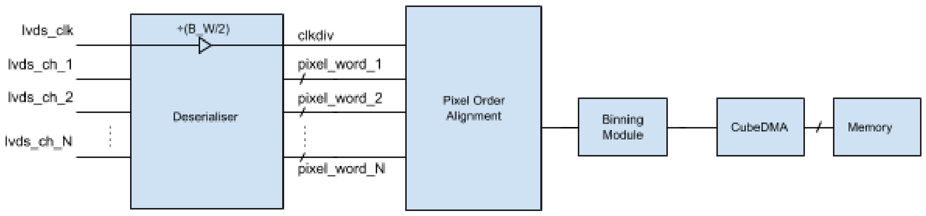


Figure 33: Overview of the implemented design.

6.2.1 Simple Deserialiser

A simple LVDS Deserialiser is implemented according to the design shown in Figure 34 to receive and convert the single bits received from the image sensor to 12 bit words that are sent further through the pipeline. This module was implemented according to the theory presented in the background section showed in Figure 20.

The sampled data from the DDR register will be fed into two shift registers, one for the rising edge data (*Q1*) and one for the falling edge (*Q2*). When the registers has been filled with all the bits corresponding to one data word from

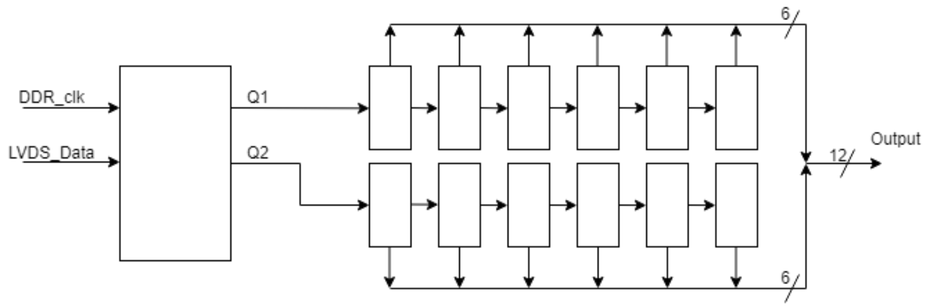


Figure 34: LVDS implementation

the image sensor, the bit word will be outputted from this sensor. There will be control logic in addition to what is shown on the figure. This is necessary to synchronise when to output the content of the shift registers, as well as handling the control bits on the LVDS control channel which dictates if the LVDS channels should be sampled. Also as the image sensor may output both 12 or 10 bit data this module should be able to handle this.

As was mentioned in the background section, this simple LVDS receiver might not be suited because of the nature of the LVDS signals. Adding a delay module such as the *IDELAYE2* at the clock and data inputs should give the opportunity to synchronise these channels, however it will not be able to handle the situation where the bit order is wrong.

6.2.2 LVDS Deserialiser with Xilinx Primitives

Figure 35 shows the LVDS deserialiser implementation able to handle some of the more tricky problems that are related with LVDS, such as bits out of order and to sample in the data eye. It is based on the *Design 2* from the background section with some additional logic to handle the control channel that the image sensor offers. It can be seen on the Figure (35) that it is only *DVAL* from the control signals of the image sensor that are required in this module, however, the necessary logic to read out all of the signals are available. Also a signal *valid_out* is added on the output to indicate to the receiver of this data if it is valid or not.

To make sure the design outputs the bits in the correct order, a state machine that trains the design prior to transferring data was implemented. This is done by applying a known pattern called *training patters* across all the LVDS channels. Then the state machine works on each channel one by one, by applying a bitslip until the known pattern is captured. When the parallel data output of all channels equals the training pattern one knows that the design outputs the correct bit order. There is not a dynamic implementation to make sure that sampling is done inside the date eye. This may however not be a problem, if it turns out to be, the user could measure where sampling takes place and then set the *IDELAYE2* in *FIXED* mode and manually add delay to move the sampling

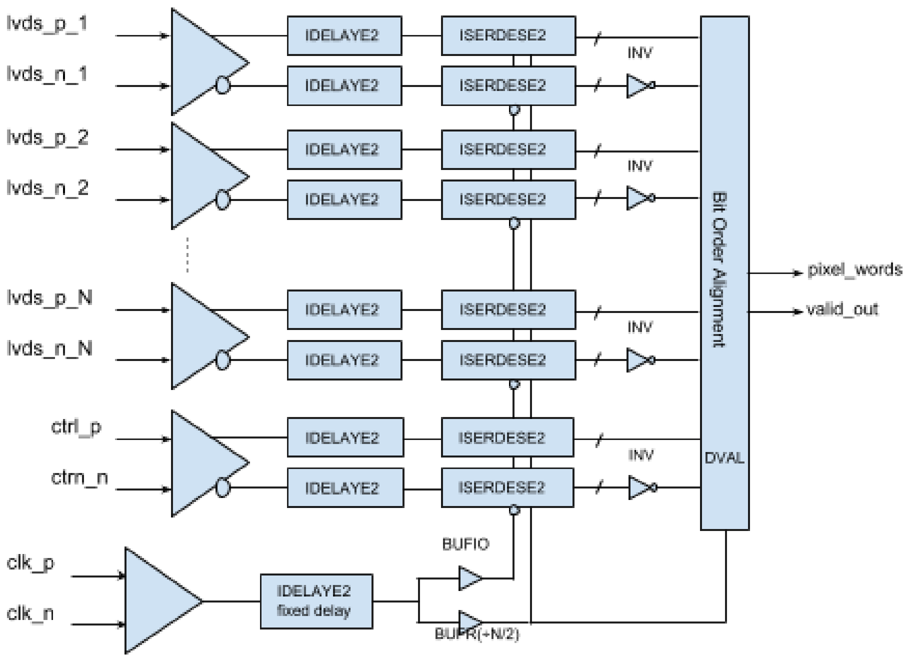


Figure 35: Block diagram of implementation of LVDS receiver.

inside the data eye.

6.2.3 Pixel Order Alignment

Because the sensor is going to be used in the 12-bit mode a Pixel Order Alignment module for 12-bit was implemented. The reasons why a 10-bit mode version was not implemented, and proposed solutions to how it can be done will be reflected upon in the discussion section.

As was shown in Figure 33 the output of the *Pixel Order Alignment Module* may be directly connected to the *Cube DMA* or through the *Binning Module*. This led to a choice that the *Pixel Order Alignment Module* will be designed to fulfil the *Cube DMA* requirements and then the *Binning Module* is adapted to work with the resulting *Pixel Order Alignment Module*. As shown in the background section in figure 19 the samples are transferred across multiple LVDS channels. This means that a row of pixel components are divided in a number of sections equal to the number of LVDS channels that are utilised. To reconstruct the pixel at the receiving side, the pixel components arriving across the channels has to be aligned. As was shown in figure 32, the first sample to arrive on the second channel has to be passed to the output after the last sample received on the first channel and so on. This is what the *Pixel Order Alignment module* will handle. This module will require the ability to buffer sample values until the sampels prior in the correct order has been outputted. The module will also require an AXI-Stream interface to communicate with the *Cube DMA* directly. Another challenge is that the LVDS deserialiser side and the *Cube DMA* side operate on different clock frequencies meaning that they will have to be synchronised. Figure 36 shows a block diagram of the connection between the *Deserialiser* and the *Pixel Order Alignment* modules.

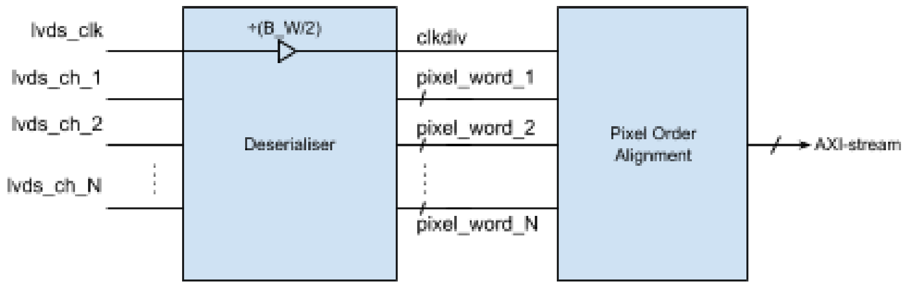


Figure 36: Block diagram of pipeline from LVDS receiver

The Image sensor outputs 12-bit data with a LVDS DDR clock of 240 MHz. This means that the parallel data (*pixel_word_n*) from the *Deserialiser* will have an output frequency of $\frac{240MHz}{12bit/2} = 40$MHz. Signal *clkdiv* is a clock synchronised with these outputs, which is used as a sampling clock in the *Pixel Order Alignment module*. The *Cube DMA* operates at 100MHz. This was synchronised by using asynchronous FIFOs. FIFOs will also satisfy the buffer requirement. Figure 37 shows a block diagram of the proposed implementation.

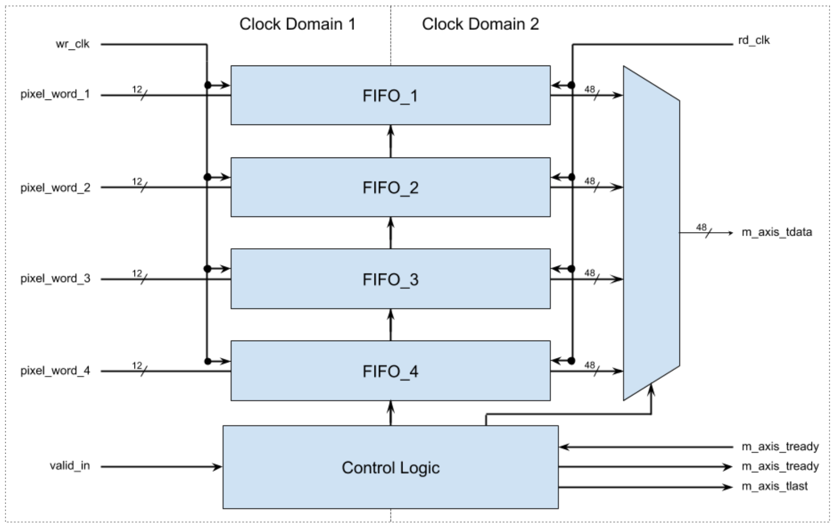


Figure 37: Block diagram of implementation

On the left side the inputs from the *LVDS deserialiser* are connected. On the right side is the AXI-stream interface. The FIFOs that are used is a premade Asynchronous FIFO from Xilinx [20]. The control block consists of a state machine to correctly output the pixel components in the correct order. The procedure of this state machine is as following. First, all of the FIFOs are filled with a number of elements equal to one pixel, which is fixed at 2048 elements from the image sensor. This means 512 elements in each FIFO. When all FIFOs are filled, the elements are passed to the AXI interface in the correct order, which is all elements in FIFO_1 then all elements in FIFO_2 and so on until FIFO_4. One crucial requirement for this to work is the size of the FIFOs. Because it will be added new samples to the FIFOs while they pass values to the output it will be necessary that they can store these values. The receiving side operates at 100MHz and the *Cube DMA* has the capacity to transfer 64 bits each cycle, which equals $5\frac{1}{3}$ samples. For simplicity it will be configured to transfer 4 samples, or 48 bits, each cycle because this adds up with the total number of samples that are stored in the FIFOs. Figure 38 illustrates transfer of 4 and 5 samples each cycle. It can be seen that in the lower case the transfers do not add up resulting in having to transfer 3 samples from the next FIFO. To avoid the extra logic of this operation transfers of 4 samples was found suited.

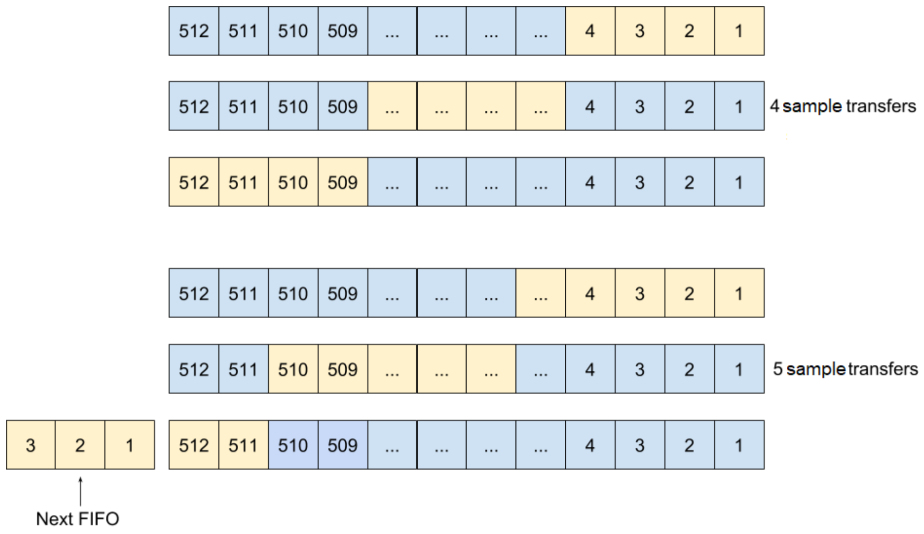


Figure 38: Difference between 4 and 5 sample transfer.

To estimate the required FIFO sizes of this module the data rates of input and output were used. The input to the *Pixel Order Alignment Module* is connected to the *Deserializer* and 12-bit samples arrive across all 4 channels with a frequency of 40 MHz. The output is connected to the *Cube DMA* which operates at 100 MHz and is configured to transfer 4 samples each cycle. This results in a input data rate of $12bit \times 4channels \times 40MHz = 1920Mbps$ and a output data rate of $4samples \times 12bit \times 100MHz = 4800Mbps$. This means that the output data rate is 2.5 times higher than the input data rate. Using this it was calculated that during the 128 cycles the *Cube DMA* uses to transfer all elements from one FIFO, approximately 52 new samples are fed into the FIFOs. Table 3 shows an estimation of the number of elements contained in the FIFOs. Cycle 0 is when all FIFOs are filled with 512 components and the *Cube DMA* starts transferring. After 128 cycles the *Cube DMA* has transferred 512 components from the first FIFO (*FIFO_1*), however 52 components has been inputted during this time. Then at cycle 256 the *Cube DMA* has transferred 512 components from the second FIFO (*FIFO_2*) and a new 52 components has been added. Repeating this operation until the 512 components from all FIFOs has been transferred, it can be seen that *FIFO_4* contained 668 elements at most in cycle 384. This means that the FIFOs should be at least larger than 668. The premade Xilinx FIFO that is used is restricted to a size that is the power of 2. This gives 1024 as the nearest size larger than 668. Even though this is a considerable amount over the requirement it is reasonable to have some margin as the *CubeDMA* has some cycles where it is not able to transfer data.

Cycle	0	128	256	384	512
FIFO_1	512	52	104	156	208
FIFO_2	512	564	104	156	208
FIFO_3	512	564	616	156	208
FIFO_4	512	564	616	668	208

Table 3: Elements contained in FIFOs.

6.2.4 Connecting the parts

As mentioned, it is not possible at this point to execute the Image Sensor due to incompatible mapping to the Zynq-7030 FPGA on the prototype and therefore the LVDS receiver will not be executed on hardware. However, if this was to be done the Figure 39 shows a possible solution to how this circuit should be connected together. Starting on the left side, it can be seen that the LVDS control (*ctrl_p_n*), data (*data_p_n*) and clock (*clk_p_n*) channels are made external. These will have to be mapped on the corresponding pins on the FPGA. Then we have the *Control_Interface_0* block. This is a simple block to map the *AXI GPIO* signals to the different signals in the design, giving the possibility to control the hardware from software. The VHDL code for this block is found in the Appendix E.4. The next block is the *processing_system7_0*, this is the proscessing system of the Zynq. It has two clock outputs, one at 100MHz for the *Cube DMA* side of the design and one for the reference clock used by the *IDELAYCTRL* inside the *Deserializer*. Further the block has some AXI interfaces and some interrupt inputs. The *lvds_deserializer* block is the Deserializer which is controlled by the *Control Interface*. Next the *pixel_alignment_0* block which is the *Pixel Order Alignment Module* are connected between the output of the *Deserializer* and the input of the *Binning Module*. Following this module is the *Cube DMA* which will store this result in memory. If binning was not desired, this module should be removed and the *Pixel Alignment Moduel* should be connected directly to the *Cube DMA*. The *axi_gpio_0* is a AXI GPIO block which can be controlled from software and interface the design through the *Control Interface*. *xlconcat_0* is a block to concatenate the interrupts from the *Cube DMA* to the interrupt input at the Zynq processing system. The *rst_ps7_0_100M* and *ps7_0_axi_periph* is auto generated by the design tool to fulfil the functionality. On the right side it can be seen that *fram_req* and *FCLK_CLK2_0* is external signals. The *frame_req_0* would be mapped to the *FRAME_REQ* signal of the image sensor and the *FCLK_CLK2_0* would be connected to the master clock input of the image sensor.

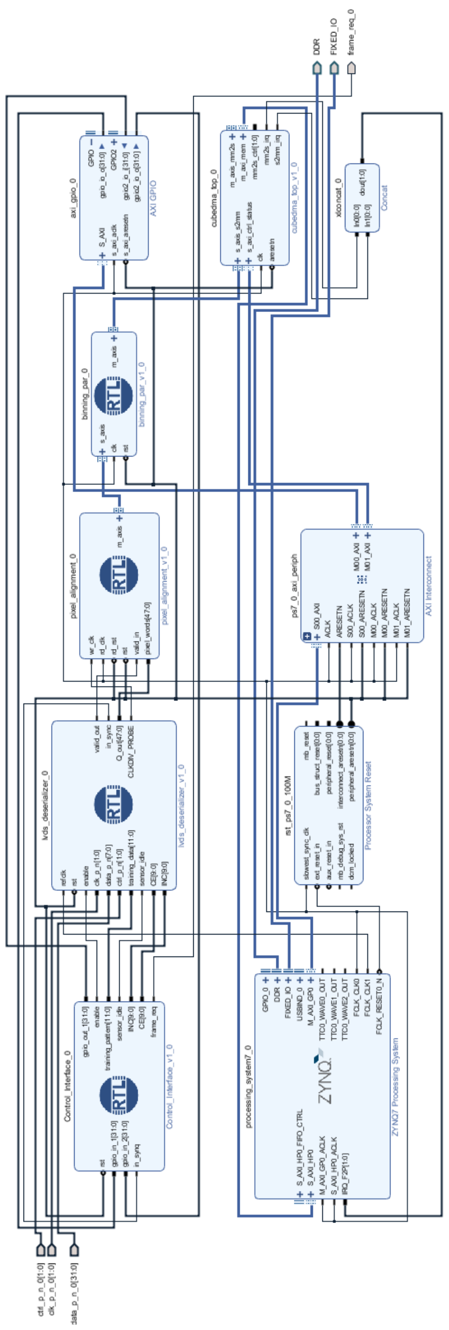


Figure 39: Block diagram of all parts connected together

6.3 Binning

An implementation for binning the data from the LVDS receiver with a binning factor of 4, 8 and 16 was implemented. Figure 40 shows the logic behind it. Because the *Pixel Order Alignment Module* is designed for the *Cube DMA* and outputs 4 samples each cycle the *Binning Module* has to adapt to this. These 4 samples are first added together through the 3 adders. Then a register samples this value. The last adder and register accumulates the result according to the binning factor. For a binning factor of 4 the accumulator is skipped and the register in the middle is directly connected to the output. For a binning factor of 8 and 16 the module accumulates 2 and 4 values respectively. At the output the result is shifted to divide the result corresponding to the binning factor that is used. For 4, 8 and 16 the required shifts are 2,3 and 4 respectively. Lastly the *counter* in the figure is a control block consisting of a counter that decides when the output is valid for the receiving side to know when to sample the value. The *Cube DMA* only has to transfer 1 sample for each transfer because this module merges multiple samples into 1. The *Binning Module* will contain a AXI-stream interface on both input and output to fit between the *Pixel Order Alignment Module* and the *Cube DMA*. This interface is not shown in the figure. The *Binning Factor* has to be 4,8 or 16 for this module to function.

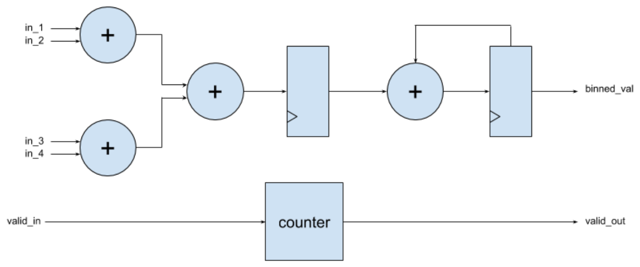


Figure 40: Binning module.

6.4 Extended Multiplicative Signal Correction

One of the tasks given in this thesis was to implement the Extended Multiplicative Signal Correction (EMSC), described in the background section, for the Zynq-7000 series. A software implementation was made in C++, this code was profiled to see what parts would benefit from being accelerated in hardware. Using these results, a hardware design to accelerate this operation was implemented. This design was then further developed to utilise even more parallelism hoping to accelerate the operation even more.

6.4.1 MATLAB script

A MATLAB function of the EMSC from the paper *Light Scattering and Light Absorbance Separated by Extended Multiplicative Signal Correction. Application* *to Near-Infrared Transmission Analysis of Powder Mixtures* [8] was used as a starting point in the implementation process. The function is shown in figure 42.

The MATLAB function takes two input arguments, *raw* and *ref_spectra*. Argument *raw* is a two dimensional representation of the cube organised such that each pixel is contained in each row. For example, a 500x500x52 cube, would be represented as a 250000x52 matrix. This transformation is illustrated in figure 41. Arguments *m* and *n* are the pixels and *components* the spectral components in each pixel.

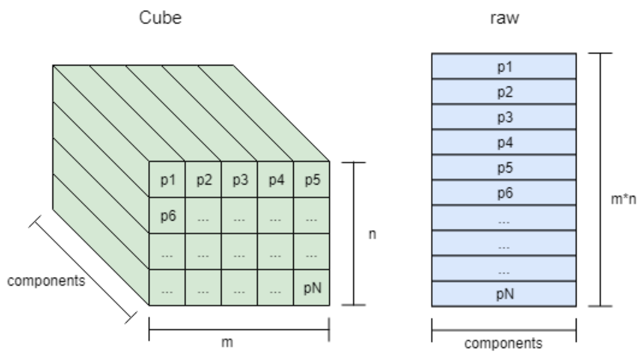


Figure 41: Cube transformed to raw.

The *ref_spectra* is the reference spectra used in the calculation. The reference spectra consists of spectral signatures defined by the user. The function starts by calculating **K** by subtracting the first row from every row in the reference spectra. This is an operation done by the user based on the application of this function. For the implementation that is to be designed, this operation will be kept outside the function, such that **K** is directly assigned to *ref_spectra*. Further on, the mean of the *ref_spectra* is calculated and stored in mean vector **m**. Lines 6-9 consist of declaring variables necessary to construct the **M** matrix described in the background section. Arguments *nVars* and *nObs* represent the number of wavelengths for each pixel and the total number of pixels respectively. Argument *wlens* is the wavelengths represented as a row vector of N linearly distributed numbers between 0 and 1, where N is the same as the number of wavelengths (*nVars*). *wlensSQ* is *wlens* squared. On line 11 the **M** matrix is constructed and one can proceed calculating the corrected spectra. In line 19 the **p** vector containing the coefficients needed for the EMSC correction is estimated using the procedure described in the background section equation 6. Then finally in line 20 the corrected spectra is calculated by subtracting the coefficients estimated from the *raw* input. Table 4 shows a description of the different variables and abbreviations that are used.

```
function [ corrected ] = emsc(raw, ref_spectra)
%EMSC_JF Summary of this function goes here
%   Detailed explanation goes here
K = bsxfun(@minus,ref_spectra(2:end,:),ref_spectra(1,:));
m = mean(ref_spectra);
nVars = size(raw, 2);
nObs = size(raw, 1);
wlens = linspace(0,1,nVars);
wlensSQ = wlens.^2;

M = [ones(1, nVars);
    m;
    K;
    wlens;
    wlensSQ];

corrected = raw;
for idx = 1:nObs
    p = raw(idx,:)*M'*pinv(M*M');
    corrected(idx,:) = (raw(idx,:) - p(1) - p(4)*wlens - p(5)*wlensSQ ) / p(2);
end
end
```

Figure 42: MATLAB function of EMSC

Element	Description
raw	Two dimensional representation of cube data
ref_spectra	Reference spectra
m	Mean of reference spectra
K	Reference spectra
nVars	Number of components in each pixel
nObs	Number of pixels. ($n \times m$ in figure 41
wlens	Wavenumber, 0 to 1 in intervals of 1/nVars
corrected	Corrected spectra containing the raw data with coefficients subtracted.
G	Containts M'*pinv(M*M')
P/p	Dot product of each pixel and G. (raw*M'pinv(M*M'))
M	Matrix constructed of ones, m, K, wlens and wlensSQ.
test	Constant zero

Table 4: Description of variables and abbreviations

6.4.2 Software implementation in C++

A software implementation of the EMSC is made in C++. The complete code can be found in appendix A but the main parts showing the EMSC calculation will be presented here in smaller parts with an explanation. This implementation utilizes an external library called *Eigen* for performing different matrix operations, including a built-in function for calculating the pseudo-inverse of a matrix. Also another approach using the Eigen library to calculate the inverse matrix computation was implemented for comparisons. The result of the pseudo-inverse and inverse matrix computation is identical if the matrix is invertible. This was also confirmed using MATLAB. The reason for using the inverse is because the pseudo inverse utilises complete orthogonal decomposition which is a powerful but demanding calculation. Therefore, the inverse could be a more effective solution for this application where the pseudo inverse is not needed. Figure 43 shows the function prototype for the function EMSC.

```
void EMSC(double** raw,
          double** ref_spectra,
          double* mean_spectra,
          double** corrected,
          int nVars, int nObs,
          int refOrder)
```

Figure 43: Function prototype

The function takes 7 arguments. Arguments *raw*, *ref_spectra* and *corrected* can be recognised from the MATLAB script and is inputted as double pointers. Argument *raw* is the cube data in two dimensional representation. *ref_spectra* is the reference spectra and *corrected* is where the corrected spectra will be stored. *mean_spectra* is the mean of the reference spectra and is passed as an input instead of calculated inside the function as in the MATLAB script. This gives the user the ability to manipulate this data outside of the function if desired. Arguments *nVars* and *nObs* represent the same as in the MATLAB script, the number of wavelengths and total number of pixels. Arguments *refOrder* is the order of the reference matrix. Dependent on how many spectral signatures the user defines in the reference spectra, the size of the *ref_spectra* will vary and *refOrder* is used to represent this.

Figure 44 shows some of the declarations made inside the function to see how the different matrices used are declared. *MatrixXF* is a type declaration from the library *Eigen* and creates a matrix of floats with size according to the parameters inside the parenthesis, to use *doubles* change the *Xf* to *Xd*. **M** is the **M** matrix and **G** is a matrix for storing the pseudo-inverse matrix. **p** is the vector containing the coefficients for calculating the EMSC corrected spectra. *initialize* is a function for allocating memory to double pointers and can be seen in Appendix A.

```
//---------------------DECLARATIONS---------------------
    MatrixXf M(refOrder + 4, nVars);
    double ** G = initialize(nVars, refOrder+4);
    double* p = (double*)malloc((refOrder + 4) * sizeof(double));
    double num = 0;
    //-----------------------------------------------------
```

Figure 44: Declarations.

In figure 45 the for-loop constructs the **M** matrix. As one can see, the indexes of *Eigen* matrices are different than pointers as you use parenthesis instead of brackets. As the comments in the figure describes, ones are added in the first row. The second row contains *nVars* of equally spaced numbers between 0 and 1 representing the wavelengths. The third row is the same as the second row squared. Then the reference spectra is added from row 3 to row $(3 + refOrder)$. In the last row the mean of the reference spectra is added and this forms the **M** matrix.

```
for (int i = 0; i < nVars; i++) {
        //Add 1 in first row
        M(0,i) = 1;

        //Add linspace and linspace squared
        M(1,i) = num;
        M(2,i) = pow(num, 2);
        num += (1.0 / (nVars - 1));

        //Add the Reference spectra
        for (int y = 0; y < refOrder; y++) {
            M(y + 3,i) = ref_spectra[y][i];
        }

        //Add mean in last row
        M(refOrder+3,i) = mean_spectra[i];
    }
```

Figure 45: Constructing the **M** matrix.

In Figure 46 the pseudo-inverse is calculated and stored in *p_inv*. First the transpose of $\mathbf{M} \times \mathbf{M}$ is calculated and stored in **M_M**. Then the transpose of **M** multiplied with the pseudo-inverse of **M_M** is calculated and stored in **p_inv**. It works by calculating the complete orthogonal decomposition of **M** and uses this to calculate the pseudo-inverse. It was discovered that read and write operations of the *Eigen* matrix type was slower than for two dimensional pointers and increased the execution time by 2-3 times because of many read operations in the *Calculating Corrected Spectra* part of the code. It was, therefore, beneficial to store the result from **p_inv** in **G**, a two dimensional pointer as shown in the code. To calculate the inverse instead of the pseudo-inverse one simple change line 2 to *MatrixXd p_inv = M.transpose() * M_M.inverse()*.

```
1  MatrixXd M_M = M*M.transpose();
2  MatrixXd p_inv = M.transpose() * M_M.completeOrthogonalDecomposition().pseudoInverse();
3  for(int i = 0; i<nVars; i++){
4      for(int y = 0; y<refOrder+4; y++){
5          G[i][y] =(double) p_inv(i,y);
6      }
7  }
```

Figure 46: Calculating the pseudo-inverse.

Lastly the result from the pseudo-inverse may be used to calculate the corrected spectra. Figure 47 shows how this is done. First a for-loop calculates the **p** which then is used to calculate the corrected spectra in a similar fashion as the MATLAB script.

A tutorial of how to use the EMSC implementation on the Zedboard is added in Appendix B.

```
sum = 0;
for (int idx = 0; idx < nObs; idx++) {
    for (int i = 0; i < refOrder + 4; i++) {
        for (int y = 0; y <nVars; y++) {
            sum += raw[idx][y] * G(y,i);
        }
        p[i] = sum;

        sum = 0;
    }
    for (int t = 0; t < nVars; t++) {
        corrected[idx][t] = (raw[idx][t] - p[0] - p[1] * M(1,t) - p[2] * M(2,t)) / p[refOrder + 3];
    }
}
```

Figure 47: Calculate the corrected spectra.

6.4.2.1 Profiling the C++ Implementation As the implementation was tested and confirmed working, profiling of the code was executed. This was done by adding a *AXI Timer* to the hardware design. This is a module which outputs the number of clock cycles that have elapsed between two points in time. The timer was used to measure the execution time of the whole code as a reference. The different parts of the code was then timed and compared to this reference time. This was meant to give a good indicator of what parts of the implementation would benefit the most being executed in hardware. The result from this profiling is shown in the result section. However, as the next section is based on this result it should be mentioned that the *Calculate corrected spectra* from last section was the best candidate to accelerate in hardware.

6.4.3 Hardware implementation

Based on the results collected by the profiling it is shown that the part of the algorithm calculating the corrected spectra shown in Figure 47 was the best candidate for implementation in hardware. This means that the software prior to this part in the code will execute as before, but when it reaches the *Calculate corrected spectra*, parts of this is executed in hardware before software fetches these results and proceeds. In other words, this will be a Hardware/Software co-design. The *Calculate Corrected Spectra* consists of two main steps, first calculating **p**, a dot product between the components in a pixel (*raw*) and the matrix G, containing the result from the $M' \times pinv(M \times M')$. This shown in equation 8.

$$\mathbf{p} = \mathbf{raw} \cdot \mathbf{G} \tag{8}$$

Then a step of arithmetic operations using **p** to get the corrected spectra shown in equation 9.

$$corrected = \frac{(raw - p(1) - p(4) \times wlens - p(5) \times wlensSQ)}{p(2)} \tag{9}$$

Because multiplications are slow the main focus of the accelerator will be in accelerating the dot product calculation in the first step, which consists of the same amount of multiplications as components across all pixels. Then the arithmetic step is executed in software with **p** received from hardware.

Figure 48 shows a brief overview of the architecture of the hardware part implemented for the EMSC accelerator. The signal names *raw*, *G* and *p* can be recognised from the equation 8 presented above. The modules *Block Ram*, *Dot Product Module* and *Output Module* will be briefly introduced here and explained in details separately later in this section. The *Block Ram Module* stores the values of *G*, which are calculated in software and written directly from the CPU to the block rams contained in this module. These values are further used calculating the dot product between raw and G (equation 8). This dot product calculation is done in the *Dot Product Module*. In the *Output Module* the results from the *Dot Product Module* are organised and synchronised for the output AXI-stream.

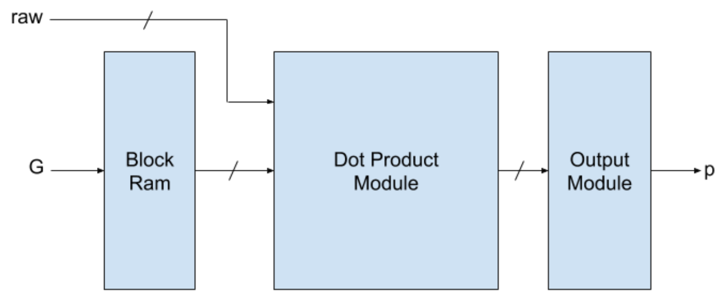


Figure 48: Overview of the implemented EMSC accelerator.

6.4.3.1 Block Ram Module As mentioned, a *Block Ram Module* was implemented. As can be seen from the MATLAB script (Figure 42) line 19, **p** is calculated from the dot product of all components in a pixel (*raw*) and **G**. To avoid having to stream this **G** repeatedly it was decided to store these values in block rams inside the accelerator. This means that the *Block Ram Module* has to be initialised with values before enabling the accelerator and calculating *p*. Figure 49 illustrates how this was implemented.

It consists of an *AXI Register Interface*, a *Block Ram Bank* and some control logic. The *Block Ram Bank* is simply a module consisting of block rams able to store **G**. It has the same amount of block rams as there is columns in **G** which again is decided by the dimensions of the reference spectra. The *AXI Register Interface* is a module containing registers that can be interfaced through software. This will be used to write the values of **G** to the block rams as well as to write control signals to the accelerator. This is done through the *AXI4-Lite* protocol as shown in the figure (49). The *AXI Register Interface* is seen by the processor as memory addresses, and is therefore interfaced by write/read operations to memory. For writing **G** this is combined with something called *keyhole* type burst meaning that the CPU writes repeatedly to one specific reg-

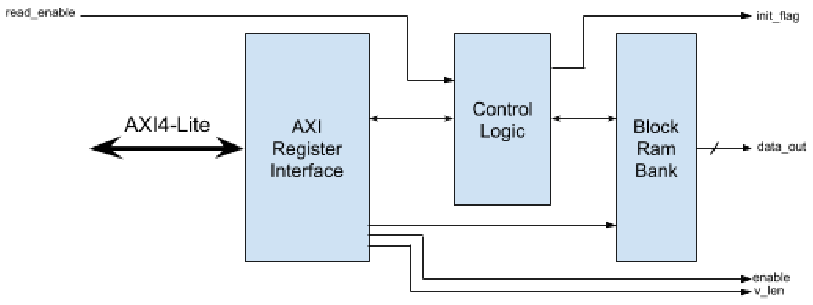


Figure 49: Block Ram Module Architecture

ister in the *AXI Register Interface* to store values in the block ram. This works because the *AXI Register Interface* has internal logic that can detect and communicate to the *Block Ram Bank* when there is a new valid input ready for storing. However, an important requirement for this to work is that the values from the processor may not arrive at a higher frequency than the operating frequency of the FPGA at 100MHz. This should however not become a problem as the processor runs up to 1 GHz, but will spend more than 10 cycles between each of the value to be stored is calculated and written to the *AXI Register Interface*. The input signal *read_enable* is used read the block ram. This data is outputted at the output signal *data_out*. At the rising edge of the clock, a high on the *read_enable* will output the next data stored in the block ram and when it reaches the end it will start over. This works because the data is stored in the same order as it is used. The output signals *enable* and *v_len* is control signals written by the processor to the control logic. The *enable* signal enables the accelerator and the *v_len* is the number of components in each pixel. The *init_flag* communicates to the rest of the accelerator if the block ram has been initialised with values.

Table 5 shows the register map for the block ram. The control register is at the base address of the block ram (0x00). *G size* is the number of elements each block ram in the *Block Ram Module* are storing. *G size* is the same value as the number of components in each pixel. *Start*, enables the accelerator if the *Block Ram Module* has been *Initalized*. Also, the Cube DMA should not start streaming values to the accelerator before this *Start* is set to *1*. *Init* is set high to program the values to store in the *Block Ram Module*. If the CPU writes to the *Input G* when *Init* is low, the value written will be ignored. On offset 0x04 from the base address of the *Block Ram Module* there is a *Input/Status* register. During *Init* is high, inputs are written to this register. When the *Block Ram Module* has completed initialization the *Initalized* bit will be set high. As there is no reason to be able to see the last **G** value written to the *Block Ram Module* this is overwritten by the *Initialized* bit. This also reuses the register, avoiding using an extra register for one status bit. At the offset (0x08) there is a *Count Register*. This is used to know how many pixels to process and when to signal the *Cube DMA* that the last pixel component has been sent. Ignoring this register is possible. But this requires to reset the *Cube DMA* after each run and is not recommended.

Field	**Description**	**Bits**
Control register (0x00)		
G size	Length of each pixel	11-0
Start	Enables the block ram module	12
Init	Sets the hardware in init state	13
Ref Order	Number of reference spectra	18:14
Input register/Status register (0x04)		
Input G	Writes to this register stores it in block ram	31:0
Initialized	status bit to show if block ram is intialized	0
Count register (0x08)		
length	Number of pixels to process	31-0

Table 5: Block Ram register map.

6.4.3.2 Dot Product Module The dot product module is the core of the accelerator and is where the calculation of **p** occurs. Figures 50-55 show the intended operation of this module.

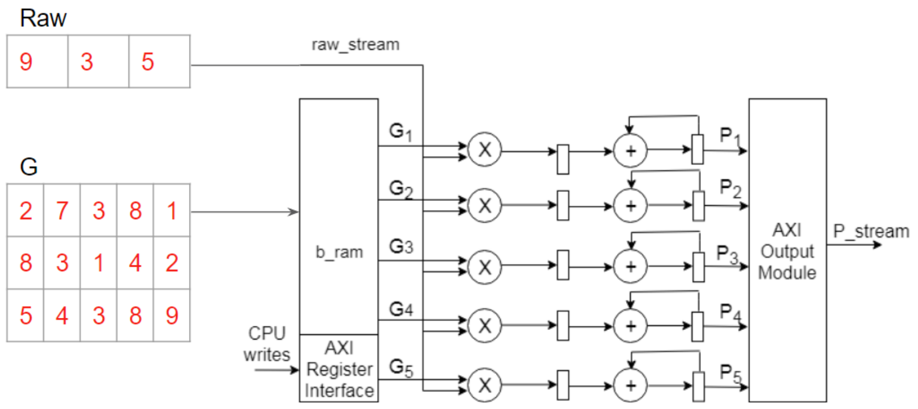


Figure 50: Dot product operation, initial step.

Figure 50 shows the initial point. A pixel with 3 components shown in the *Raw* table will be streamed through the *raw_stream*. The values in the **G** table illustrates the values that have been stored in the block ram during the initialisation phase prior to enabling the accelerator. Just to clarify, in the example the values in the **G** table are removed when used, this is however not happening inside the Block ram. Because the **G** values are used multiple times, they are stored in the Block ram until the user decides to overwrite them with new values.

In the first step (figure 51) the first pixel component (5) and the first row of **G** is outputted to the multipliers. The product of these multiplications are then stored in a register.

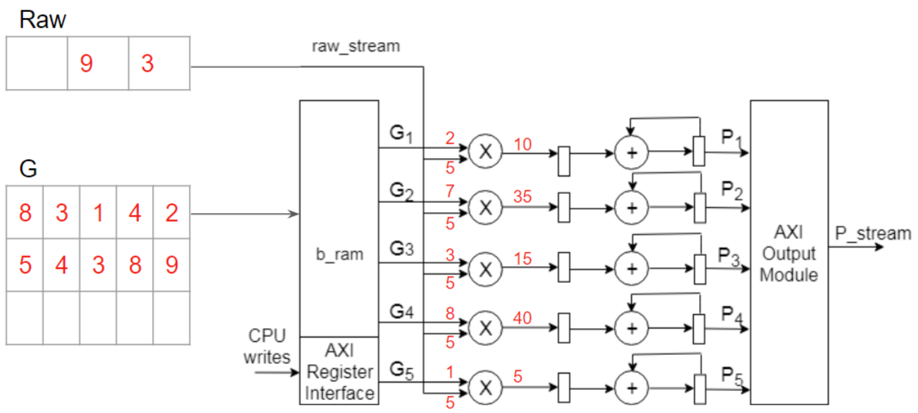


Figure 51: Dot product operation, step 1.

In the second step (figure 52 the first product is outputted to the accumulator which is initialised to 0. At the same time as the next pixel component and row from **G** is multiplied.

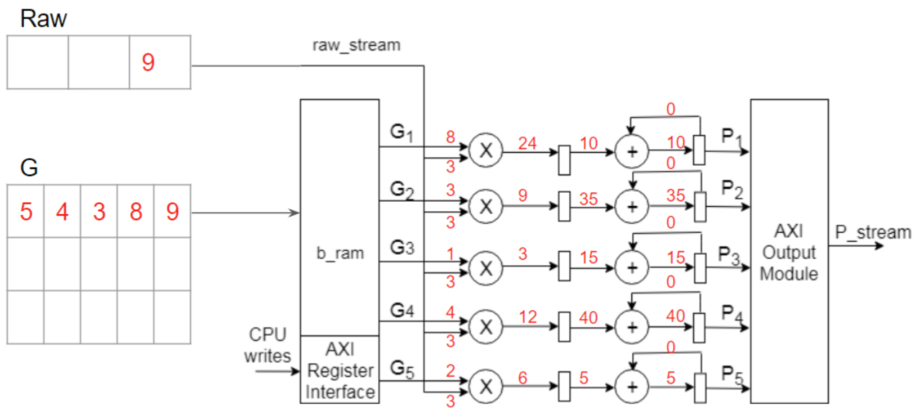


Figure 52: Dot product operation, step 2.

Figure 53 shows the third step where the first and second product is added together in the accumulator while the last multiplication is executed in the first stage.

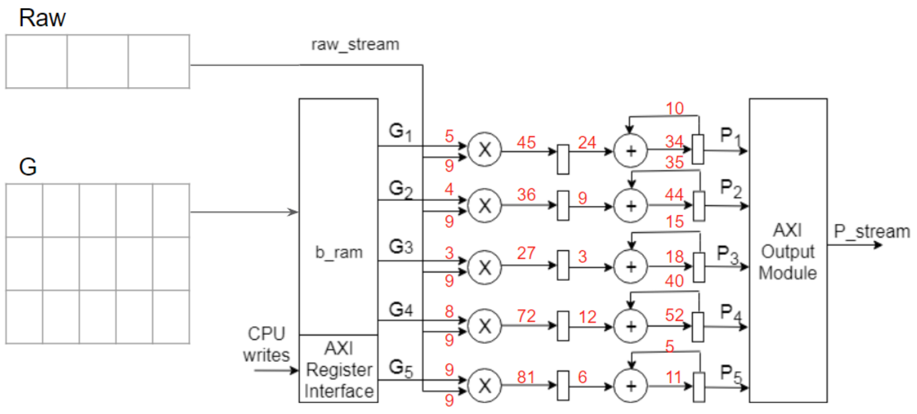


Figure 53: Dot product operation, step 3.

The last product is added to the sum of the two first products in the fourth step (figure 54). This concludes the dot products and the circuit is now ready to output the results.

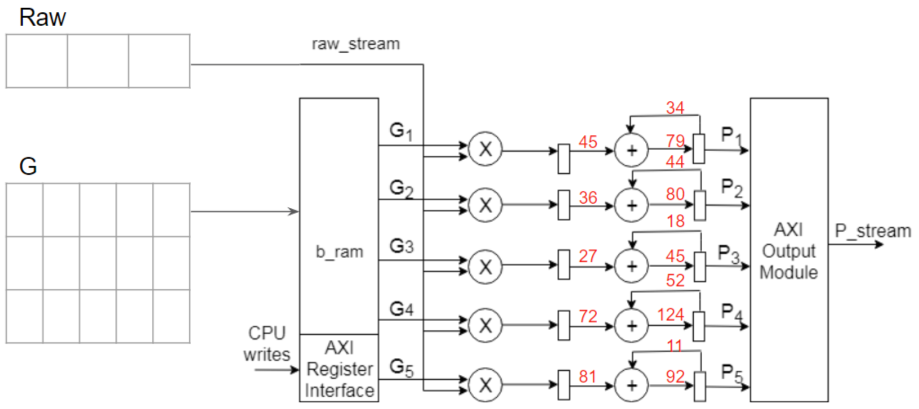


Figure 54: Dot product operation, step 4.

The values are streamed out of the *P_stream* in the last step of the dot product calculation (figure 55. These values are streamed out in a serial fashion, meaning that for this example it would require 5 clock cycles to output the 5 **p** values.

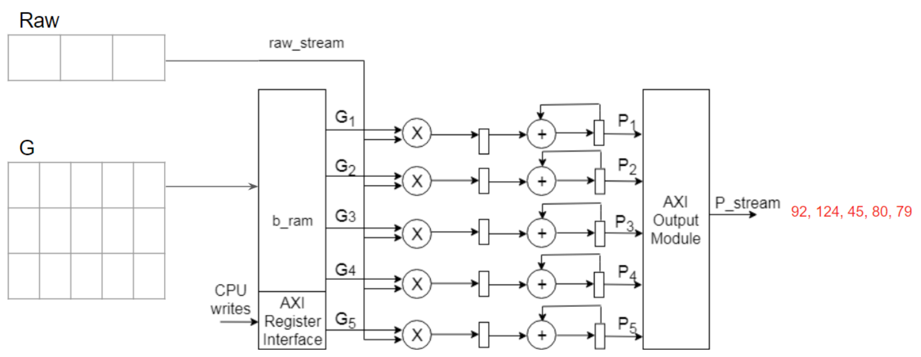


Figure 55: Dot product operation, last step

Even though this example is small compared to the real application which could contain hundreds of pixel components, it shows the principle of the *dot product module* and demonstrates how it works.

6.4.3.3 Output Module Figure 56 shows a simple block diagram of the *Output Module*. Each time a dot product is calculated, a signal called *p_rdy* will trigger the *Output module*. When such a trigger occurs, all elements of the **p** will be sampled into a register inside the Output module. These values will be shifted out on the *axis_data* signal at the same time as the *Control Logic* block organises the AXI control signals, *axis_ready*, *axis_valid* and *axis_last*. Additional to outputting the results from the dot product module, the output module is responsible to stall the pipeline if the Cube DMA that is connected to the AXI output is not able to receive data. This is because if the pipeline runs while the DMA is stalling one may risk that the **p** values stored in the register is overwritten by new results before they are fetched by the DMA and stored in memory.

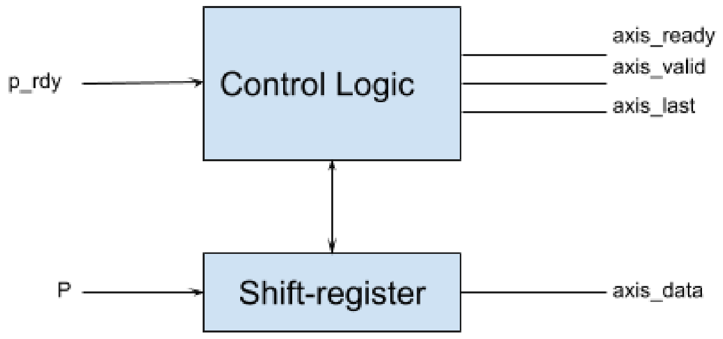


Figure 56: Output module overview

6.4.3.4 Sequential EMSC HW design The building blocks described above was connected together as shown in figure 48 forming what from now on will be called the *Sequential EMSC HW design.* This was found a suiting name as it only fetches and processes one pixel component sequentially. This design was further developed into a new design being able to fetch and process multiple pixel components in parallel. This design is therefore named the *Parallel EMSC HW design* and will be presented now.

6.4.3.5 Parallel EMSC HW design The parallel version of the EMSC hardware part uses the same building blocks as described above but has some additional logic to control and synchronise the dataflow. Figure 57 shows an overview of this version.

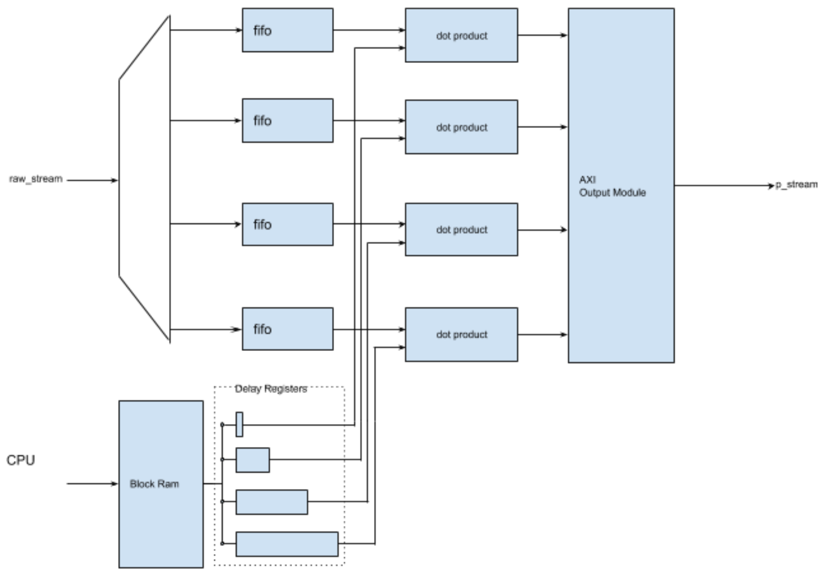


Figure 57: Overview of parallel version

It was desired to implement a solution which utilises the building blocks that already are implemented above. However, because the pixels from the cube will be stored in memory after each other as shown in figure 58, some additional logic had to be used to be able to use the Dot Product module for this.

The design functionality will be explained by using an example. The test data that was available consisted of a 500x500x52 cube. This means that there was 250000 pixels with 52 spectral components in the cube. Each pixel component is 16 bit making the *Cube DMA* able to transfer 4 pixel components each cycle. When the accelerator starts, each of the FIFOs are filled with one whole pixel. Starting with the upper FIFO, it will use 13 cycles to fill all the 52 components. When it is filled it starts outputting data to the corresponding *Dot Product Module* at the same time as the next FIFO is getting filled. This way,

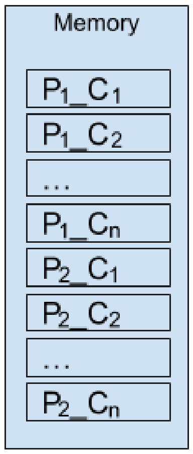


Figure 58: Pixels stored in memory. **p** is pixel number and C is component number.

all *Dot Product Modules* will start with an offset of 13 cycles. The *Block ram* is duplicated from the other version. The only change is some delay registers that synchronises the **G** with the output of the FIFOs. Because one cycle is needed to read the FIFO, there is one delay register from the *Block Ram* to the first *Dot Product Module*, 14 cycles delay from the second one, 27 cycles delay from the third one and so on. When the *Dot Product Modules* has calculated a result, it passes this to the *AXI Output Module* which streams the result on the output. The dimension of the *ref_spectra* decides how many numbers each *Dot Product Module* produces. An important requirement which can be illustrated for this specific example, is that if the number of elements in **p** is larger than 13, the whole module will have to be stalled to be able to output all of the results. This happens because the module will produces more than 13 values to output each 13th cycle. Each of these 13 values need one cycle to be outputted to AXI-stream. However, as the application will probably contain more components in each pixel, this also increases the number of cycles between the outputs.

This example is based on some specific test data with given dimensions. This means that the circuit might need additional logic if other data is used, however the core will remain. If the number of spectral components had been 53. It would take 13 cycles to transfer 52 of them. Then the next cycle, there would be 1 component from the last pixel and three components from the next pixel. These components would need to be separated.

As this design execute 4 calculations in parallel it would theoretically give a speedup of 4 compared to the sequential version.

6.4.3.6 Design Choices Different choices had to be made concerning the hardware implementation. These choices includes bit widths, rounding precision, using floating point or integer and more. This part was found to be complicated and many hours was spent in considering all the factors that has to be taken into account.

The first important decision that had to be made was to find a reasonable bit width of **G**, the values that are going to be stored in the *Block Ram Module* and used in the multiplication in the *Dot Product Module*. A good solution for implementing the Dot Product Module was to utilise Digital Signal Processing (DSP) slices. These are in simple terms, optimised building blocks that performs different hardware implemented algorithms. Figure 59 taken from Xilinx datasheet *7 Series DSP48E1 Slice* [21] shows an overview of the *DSP48E1*.

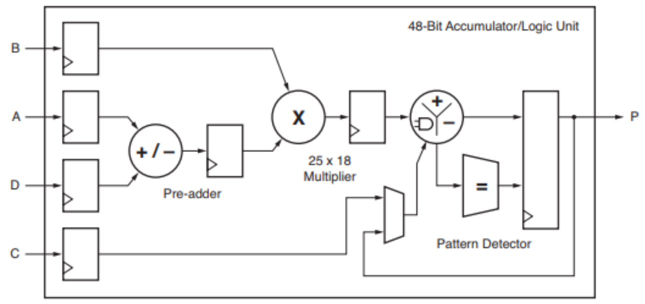


Figure 59: Overview of a DSP48E1. [21]

It can be seen that it contains with other things, a 25x18 bit multiplier and a 48-Bit accumulator which is the functionality that is needed for the *Dot Product Module*. A multiplier to multiply each element with same index of both vectors and an accumulator to sum up these multiplications. Additional to having the required functionality, the DSP slice is able to execute the multiplication in one clock cycle, which is efficient. Another reason for using DSP is that it is efficient in terms of not spending time on implementing a multiplier from scratch. As the multiplier is made for 25x18 bits, 25 bits was chosen as a starting point bit width for **G** and the 18 bits is sufficient for the pixel components, which may be outputted as 10 and 12 bit from the image sensor. As will be shown, the 25 bit width might not be sufficient for the desired precision.

Another decision to make was to implement the hardware to directly handle floating point numbers or integers. Handling floating point numbers in hardware is complicated but as Xilinx provide a floating point core in their IP catalog [19] which provides all the required operations this could be possible. However, this core has a high cost in resources and requires multiple clock cycles to perform single multiplications and additions. Also, as will be seen below, using integers and carefully choosing the bit width, gives a high precision and the benefit from using the floating point core disappears. Therefore a solution using integers was chosen. When **G** is calculated, it consists of numbers with decimals. These numbers will then be scaled with a factor that is chosen by the user and rounded to the closest integer. Choosing this factor and how it impacts the result is what is going to be presented now.

Finding a reasonable factor was calculated using MATLAB and mimicking the behaviour from the implemented SW/HW design to calculate an estimate for the error.This behaviour is as following. First the software calculates **G**, as this elements are transferred to the block ram they are multiplied by a factor and rounded down to the nearest integer. Then the Hardware implementation is executed and the software may fetch the calculated **p** from memory. As this **p** is read, the value is now scaled down by dividing by the same factor. The larger the factor used the smaller the error due to the rounding will be. Table 6 shows the result obtained from MATLAB. The results show that the factor should be larger than 2^{19} because at this point the error is decreasing by increasing the number of bits. It can be seen that a bit width of 49 is required to achieve a error below 1. The *Bits Required* are calculated by multiplying the data input with the factor and then estimating how many bits are required to represent the maximum and minimum values.

Multiplication factor	Bits Required	Largest Error
2^{17}	23	1.5775e+04
2^{18}	24	2.9872e+04
2^{19}	25	1.5549e+03
2^{20}	26	663
2^{21}	27	259
2^{22}	28	144.69
2^{43}	49	0.8035

Table 6: Description of different used letters and abbreviations

When a set of EMSC coefficients (**p**) has been calculated by the accelerator and is ready to be outputted there are different ways of doing this. The chosen implementation was to use the Cube DMA to fetch results from the design and store this in memory. To fully utilise parallelism it was desired to make the CPU start processing data stored in the memory before the accelerator has completed execution. The solution that was implemented for this was that the accelerator interrupts the CPU when a result has been transferred to memory. A drawback by this solution is that it might lead to huge amount of interrupts, resulting in the CPU being blocked in such a degree that the total execution time either gets a small speedup, no speed up at all or even are slower than the software version. An alternative to this implementation was to use the same *keyhole burst transfer* used to store **G** in the block ram. This resulting in the CPU directly fetching results from the accelerator through an *AXI Register Interface*. However as the *Cube DMA* has two channels, one for fetching from memory and one for storing data in memory which runs in parallel it was desired to use utilise both these capabilities. Also the accelerator stalls if the receiving part is not able to fetch data immediately, this would put a higher stress on the CPU in the role of the receiving part.

6.4.4 Combining Hardware and Software

Figure 60 shows the architecture of the hardware/software codesign for the EMSC algorithm. The algorithm starts on the left side with the green inputs to the *Construct Matrix M* block, where the **M** matrix is constructed. Then **G** is calculated from **M** and the values are loaded into the block ram in the hardware part. Then the *Cube DMA* is initialised to fetch pixel data from memory and stream it through the accelerator. The results are then fetched by the *Cube DMA* and stored a different place in memory where the software may take over and calculate the corrected spectra using these values. This overview holds for both the sequential implementation and the parallel EMSC version, which will be presented later in the section. The only difference would be the number of bits transferred from memory to the *Calculate P* block.

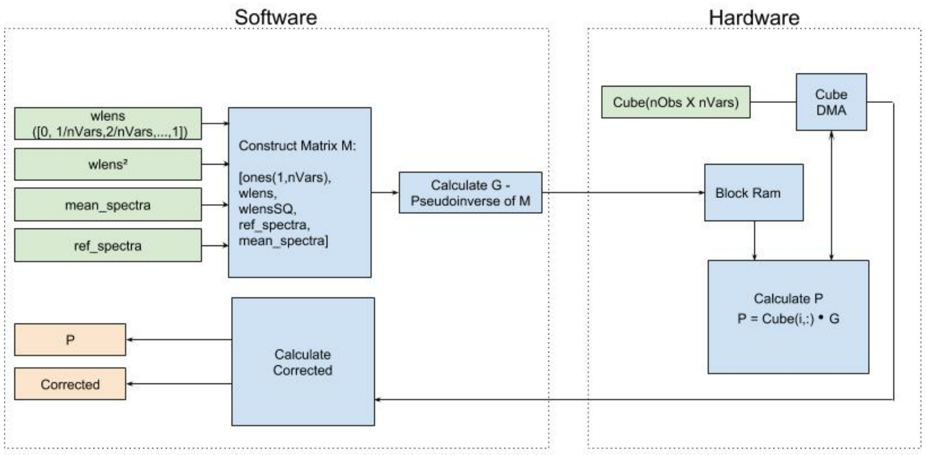


Figure 60: Overview of the first EMSC implementation

6.4.4.1 Software implementation The software implemented for the SW/HW implementation has many similarities with the pure software implementation that was implemented in the earlier stages. The entire code is available in Appendix G.1 but the most important parts will be presented in this section. Figure 61 shows the prototype of the EMSC function. It can be seen that it is identical to the pure software version except that the raw input is removed. Instead of passing this as a input from main it is read directly from memory inside the function.

Because the construction of the **M** Matrix is similar as in figure 45 from the software implmentation this will not be repeated here. Figure 62 shows the initializing of the *Block Ram Module*. It can be seen on line 4 and 6 that pointers to the control and input/status register are declared. In line 9 the control register are set to 0x2034 which corresponds to setting *G size* = 52 and asserting the *Init* bit. Then the pseudo-inverse is calculated on line 13. To

```
void EMSC(double ** ref_spectra,
          double * mean_spectra,
          double ** corrected,
          int nVars, int nObs,
          int refOrder)
```

Figure 61: Function Prototype.

instead perform a inverse line 13 is switched with the commented line 14. The for-loops at lines 17-21 multiplies the values in **G** by a factor called *multiplier*, rounds this result using *floor* and writes this directly in to the *Block Ram Module*. In line 23 the *Init* is set to '0' and line 25 enables the accelerator.

```
1       //Initiate Block Ram
2       //---------------------------------------------
3       //Create pointer to Block Ram base address
4       u32 * init = (u32*)0x43c10000;
5       //Creates a pointer to the address to write G
6       u32 * in_G = (u32*)0x43c10004;
7
8       //G_size = 52, init = '1'.
9       *init = 0x2034;
10
11      //Execute pseudo-inverse of M
12      MatrixXd M_M = M*M.transpose();
13      MatrixXd p_inv = M.transpose() * M_M.completeOrthogonalDecomposition().pseudoInverse();
14      //MatrixXd p_inv = M.transpose() * M_M.inverse();
15      xil_printf("Pseudo-Inverse Completed!\n");
16
17      for(int y = 0; y<refOrder+4; y++){
18          for(int i = 0; i<nVars; i++){
19              *in_G  =(int) floor(p_inv(i,y)* multiplier);
20          }
21      }
22      //init set to '0', keeps G_size value.
23      *init = 0x34;
24      //enable set to '1', keeps G_size valueTh.
25      *init = 0x1034;
26      //---------------------------------------------
```

Figure 62: Initialising the Block Ram Module.

The next part is to initalize the Cube DMA. The code is shown in figure 63. Line 3 and 4 creates pointers to the Cube DMA mm2s (memory map to stream) channel and s2mm (stream to memory map) channel control registers. Line 7 makes sure that the s2mm channel is not enabled. The memory address for storing the results from the accelerator is written to the corresponding register in line 8. Line 9 sets the *Completion IRQ enable* bit and the *Start* bit in the s2mm control register. The next step is to program the mm2s channel. First, line 12 makes sure the channel is disabled. The memory address to fetch the cube data is written to the corresponding register in line 13. The DMA wants to know the dimensions of the data to transfer, this is programmed in lines 17-18 for the large cube (500x500x52). line 17-18 is switched with lines 15-16 for the small cube (100x100x52). Then the *Completion IRQ enable* bit and enable bit for mm2s is set in line 20. At this point the Cube DMA will start transferring cube data to the accelerator. Line 21 is a while loop that will block the program until the flags *s2mm_complete* and *mm2s_complete* are assigned. These flags are assigned inside an interrupt handler executed when the Cube DMA triggers interrupts for completion on both channels.

```
//Initiate and enable Cube DMA
//---------------------------------------------
u32* mm2s = (u32*)0x43c00000;
u32* s2mm = (u32*)0x43c00020;

// Program S2MM DMA
s2mm[0] = 0x0;
s2mm[2] = 0x0F0BDBF0;
s2mm[0] = (1 << 5) | 1;

// Program MM2S DMA
mm2s[0] = 0;
mm2s[2] = 0x100010E0;

//mm2s[3] = 0x1001001; //Small cube
//mm2s[5] = 520000; //Small cube
mm2s[3] = 0x341F41F4; //Large cube
mm2s[5] = 0x6590;//Large cube

mm2s[0] = (1 << 5) | 1;
while (!s2mm_complete || !mm2s_complete);
//---------------------------------------------
```

Figure 63: Initialising the Cube DMA.

When both Cube DMA completion flags are triggered the result from the accelerator has been saved in memory and the software may start calculating the *corrected spectra*. Figure 64 *version 1* show how this was done. First two pointers are declared, one to the location of **p** and another to the location of cube data (raw). The while loop on lines 10-18 are making sure that the calculations are executed until all pixels has been processed. The *for* loop starting at line 11 is fetching all components in one row of the **p** and dividing this by the multiplication factor that was used when values was inputted to the accelerator. The second *for* loop at line 14, fetches a pixel component and calculates the corrected spectra in the same way as the MATLAB script.

```
1       //Calculate the corrected spectra
2       //----------------------------------------
3       int64_t * P_ptr = (int64_t*)0x0F0BDBF0;
4       u16 * raw_ptr = (u16*)0x100010E0;
5       double p_st[8];
6       u16 pixel_component;
7       int counter = 0;
8       //----------------------------------------
9       //Version 1
10      while(counter < nObs){
11              for(int i = 0; i < 8; i++){
12                  p_st[i] = P_ptr[i+counter*8]/multiplier;
13              }
14              for(int cols = 0; cols < nVars; cols++){
15                  pixel_component = raw_ptr[counter*52+cols];
16                  corrected[counter][cols] = (pixel_component - (p_st[0] + p_st[1]*corr_M[0][cols] + p_st[2]*
                            corr_M[1][cols]))/p_st[refOrder + 3];
17              }
18              counter++;}
19      //-----------------------------------------
20      //Version 2
21          while(counter < nObs){
22          if((int_counter - counter > 10) || (nObs - int_counter < 10)){
23              for(int i = 0; i < 8; i++){
24                  p_st[i] = P_ptr[i+counter*8]/multiplier;
25              }
26              for(int cols = 0; cols < nVars; cols++){
27                  ah = raw_ptr[counter*52+cols];
28                  corrected[counter][cols] = (pixel_component - (p_st[0] + p_st[1]*corr_M[0][cols] + p_st[2]*
                            corr_M[1][cols]))/p_st[refOrder + 3];
29              }
30              counter++;}}
```

Figure 64: Calculate corrected spectra.

Because this method blocks the processor while the accelerator is running a non blocking method was implemented. Figure 64 *version 2* shows the code for this implementation. In addition to using this code the line 21 in figure 63 was removed so the processor does not wait for the DMA. Instead the accelerator has it own interrupt that triggers each time a new row of **p** is calculated. Using this the processor can start process the **p**'s that are stored in memory before the accelerator has completed. However, it is important that the processor does not access memory where it not yet has been stored any values of **p**, because this would corrupt the results. Also, when the accelerator triggers the interrupt that a result is ready, the Cube DMA will still use some clock cycles before this result is stored in memory so this has to be handled. In the figure (64 at line 22 this is handled. This *if* makes sure that *int_counter* is larger than *counter*. *int_counter* counts how many results has been reported ready to process and *counter* counts how many results have been processed. In the figure, 10 is used as an example, but this value could be changed if the result seems to be corrupted. Also another condition is required to make sure that the last 10 results also get processed. This implementation as mentioned before will trigger the same amount of interrupts as there is pixels, this might give a performance reduction compared to the blocking version.

Memory mapping Table 7 shows the memory mapping that was used for the difference elements. The DDR memory on the Zynq is restricted to 512 MB so if very large cubes are to be used this could exceed this. A solution could be to either utilise a SD card directly or to process sub-cubes of the large cube multiple times.

Memory Address Range	Data
0x00100000-0x0C800000	Software Memory
0x0C800000-0x0F0BDBF0	Free
0x0F0BDBF0-0x10000000	P
0x10000000-0x10001000	Reference spectra
0x10001000-0x100010E0	Mean
0x100010E0-0x19CD1534	Raw
0x19CD1534-0x1FFFFFFF	Corrected Spectra

Table 7: Memory Mapping EMSC software

6.4.4.2 Testing the implementations A testing procedure was used to collect results from the different designs that was implemented. As mentioned above, profiling was executed to measure the execution times of the different parts of the design. Additionally to this, MATLAB was used to analyse the results to measure the precision.

7 Results

7.1 Image sensor pipeline

As mentioned the clock input to the Image Sensor was mapped to a pin from the HP bank of the Zynq. This is not able to provide the necessary voltage level therefore the image sensor has not yet been able to test. The result in this section will therefore be based on the information gathered from the design tool Vivado. This includes simulations showing the correct behaviour and utilisation reports from synthesis. The source files and testbenches for all of this modules can be found in the Appendix.

7.1.1 LVDS Deserialiser

The results from simulation and synthesis of the design presented in section *LVDS Receiver with Xilinx Primitives* is presented in this section.

Simulation Figure 65 shows a waveform of the initial phase of the receiver when training is done. The training pattern that was applied on all channels was "1010 1101 0010" which is 2770 in decimal representation. It can be seen that the receiver captures the correct bits "1101 0010 1010" (3370) but that these are not captured in the correct order. However, after 4 bitslip operations the correct training pattern is captured. It can be seen that the state machine does the synchronisation operation on every channel sequentially and that this requires the training pattern to be applied until all channels are synchronised which is indicated with the *in_sync* signal.

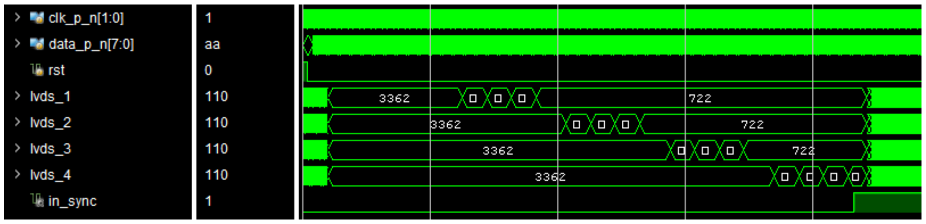


Figure 65: Waveform of training state of receiver

After the training phase has completed the testbench transfers 100 pixel components in incrementing order from 0, where 12 of these have been highlighted in the waveform shown in figure 66. It can be seen that all channels captures the correct values.

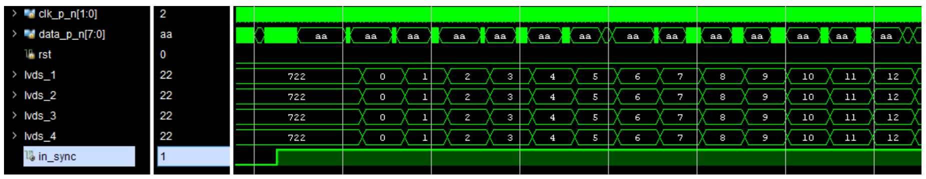


Figure 66: Waveform of transfer state of receiver

Synthesis The utilisation results from the synthesis are shown in table 8.

Pixel Bit Width	NUM_LVDS_PAIRS	LUTS	Registers	IDELAYE2	ISERDESE2
12	4	152	49	11	10
10	16	251	75	35	34

Table 8: Utilization report Deserialiser

7.1.2 Pixel Order Alignement Module

Simulation Figure 67 shows the waveform from simulating the *Pixel Order Alignement Module*. The module starts when *valid_in* are set high. The input channels are not visible in the waveform as a selection of important signals was included. *m_axis_tdata* shows the output to the *CubeDMA*. *m_axis_tvalid*, *m_axis_tready* and *m_axis_tlast* are the associated AXI-stream interface signals. *wr_cnt* is a counter that counts the number of elements written to the FIFOs. *data_out_1* to *data_out_4* are the outputs from the FIFOs that are distributed to the *m_axis_tdata* with a MUX. *rd_en* is the *read enable* signals to the FIFOs. Here it can be seen that all FIFOs are read once initially. *state* shows the current state of the state machine inside the *Control Logic* block showed in figure 37. The *fifo* signal is the signal deciding which FIFO the mux should pass to the *m_axis_tdata*. *component_cnt*, *row_cnt* and *frame_cnt* are counters that monitor the progress of the module. These are programmed by the user depending on the number of and size of the frames to know when to signal the *Cube DMA* that last value has been passed.

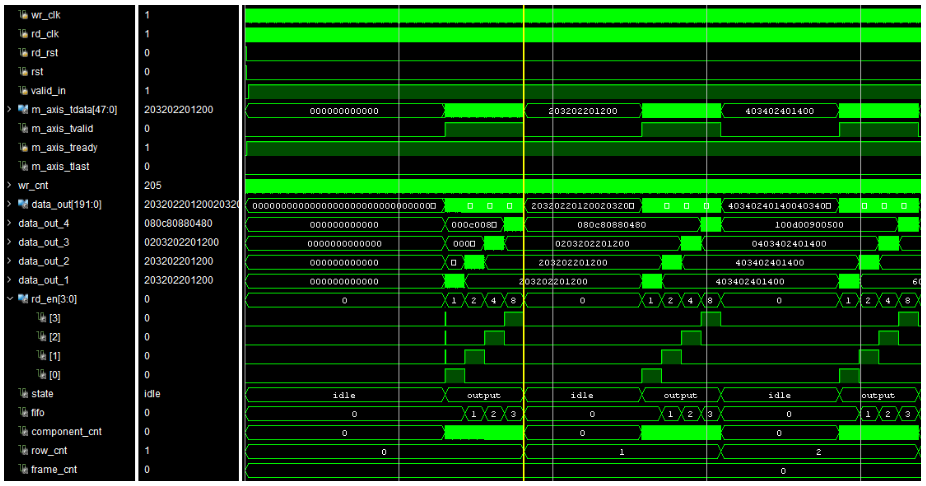


Figure 67: Waveform from simulation of Pixel Order Alignement Module.

It can be seen that the transfers to the *CubeDMA* happens in bursts. This is controlled by the *wr_cnt*. When the number of elements written to the module reaches the number set by the user the state machine starts outputting data. The *wr_cnt* is reset to 0 at this point starting to count written elements of the next row. The yellow marker in the figure is placed at the end of the first cycle of reading all the FIFOs and it can be seen in this example, with a row size of 512, that *wr_cnt* has reached 205 elements at this point which was calculated in the Method section in table 3 ($51.2 * 4 = 204.8$).

Synthesis The synthesis values for the relevant configuration are presented in table 9.

Pixel_Bit_Width	NUM_LVDS_PAIRS	PIXEL_ROW_SIZE	LUTS	Registers	Block Rams
12	4	512	995	1187	4

Table 9: Utilization report Pixel Order Alignement Module

7.1.3 Binning Module

The results from simulation and synthesis of the binning module is presented below.

Simulation Figure 68 shows a waveform from simulation of the *Binning Module* with a *Binning Factor* of 4. *s_axis_tvalid*, *s_axis_tready*, *s_axis_tlast* and *s_axis_tdata* is the signals for the input AXI-stream interface to the *Binning Module*. *m_axis_tvalid*, *m_axis_tready*, *m_axis_tlast* and *m_axis_tdata* is the output AXI-stream interface.

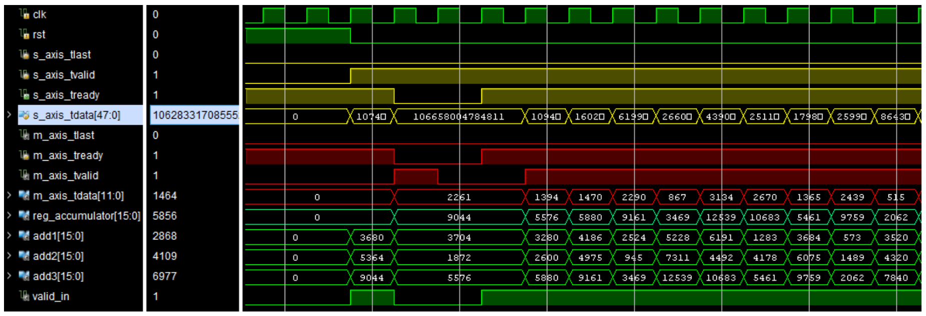


Figure 68: Waveform from simulation Binning Module with *Binning Factor* equal to 4.

reg_accumulator is the register storing the additions between the four inputs. *add1, add2* and *add3* is the resulting addition between input 1 and 2 (add1), input 2 and 3 (add2) and these results added together (add3). It can be seen that the values on the *m_axis_tdata* equals the value in the *reg_accumulator* divided by four, resulting in the average value of the four input pixel components. Also a *bubble* from the *Cube DMA* is added in the simulation meaning that there is a cycle where it needs a break. The module will in these situation, stop the output from the *Pixel Order Alignment Module* and stall until the *Cube DMA* is ready again.

Figure 69 shows a simulation of the same module with a *Binning Factor* of 8. In this case it can be seen that the *reg_accumulator* stores the result from the additions twice resulting in the sum of 8 pixel components. Then the *m_axis_tdata* outputs this result divided by 8.

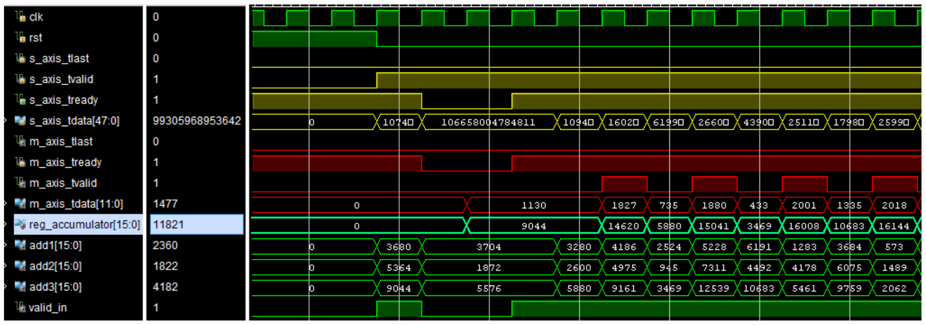


Figure 69: Waveform from simulation Binning Module with *Binning Factor* equal to 8.

Lastly is the configuration of *Binning Factor* of 16 showed in figure 70. Here it can be seen that the *reg_accumulator* accumulates the results from the additions 4 times, resulting in the sum of 16 pixel components. This is then divided by 16 and outputted at *m_axis_tdata*.

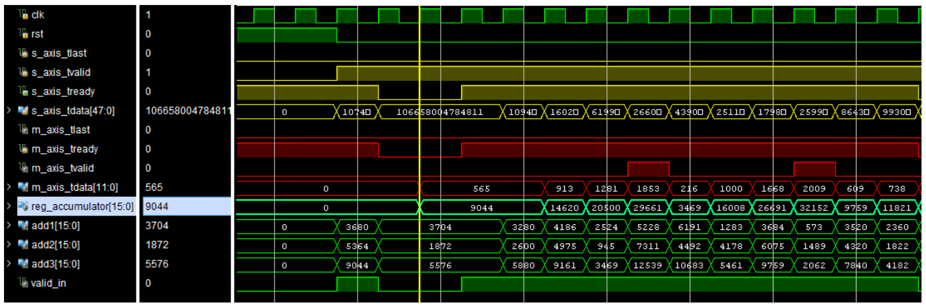


Figure 70: Waveform from simulation Binning Module with *Binning Factor* equal to 16.

Synthesis The synthesis values for the relevant configurations are presented in table 10.

Pixel_Bit_Width	NUM_LVDS_PAIRS	BINNING_FACTOR	LUTS	Registers
12	4	4	43	16
12	4	8	97	66
12	4	16	101	68

Table 10: Utilization report Binning Module

7.2 EMSC

In parallel with this thesis, a team was working on how data should be preprocessed and how the EMSC could be applied to hyperspectral imaging. Therefore the results from the EMSC implementations will not be analysed in the perspective of hyperspectral imaging but to how well the results matches what is calculated with the MATLAB script. This means that it is unknown how precise the results need to be compared to the MATLAB script but keeping the design as generic as possible lets the end user customize this to his use.

7.2.1 Software Implementation

Testing of the EMSC implementation was done concerning both the precision of the result and the time consumed executing the algorithm. A cube of 500x500 pixels containing 52 spectral components was used as test data. For profiling the cube size was also interesting so a subcube with 100x100 pixels was also used. This resulted in *raw* matrices 10000x52 and 250000x52. The test was executed by creating the *raw* matrix from a cube in MATLAB and writing this as *floats* to a binary file. This binary file was uploaded to the DDR memory of the Zynq on a Zedboard using the *XSDB* tool in the Vivado TCL shell. The EMSC software was executed. And the result was written as *floats* to a binary file, downloaded from the Zynq and analyzed in MATLAB.

7.2.1.1 MATLAB analysis of software implementation results MATLAB has been an important tool in analysis of the result. Both the available algorithms and the possibility to create visual representations have been helpful. The analysis will follow the same steps as they where done during this work. As there was a parallel development of the EMSC algorithm by the organisation whom handed out this task, there was not yet been decided what preprocessing was necessary on the cube data to give a correct result in term of a good way of representing the results in terms of the hyper spectral image that this data represents. In other words, there was not yet found a way to see if the EMSC produced a result only containing the chemical absorbance spectra. Therefore the quality of the results produced in this thesis was measured by comparing the results produced by the MATLAB script to the results produced from the implementations executed on the Zynq.

First off, both the Cubes (100x100x52 and 500x500x52) was streamed through the software implementation and the calculated corrected spectra was downloaded and inputted to MATLAB. Then it was compared to the result achieved with the MATLAB script. As the smallest cube is a subcube of the larger one, it was only interesting to analyse the large cube in terms of correctness as this would include the smaller cube.

Figure 71 show the inputted cube data. The x-axis shows the wavelengths from 1 to 52 and the y-axis represent the cube data values (raw).

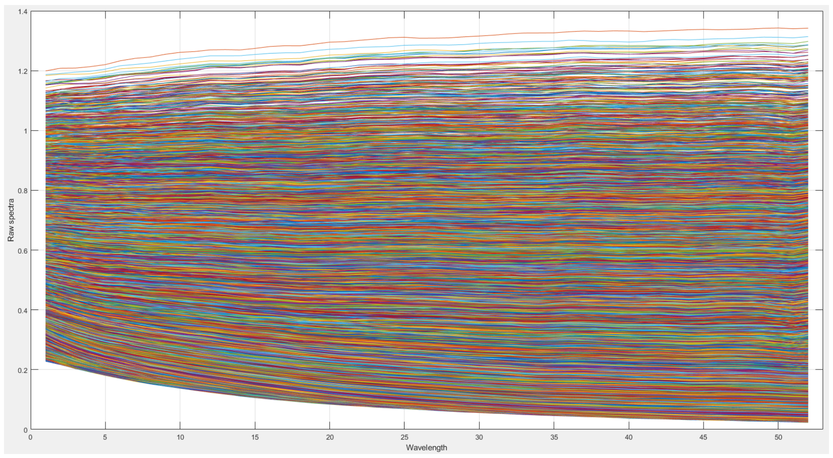


Figure 71: Plot of the raw data for the large cube (500x500x52)

Figure 72 shows the corrected spectra that was generated with the MATLAB script. This is interesting for comparison with the output from the implemented design.

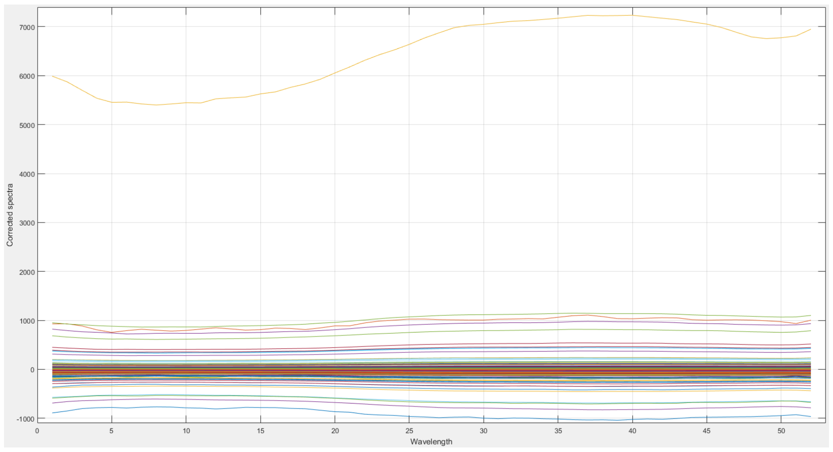


Figure 72: Plot of the corrected spectra for the cube produced with MATLAB

The corrected spectra produced by the software implementation executed on the Zynq is showed in figure 73. It is hard to visually observe any differences from these results compared to the ones generated with the MATLAB script.

By using MATLAB to compare the different corrected spectras it was easy to see the difference. This result is shown in the figure 74. First there are one pixel around the middle which has a mismatch of 150 and a few pixels at the rightmost side with a difference of around 25. The mean of differences across all pixels are -7.21e-04 showing that the majority of the result is pretty accurate to the result produced with the MATLAB script. As a reference to the

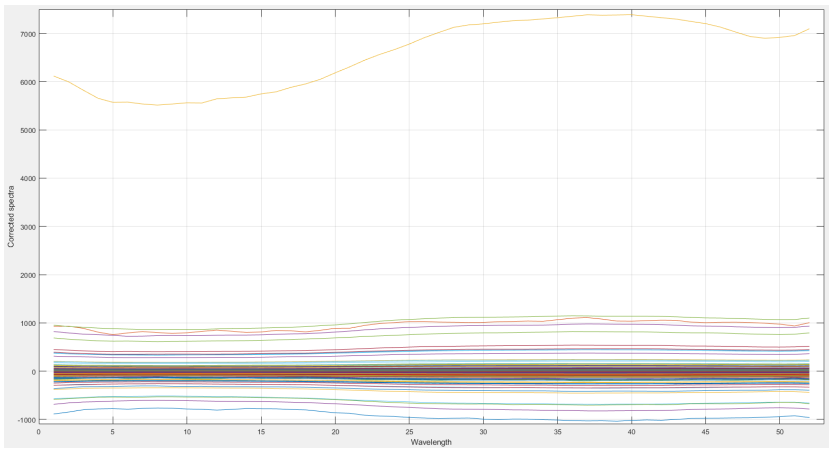


Figure 73: Plot of the corrected spectra for the large cube produced by software implementation on Zynq.

differences that was found above, the corrected spectra contains values in the range -1.036e+03 to 7.166e+03.

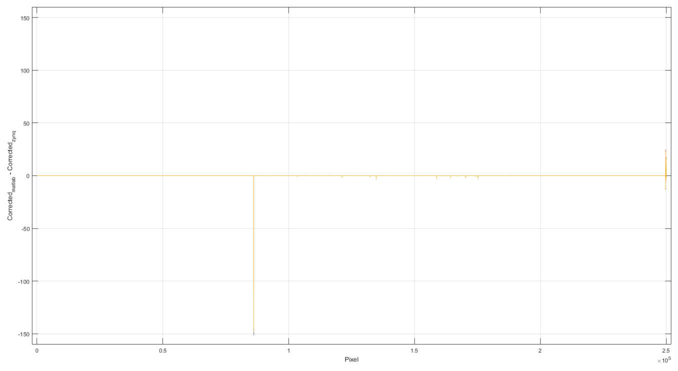


Figure 74: Plot of the difference between corrected spectra produced in MATLAB and in Zynq for the large cube (500x500x52)

Even though the analysis of the result collected from the cube data showed that the mean error was low a further analysis was initiated to investigate why there was some extreme values in some of the pixels. The procedure for this was to look at the **p** produced in the Zynq which is used to calculate the corrected spectra. Looking at this may reveal a bug or some other reason producing these pixel errors. Figure 75 shows the results from this. With the pixels along the x-axis and $P_{matlab} - P_{Zynq}$ on the y-axis. It can be seen that there is small differences accross the pixels but that the magnitude of this is below

3e-06. This should indicate that the calculation of **p** is correct, however as there is some conversions between float and double as the files are transferred between MATLAB and Zynq, this could be the result of this. Also floating point calculations could have some differences across architectures. Argument **G** was also analysed to look for sources that contributes to the error in calculation of the corrected spectra.

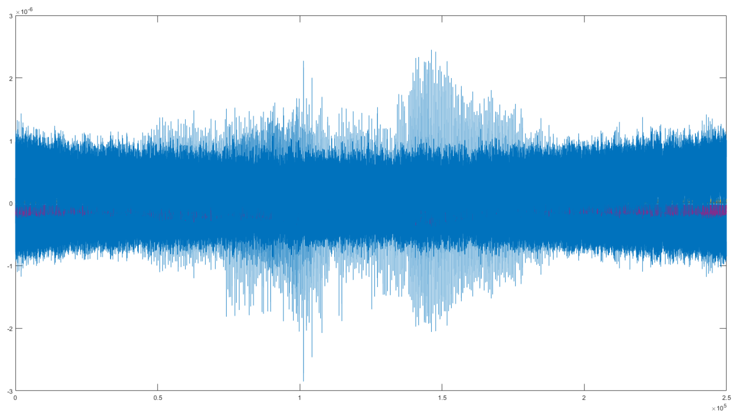


Figure 75: Plot of the difference between **p** produced in MATLAB and in Zynq for the large cube (500x500x52)

To recall, **G** is calculated by equation 10 and to achieve the pseudo inverse operation an external library called *Eigen* is used.

$$G = M' pinv(M * M') \tag{10}$$

Figure 76 shows the difference between the MATLAB and Zynq generated G. It can be seen that there are some differences but with a small magnitude. The largest difference has a magnitude of around -1.5e-04. The impact of these difference will have to be further analyzed with MATLAB.

As analysis of these results did not give a clear answer of what is causing this error. The results from the Zynq was tried reconstructed in MATLAB. This was done by using the **p** calculated on the Zynq in MATLAB to calculate the corrected spectra. Doing this gave the same error in the result and a conclusion could be done. By finding the specific pixel that had this large difference in figure 74 and looking at the corresponding **p** values leading to this values showed the problem. In the formula of calculating the corrected spectra repeated below (equation 11) it can be seen that the whole result is divided by the coefficient b_i. This is found in the last column of **p**. Comparing these values in the **p** generated in MATLAB and on the Zynq revealed the source of the problem. The MATLAB generated **p** had a value of 2.5548e-05 and the Zynq generated a value of 2.4786e-05. The difference between these values has a magnitude

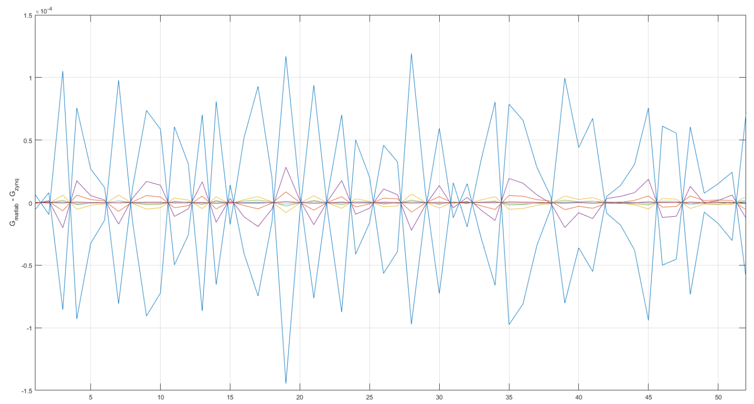


Figure 76: Plot of the difference between **G** produced in MATLAB and in Zynq.

of 7.63e-07 which is really small, however if you divide these values by 1, one gets a large difference of 1.205e+03 which leads to the inaccurate results in the corrected spectra.

$$\mathbf{z}_{i,corrected} = (\mathbf{z}_i - a_i - d_1\lambda - e_1\lambda^2)/b_i \tag{11}$$

This could indicate that the small differences when comparing the calculation of **G** in MATLAB and on the Zynq has a great impact on the final results and that a the problem seems to be in calculating the pseudo inverse on the Zynq. Calculating the inverse instead of the pseudo-inverse produced the same result as presented above.

7.2.1.2 Profiling The results from the profiling of the C++ implementation of the EMSC algorithm is shown in tables 11 and 12, for the small and large cube respectively. The *Inverse* and *Pseudo-Inverse* measurements from two different executions utilising these functions. The results shows that the *Inverse* calculation uses about 0.67 the time of the *Pseudo-Inverse* calculation. It can be seen that calculating the corrected spectra is the part that is running definitely slowest and would gain the most benefit from being implemented in hardware. The profiling was measured on the same cubes as mentioned earlier in the section. It can be seen that the percentages on the different code parts are not adding up to a 100%. This happens because the measurements was done over different runs. However, the results is precise enough to draw a conclusion of what parts dominates the execution time.

Code part	Clock Cycles	Time	Percentage
Entire EMSC function	31393157	0.314s	100%
Constructing M-Matrix	30448	0.304ms	≈0.1%
Inverse	319606	3.196ms	≈1.02%
Pseudo-Inverse	471393	4.714ms	≈1.5%
Calculating Corrected Spectra	31009146	0.310s	≈98.78%

Table 11: Results profiling of C++ implementation of small cube

Code part	Clock Cycles	Time	Percentage
Entire EMSC function	776305256	7.76s	100%
Constructing M-Matrix	30463	0.304ms	<0.01%
Inverse	319299	3.192ms	<0.1%
Pseudo-Inverse	472023	4.720ms	<0.1%
Calculating Corrected Spectra	775561033	7.75s	≈99.87%

Table 12: Results profiling of C++ implementation of large cube

7.2.2 Hardware Implementation

The hardware implementation was simulated block by block and all blocks connected together. It was experienced that if the simulation showed the correct result the implementation would execute correctly on the Zynq. The test bench that was used can be found in the appendix. It is written in verilog and utilises the ability to read and write to files. MATLAB was used to write the inputs to binary files which then could be read using the test bench. The results from the test bench then could be written to a binary file and MATLAB could be used to confirm a correct result. Figures 77-79 is examples of how the waveforms would look when simulating the *Block Ram Module*, *Dot Product Module* and the *Output Module*. In figure 77 the *Block Ram Module* is simulated. At the bottom it can be seen that the *v_len* and *R_order* has been written to through the AXI4-lite interface. The *v_len* is number of components in a pixel and is set to 52 and *R_order* is the number of rows in the *Reference spectra* and is set to 8. When *init* is set high the block rams are written to one by one and when 8 block rams is filled the *Initialized* signal is set high, indicating that the *Block Ram Module* is initialized. It can be seen that there is 16 block rams available but because the *R_order* is set to 8 only 8 of them is written to.

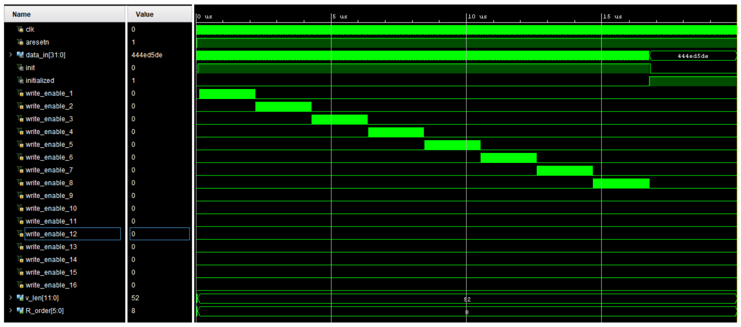


Figure 77: Waveform from simulating the Block ram module.

Figure 78 shows a simulation of the *Dot Product Module*. The waveform shows the calculation of the dot product between two pixels and the 4 first rows in the **G** matrix. The different colors separates the 4 different dot product calculations. When the *p_rdy* signal is set high the value in the accumulator, in this waveform the values in **p**, are outputted to the *Output Module*.

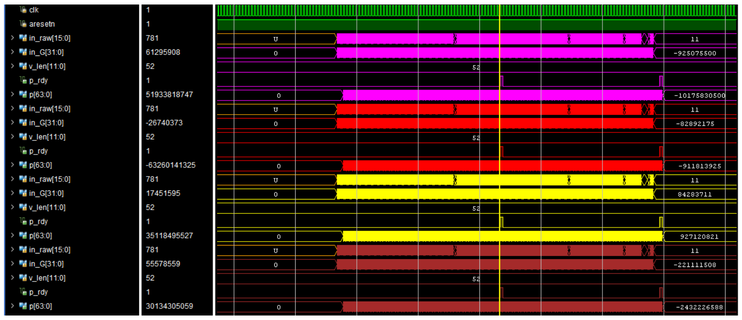


Figure 78: Waveform from simulating the Dot Product Module.

Figure 79 shows the simulation of the *Output Module*. *m_axis_tdata*, *m_axis_tvalid*, *m_axis_tready* and *m_axis_tlast* forms the AXI-stream interface for outputting the data.

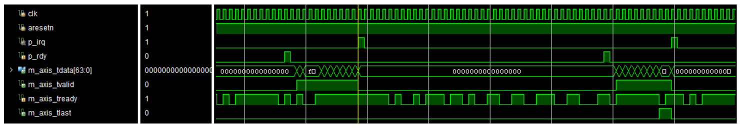


Figure 79: Waveform from simulating the Output Module.

When *p_rdy* is asserted the output module starts outputting the values to the *Cube DMA*. It can be seen that *bubbles* is simulated from the *Cube DMA* when it is not able to receive data. The *Output Module* however is designed to handle this. When a set of **p** has completed transfer the *p_irq* is asserted to signal the CPU that values are stored in memory.

7.2.3 HW/SW Implementation

The SW/HW implementation was tested in a similar way as the SW implementation. The same cubes, small and large, was streamed through the design and the result was analysed using MATLAB. Figure 80 shows the resulting design illustrated as a block diagram generated in Vivado. On the right side is the processing system of the Zynq 7000. In the middle we have *cubedma_top_v1_0* which is the Cube DMA and *top_0* which is the EMSC hardware accelerator design. The *axi_timer_0* is the timer which was used to measure the time for different part of the design. *Concat* is a block which concatenates the four interrupt signals into one 4 bit vector. The two blocks on the left *AXI Interconnect* and *Processor System Reset* is blocks which automatically is generated when the other blocks mentioned is connected to the Zynq 7000, which handles the AXI connections to the Zynq and the reset of the system. This design looks identical for both the Sequential and Parallel version. The only difference is how the *Cube DMA* is configured, depending on how many pixel component to transfer each cycle.

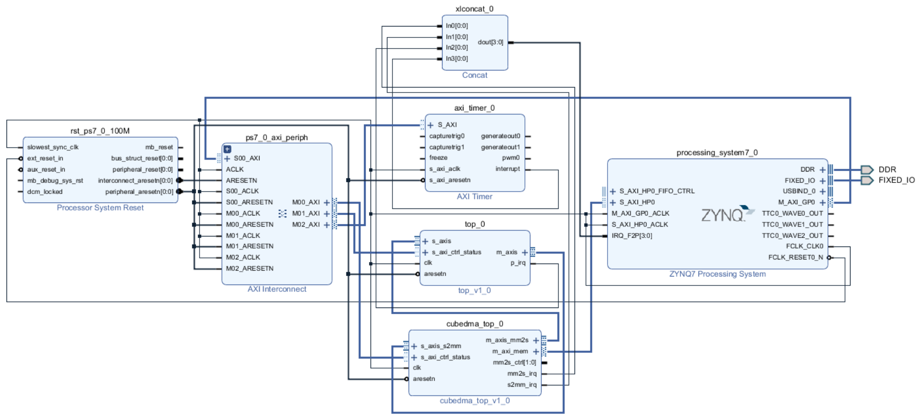


Figure 80: Block diagram of design from vivado.

7.2.3.1 Sequential Design

Synthesis To be able to run the hardware accelerator on the Zynq the design has to proceed through synthesis, implementation and a bitstream generator. The former produces the data that is uploaded to the Zynq and programs the FPGA. During synthesis and implementation, utilisation reports are generated which shows the resources spent to fulfil the design. Table 13 shows the default values chosen for this module. The synthesis was then done on multiple versions where the different parameters was changed. The generics in the table can be recognised in the VHDL implementation code in Appendig G.1.2.

Generic	Description	Default Value
B_RAM_SIZE	Max number of elements in Block ram	400
NUM_B_RAM	Max number of columns in G.	16
RAW_BIT_WIDTH	Bit width of raw components	16
G_BIT_WIDTH	Max bit width of G components	32
P_BIT_WIDTH	Bit width of accumulator in *Dot Product Module*	48

Table 13: Synthesis results

Figures 81-83 shows plots of the utilisation reports from the synthesis of the sequential design with different parameters. The results presented some interesting points which was not considered. It can be seen in figure 81 that increasing the bit width of **G** to 32 bits the synthesis tool divides the multiplication into two DSPs reducing the number of LUTS and registers. Timing reports showed that maximum clock frequency was 119MHz between 25-29 bits and 132 MHz between 30-32 bits meaning that increasing the bit width of **G** into using two DSPs in chain would still meet timing requirements of 100MHz.

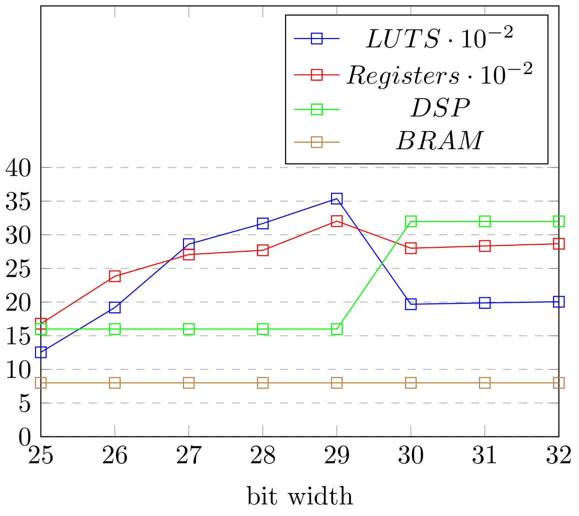


Figure 81: Utilization by increasing bit width of G

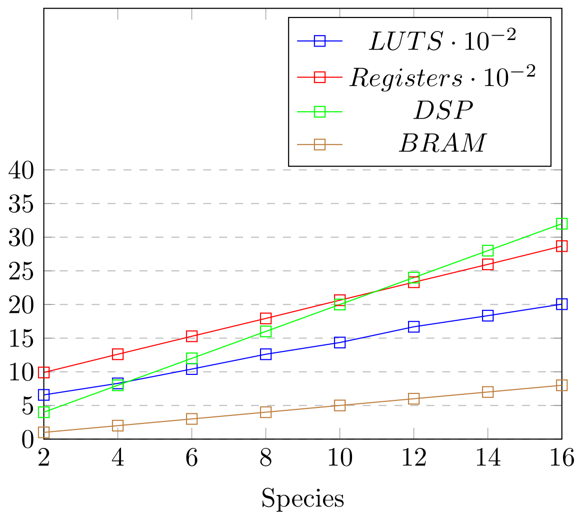


Figure 82: Utilization by increasing size of reference spectra

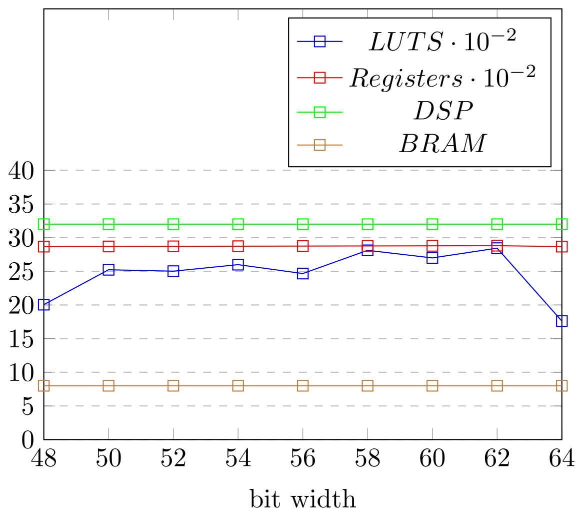


Figure 83: Utilization by increasing bit width of **p**

7.2.3.2 Profiling The same profiling was done for the SW/HW implementation and the results is shown below in table 14. Elements in the table with the subscripts illustrates where the blocking(^{1}) and non-blocking(^{2}) version of the *Calculating Corrected Spectra* was used. In the non-blocing version the execution of the *Cube DMA* is parallel with the *Calculating Corrected Spectra* and therefore the part *Initiate and executeCube DMA* only applies for the blocking version.

Code part	Clock Cycles	Time	Percentage
Entire EMSC function[1]	189563713	1.896s	100%
Entire EMSC function[2]	181243972	1.812s	100%
Constructing M-Matrix	30142	0.301ms	<0.1%
Initiate Block Ram	654965	6.550ms	<1%
Inverse	319299	3.193ms	<1%
Pseudo-Inverse	472023	4.720ms	<1%
Initiate and execute Cube DMA[1]	13000390	0.130s	6.9%
Calculating Corrected Spectra[1]	176753247	1.768s	≈93.2%
Calculating Corrected Spectra[2]	180741807	1.807s	≈99.7%

Table 14: Results profiling of sequential SW/HW implementation of large cube

Code part	Clock Cycles	Time	Percentage
Entire EMSC function[1]	14516844	0.145s	100%
Entire EMSC function[2]	13889585	0.139s	100%
Constructing M-Matrix	30142	0.301ms	<0.3%
Initiate Block Ram	654965	6.550ms	4.5%[1]-4.72%[2]
Inverse	319299	3.193ms	2.2%[1]-2.3%[2]
Pseudo-Inverse	472023	4.720ms	3.3%[1]-3.4%[2]
Initiate and execute Cube DMA[1]	520210	5.2ms	3.1%
Calculating Corrected Spectra[1]	12639504	0.126s	≈87.1%
Calculating Corrected Spectra[2]	13436007	0.134s	≈96.7%

Table 15: Results profiling of the sequential SW/HW implementation of small cube

7.2.3.3 Parallel implementation Also for the synthesis results of the parallel implementation some default values was chosen. These values are shown in table 16. The result found in the sequential version by increasing bit width of **G** also applied here. Timing however showed a bit lower max frequencies, 110MHz for bit widths between 25-29 bits and 119MHz for bit widths between 30-32 bits. This still meets timing requirement of 100 MHz. The results from the utilisation reports are reflected in the plots showed in figures 84-86.

Generic	Description	Default Value
B_RAM_SIZE	Max number of elements in Block ram	400
NUM_B_RAM	Max number of columns in G.	16
RAW_BIT_WIDTH	Bit width of raw components	64
G_BIT_WIDTH	Max bit width of G components	32
P_BIT_WIDTH	Bit width of accumulator in *Dot Product Module*	48
FIFO_DEPTH	Max number of elements in FIFO	512

Table 16: Synthesis results

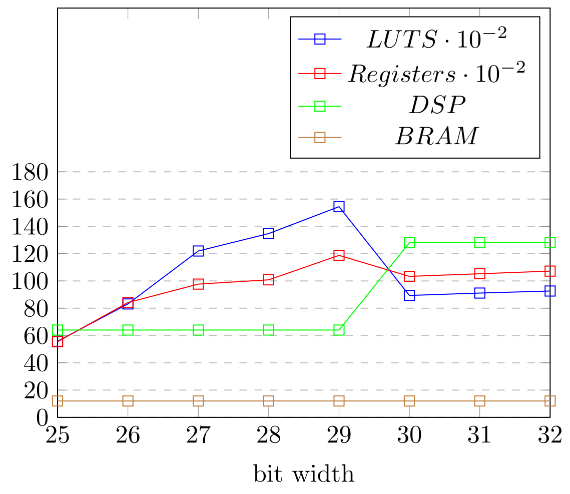


Figure 84: Utilization by increasing bit width of G

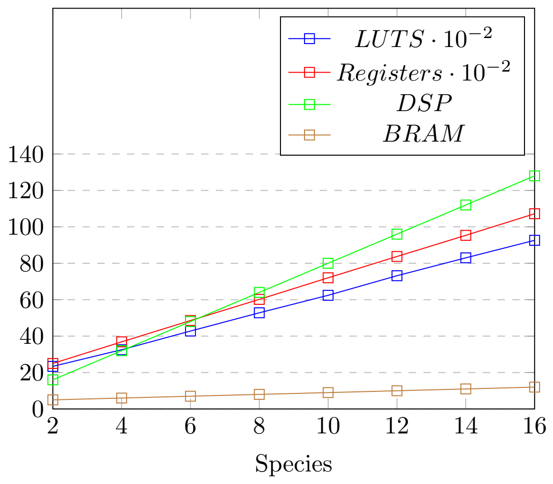


Figure 85: Utilization by increasing size of reference spectra

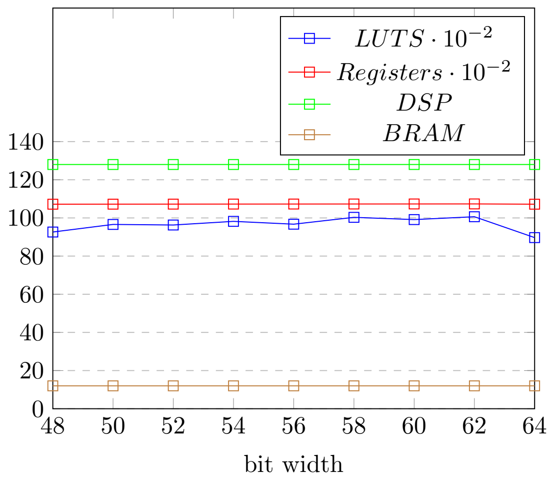


Figure 86: Utilization by increasing bit width of P

7.2.3.4 Profiling

Tables 17 and 18 shows the result from profiling the parallel implementation. As the only different values from earlier results is the *Cube DMA* execution time and the *Calculating Corrected Spectra* part all other parts has been omitted. It was not possible to produce results by using the non-blocking version on this design as the results was buggy and incorrect. This may be due to the heavier load of interrupts occuring.

Code part	Clock Cycles	Time	Percentage
Entire EMSC function	177980981	1.78s	100%
Initiate and execute Cube DMA	3251579	32.5ms	≈1.8%
Calculating Corrected Spectra	174247591	1.742s	≈97.9%

Table 17: Results profiling of the parallel SW/HW implementation of large cube

Code part	Clock Cycles	Time	Percentage
Entire EMSC function	10745513	0.107s	100%
Initiate and execute Cube DMA	131025	1.3ms	≈1.2%
Calculating Corrected Spectra	10520111	0.105s	≈97.9%

Table 18: Results profiling of the parallel SW/HW implementation of small cube

7.2.3.5 MATLAB analysis of HW/SW Implementation

MATLAB was used to both estimate the results and to compare this with actual results. The estimations was based on recreating the same operations executed on the Zynq in MATLAB. As the sequential and parallel version uses the same *Dot Product Module* the result was identical for both of them. The estimated calculated spectra was found by changing the lines 20-24 in the MATLAB script (figure 42 with the lines below (figure 87).

```
G = M'*pinv(M*M');
G = floor(G*mult);
P = zeros(nObs,size(M,1));
corrected = raw;
for idx = 1:nObs
    p = raw(idx,:)*G;
    P(idx,:) = p/mult;
    corrected(idx,:) = (raw(idx,:) - P(idx,1:mOrder+1) *M(1:mOrder+1,:)) / P(idx,end);
end
```

Figure 87: Lines replaced in MATLAB script to calculate corrected spectra

It is seen that the **G** is multiplied with a factor called *mult* and rounded, then **P** is divided with the same factor to calculate the corrected spectra. **P** was not declared in the original MATLAB script, but it was found practical to be able to output this from the function.

This method showed to be efficient as the calculated results was very close to the measured result from the data received from the Zynq. Starting with a multiplication factor of 2^{18} gave the calculated result showed in 88 and measured result showed in figure 89. Looking at the results however shows that using the 2^{18} gives multiple errors with an magnitude between 0-4000. En erroneous result using multiplication factor 2^{18} was predetermined in the *Design Choices* in the *Method* section. This means that the multiplication factor needs to be increased. However, this factor cannot be chosen randomly due to the risk of overflow in the accelerator.

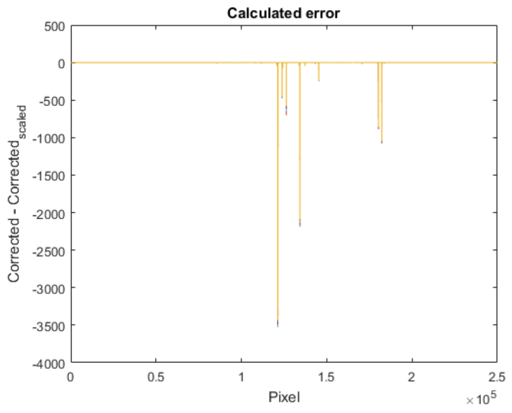


Figure 88: Calculated results for factor 2^{18}

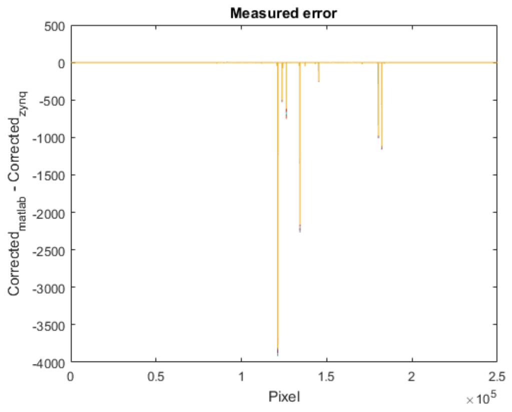


Figure 89: Measured results for factor 2^{18}

A way of calculating this is to take the absolute value of all elements in **G** and find the largest value. This can be used to calculate how many bits is needed to represent this value, which at this point cannot exceed 32 bits. Using equation (12) the range that can be represented with n-bits is calculated.

$$[-2^{n-1} \ to \ 2^{n+1} - 1] \tag{12}$$

Using this equation, it was found that 2^{22} was the highest factor that could be used and this gave the calculated and measured results as shown in figures 90-91.

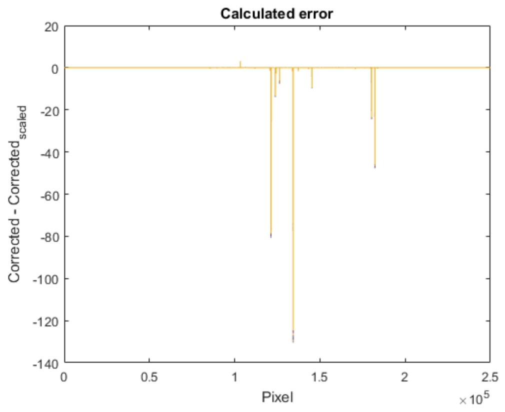


Figure 90: Calculated results for factor 2^{25}

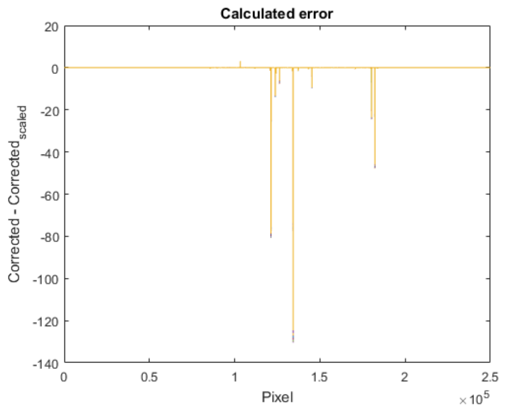


Figure 91: Measured results for factor 2^{25}

It can now be seen that the error is reduced below 150 which is a great improvement and shows that increasing the bit width increases the precision.

8 Discussion

The discussion section will be divided in two parts, one covering the *Image Sensor Pipeline* and one for the *EMSC* implementation.

8.1 Image Sensor Pipeline

The *LVDS Receiver*, consisting of the *Deserialiser* and the *Pixel Order Alignment Module* was a challenging design to implement. Starting with the *Simple Design* it was early in the process revealed from other designers experiences, that this design most likely where to naive and simple to fulfil the requirements of capturing data from LVDS transfers. However, as the Zynq 7000 platform already had primitives designed for this application as well as good documentation, both from Xilinx it self and other contributors on how to use these building blocks, the design developed to something more robust against the nature of the LVDS signals. The results from simulating the different blocks showed that they worked as intended. However, as was experienced during the EMSC development, is that the requirements are much stricter when the designs are to be executed on actual hardware. There was no opportunity to test the LVDS implementation with the Image Sensor which would have revealed the present flaws and bugs of the design, which there usually is before this have been done. Running the designs on hardware would also showed how the effects from different latencies accross the LVDS data channels and clock channel would have impacted the result. Synthesis results of the *Deserialiser* showed that it spends only a small amount of LUTS and registers because it utilises the Xilinx primitives available, which is good as long as these primitives are not needed for something else. For the *Pixel Order Alignment Module* the challenging part was the clock domain crossing from the 40 MHz on the LVDS side to the 100 MHz to the Cube DMA side. The asynchronous FIFOs that was found made this task feasible and the results from simulation satisfied the requirements. Also this module would have benefited from been executed on hardware to see if the assumptions that was made would hold. For example, the FIFO sizes which depends on the *Cube DMA* being able to transfer without a lot of *bubbles* to avoid full FIFOs resulting in loss of values.

Only 12 bit version of the *Pixel Order Alignment Module* was implemented. The main reason for this was that 12 bit was the desired operation mode for the Image Sensor. Another reason was that because the Cube DMA has a maximum data rate to memory of $64bits \times 100MHz = 6400\frac{Mbit}{s}$ and the 10 bit operation mode over 16 LVDS channels has a data rate of $10bits \times 16ch \times 48MHz = 7680\frac{Mbit}{s}$. This means that the *Cube DMA* would not have been able to receive an output data rate equal to the input data rate. A solution to this could be to use two *Cube DMAs*, however this would be complicated as they would need to be synchronised and know when and where the other *Cube DMA* wrote in memory. Or it could be sufficient with one *Cube DMA* if the *Binning Module* is used to reduce the input data rate.

Further down the pipeline we have the *Binning Module*. As this module is connected to the *Pixel Order Alignment Module* it was found reasonable to only implement this one for 12-bit as well. Simulations shows that this module works as intended. This is a simple module because the Binning operation is simple. However, this module might get more complicated in the future if it is found that a more sophisticated binning operation is desired.

8.2 EMSC

Looking at the results from the EMSC algorithm that was implemented it shows that the process from the pure software version to the hardware/software co-design version resulted in speedup. From the tables presented in the profiling section of the results, it can be seen that the sequential version resulted in a speedup of 2.17 and 2.26 for the blocking and non-blocking version respectively on the small cube (100x100x52). Speedup of 2.93 on the small cube with the parallel version. For the large cube (500x500x52) it resulted in a speedup of 4.1 and 4.28 for the blocking and non-blocking sequential versions and 4.36 for the parallel version. The differences in speedup between the different cube sizes might be explained by the overhead time for initiating the accelerator which is fixed for both cube sizes. This will then impact the speedup more the smaller the data size is. It should be mentioned that the pure software version used around 1.7s and 22s for the small and large size in its first version. This was due to read and write operations directly on the *Eigen* type matrices inside the *Calculating Corrected Spectra*. By first copying these matrices into a standard C++ pointer the computing time was reduced to what was presented in the result section. The accelerator that was implemented was focused around producing the dot product that was needed to calculate the correct result in software. When the parallel version was implemented the dot product calculation reached a point where the arithmetic operations in software was the main contributor to the execution time. This resulted in that the parallel implementation did not improve the speedup as much as expected. However, if a version including the arithmetic operations in the accelerator was implemented this could show the true potential of the parallel implementation.

Concerning the precision it was helpful to find that the results could be predicted by using MATLAB, and that by using the largest bit width of **G** possible in this version, the precision was almost as good as the software version. There was a few pixels with a significant incorrect result but across all pixels the mean error was low. The prediction however indicated that by increasing the bit width of **G** even more this error was further lowered. This would however required some changes to this design. The AXI register interface that was used in the implementation which among other things transfer data to the block rams in the initialisation phase was set to transfer data of 32-bits. If the bit width of **G** was to be increased this could either be solved by increasing the data size in this AXI register to for example 64 bits. Or a 64 bit number could have been transferred using two 32 bit register. This is something that is interesting for future work to see what precision could be achieved. It was also gathered results in the synthesis that was not considered at the start of the design process. That multiple DSPs was chained to support larger bit widths could have changed the initial design choices where it first was thought that **G** had to be restricted to 25 bits to fit the DSP. If the implementation initially was designed for this, the resulting implementation may have been designed for larger bit widths through the process. However, this is also something that will be addressed in the future work section.

It was also shown in figure 83 and 86 that increasing the bit width of **p** did not effect the resource usage considerable and that a bit width of 64 bits actually had a smaller resource usage than the default value of 48 bits. Therefore the value of this bit width should be set to 64 bits as default. The reason this was set to a default of 48 bits is that the size of the accumulator in the DSP has a bit width of 48 bits. It seems that the synthesis tool do not use the accumulator in the DSP to sum the products of the multiplications in the *Dot Product Module* as first assumed but instead implements different logic to do this operation. This is also something that will be mentioned in future work as a version where the accumulator inside the DSP is used when the bit width of **p** is less than or equal to 48 bits to reduce surrounding logic.

Lastly, the importance of verification and testing should be mentioned. It was early discovered in this process that spending time on testbenches was really important to be able to make this design execute on the hardware. Many hours was spent debugging and trying to understand why the designs would not work when they executed on the FPGA. The debugging cores contained in Vivado was a really helpful tool as well as trying to cover all possible sources to error in the testbenches that was made. For example, the *Cube DMA* had some cycles where it was not able to receive data, earlier referred to as *bubbles*. These was really hard to detect with the debugging core as they might occur in the 15 transaction or the 13000000 transaction. When this was included into the testbench and the design was modified to handle these, it suddenly worked when executed on the FPGA. This are just an example of many illustrating the importance of the simulation and the requirements of the testbench.

9 Conclusion

As was mentioned in the discussion part, the initial stages of the design process is important and being able to map and organise the requirements for the design in such a way that good choices are made through the whole loop. For the LVDS receiver interface it was revealed that the first simple design was in many cases not sufficient to fulfil requirements because of the nature of LVDS signals. This was considered and resulted in a more robust solution, utilising primitives from Xilinx own ip catalogue, building blocks tailored for this application. It must however be stressed, after the HW/SW co-design process of EMSC, the path from having a module working seamlessly in simulation to have it working on the FPGA may be long. By adapting this experience for the LVDS receiver interface, one would probably reveal bugs and improve this design if it was executed and tested on actual hardware. The EMSC implementation showed that the process of making hardware and software working together in a so called Hardware/Software co-design is a challenging process consuming many hours of testing and failing. However, when the designs finally worked, the results both showed promising points as well as potential improvements of the designs.

10 Future Work

As there are multiple potential improvements to the designs there a few thing that should be done in the future. First of all the LVDS receiver interface should be tested with the image sensor to capture data. Also as was mentioned in the discussion section, increase the maximum bit width of **G** from 32 to 64 bits to see if improvements in precision is achievable. Also a better framework for the software implementations that are done could be improved, as changes in the code will need to be done multiple places if different data sizes of *raw* is used. Another thing mentioned in the *Result* section was that a improvement that make sure the accumulator inside the DSP is used when the bit width of **p** is less than or equal to 48 bits which will increase the efficiency of the design. This could alternatively be a parameter that lets the user configure this as desired. Lastly *Pixel Alignment Module* and *Binning Module* could be implemented to support 10-bit as the *LVDS Deserializer* supports this. As this is not currently a requirement because the Image Sensor is to be used in 12-bit mode it should not be a priority, however for the future if this LVDS design is to be used on different devices it could be nice to have.

References

[1] ARM. Amba® axi™ and ace™ protocol specification, 2003.

[2] AVNET. Picozed.

[3] Greg Burton. Xilinx: 16-channel, ddr lvds interface with per-channel alignment, 2006.

[4] CMOSIS. Cmv2000-datasheet-v3.8, 2015.

[5] Marc Defossez. Xilinx: Serial lvds high-speed adc interface, 2006.

[6] Johan Fjeldtvedt. *Direct Memory Access for Hyperspectral Imaging Applications.* NTNU, 2017.

[7] Thomas Grob. Implementation of a fpga-based interface to a high speed image sensor, 2010.

[8] Jesper Pram Nielsen Harald Martens and Søren Balling Engelsen. *Light Scattering and Light Absorbance Separated by Extended Multiplicative Signal Correction. Application to Near-Infrared Transmission Analysis of Powder Mixtures.* 2003.

[9] TEXAS INSTRUMENTS. *Understanding Serial LVDS Capture in High-Speed ADCs.* July 2013.

[10] John Goldie National Semiconductor: Syed B. Huq. An overview of lvds technology, 1998.

[11] A. Kohler Nils.K Afseth. *Extended multiplicative signal correction in vibrational spectroscopy, a tutorial.* ELSEVIER, 2012.

[12] The University of Sydney. What is vibrational spectroscopy?

[13] Accessed 12.03.18 R. Mark Elowitz. What is imaging spectroscopy (hyperspectral imaging)?

[14] Julian Veisdal. Hardware hsi design/elecronics design.

[15] Julian Veisdal. Ntnu small satellite lab wiki/mechanical design, 2018.

[16] Wikipedia. Spectrograph, 2018.

[17] Xilinx. 7 series fpgas selectio resources.

[18] Xilinx. Logicore ip axi4-lite ipif v2.0, 2013.

[19] Xilinx. Logicore ip floating-point operator v7.0, 2014.

[20] Xilinx. Vivado design suite 7 series fpga and zynq-7000 all programmable soc libraries guide, 2016.

[21] Xilinx. 7 series dsp48e1 slice, 2018.

A EMSC Software Implementation

Listing 1: C++ code using listings

```
#include <stdio.h>
#include "xil_printf.h"     //Printf for Uart
#include "Eigen/dense"      //Eigen
#include <stdlib.h>         //atof
#include <math.h>           //Pow, sqrt
#include <float.h>
#include "xparameters.h"    //Board specific parameters
#include "xuartps.h"        //Uart
#include <string.h>
#include "xtmrctr.h"        //Axi Timer

//Eigen
using Eigen::MatrixXd;

//Axi timer
#define TMRCTR_DEVICE_ID  XPAR_TMRCTR_0_DEVICE_ID
#define TIMER_COUNTER_0  0
XTmrCtr TimerCounter;

//Uart
#define UART_DEVICE_ID      XPAR_PS7_UART_1_DEVICE_ID
XUartPs Uart_Ps;


/*Function Prototypes ******************************/
void mean(double** ref_spectra, double* mean,  int nVars, int refOrder);
void EMSC(double ** raw, double ** ref_spectra, int nVars,
          int nObs, int refOrder);
double ** initialize(int rows, int columns);
int init_timer(u16 DeviceId, u8 TmrCtrNumber);
u32 start_timer(u8 TmrCtrNumber);
u32 stop_timer(u8 TmrCtrNumber);
/**************************************************/




double ** initialize(int rows, int columns) {
    double **temp;
    temp = (double **)malloc(rows * sizeof(double*));
    for (int row = 0; row < rows; row++) {
        temp[row] = (double*)malloc(columns * sizeof(double));
    }
    return temp;}


void EMSC(double ** raw, double ** ref_spectra, double * mean_spectra,
          int nVars, int nObs, int refOrder){

    //DECLARATIONS---------------------
    MatrixXd M(refOrder + 4, nVars);
    double ** G = initialize(nVars, refOrder+4);
    double ** corr_M = initialize(2, nVars);
    double* p = (double*)malloc((refOrder + 4) * sizeof(double));
    double num = 0;
    //----------------------------------
    //Start timer
    u32 value1, value2;
    init_timer(TMRCTR_DEVICE_ID, TIMER_COUNTER_0);
    value1 = start_timer(TIMER_COUNTER_0);

    xil_printf("Constructing M!\n");
    for (int i = 0; i < nVars; i++) {



        //Add 1 in first row
        M(0,i) = 1;

        //Add linspace and linspace squared
        M(1,i) = num;
        corr_M[0][i] = num;

        M(2,i) = pow(num, 2);
        corr_M[1][i] = pow(num,2);
        num += (1.0 / (nVars - 1));

        //Add reference spectra
        for (int y = 0; y < refOrder; y++) {
            M(y + 3,i) = ref_spectra[y][i];
        }
        //Add mean in last row
        M(refOrder+3,i) = mean_spectra[i];
    }
    //Stop timer and output results
    value2 = stop_timer(TIMER_COUNTER_0);
    xil_printf(" Construct M Timer: %d\n", value2-value1);

    //float * mem_ptr = (float*)0x13197508;
    //int index = 0;
    //float * mem_ptr = (float*)0x13197508;
    //int index = 0;

    //Start timer
    init_timer(TMRCTR_DEVICE_ID, TIMER_COUNTER_0);
    value1 = start_timer(TIMER_COUNTER_0);

    //Execute pseudo-inverse of M
    MatrixXd M_M = M*M.transpose();
    MatrixXd p_inv = M.transpose() * M_M.completeOrthogonalDecomposition().pseudoInverse();
    //MatrixXd p_inv = M.transpose() * M_M.inverse();
    xil_printf("Pseudo-Inverse Completed!\n");
    for(int i = 0; i<nVars; i++){
        for(int y = 0; y<refOrder+4; y++){
            G[i][y] =(double) p_inv(i,y);

        }
    }
    //Stop timer and output results
    value2 = stop_timer(TIMER_COUNTER_0);
    xil_printf(" p_Inverse Timer: %d\n", value2-value1);

    float * mem_ptr = (float*)0x19CD1534;
    int index = 0;

    //Start timer
    init_timer(TMRCTR_DEVICE_ID, TIMER_COUNTER_0);
    value1 = start_timer(TIMER_COUNTER_0);

    //Calculate the corrected spectra
    xil_printf("Calculating Corrected Starting!\n");
    double sum = 0;
    for (int idx = 0; idx < nObs; idx++) {

        for (int i = 0; i < refOrder + 4; i++) {
            for (int y = 0; y <nVars; y++) {
                sum += raw[idx][y] * G[y][i];
            }
            p[i] = sum;
            sum = 0;
        }

        for (int t = 0; t < nVars; t++) {
            mem_ptr[index++] = (raw[idx][t] - p[0] - p[1] * corr_M[0][t] - p[2] * corr_M[1][t]) / p[refOrder
                  + 3];
        }

    }
    //Stop timer and output results
    value2 = stop_timer(TIMER_COUNTER_0);
    xil_printf(" Calculating corrected Timer: %d\n", value2-value1);


}

//Axi-Timer
//-----------------------------------------
int init_timer(u16 DeviceId, u8 TmrCtrNumber){
    int Status;
        XTmrCtr *TmrCtrInstancePtr = &TimerCounter;
        /*
         * Initialize the timer counter so that it's ready to use,
         * specify the device ID that is generated in xparameters.h
         */
        Status = XTmrCtr_Initialize(TmrCtrInstancePtr, DeviceId);
        if (Status != XST_SUCCESS) {
            return XST_FAILURE;
        }
        /*
         * Perform a self-test to ensure that the hardware was built
         * correctly, use the 1st timer in the device (0)
         */
        Status = XTmrCtr_SelfTest(TmrCtrInstancePtr, TmrCtrNumber);
        if (Status != XST_SUCCESS) {
            return XST_FAILURE;
        }
        /*
        * Enable the Autoreload mode of the timer counters.
        */
        return XST_SUCCESS;}

u32 start_timer(u8 TmrCtrNumber){
    XTmrCtr *TmrCtrInstancePtr = &TimerCounter;
    XTmrCtr_SetOptions(TmrCtrInstancePtr, TmrCtrNumber,
                            XTC_AUTO_RELOAD_OPTION);
    u32 val = XTmrCtr_GetValue(TmrCtrInstancePtr, TmrCtrNumber);
    XTmrCtr_Start(TmrCtrInstancePtr, TmrCtrNumber);
    return val;}

u32 stop_timer(u8 TmrCtrNumber){
    XTmrCtr *TmrCtrInstancePtr = &TimerCounter;
    u32 val = XTmrCtr_GetValue(TmrCtrInstancePtr, TmrCtrNumber);
    XTmrCtr_SetOptions(TmrCtrInstancePtr, TmrCtrNumber, 0);
    return val;
}
//------------------------------------------
int main(){
    //Adding pointer to location of stored cube.
    float * mem_ptr = (float*)0x10000000;
    int nVars = 52; //number of wavelenghts
    int nObs  = 250000; //total number of pixels
    int refOrder = 4; //numbers of species in spectra
    double ** raw = initialize(nObs,nVars);
    double ** ref_spectra = initialize(refOrder, nVars);
    //double ** corrected = initialize(nObs, nVars);
    double * mean_v =  (double*)malloc(nVars * sizeof(double));

    //Fill raw matrix1
    int index = 0;
    mem_ptr = (float*)0x100010E0;
    for(int rows = 0; rows < nObs; rows++){
        for(int cols = 0; cols < nVars; cols++){
            raw[rows][cols] = (double)mem_ptr[index++];
        }
    }
    //Construct some reference spectra
    //Just using some spectras from raw in this case
    //as an example.
    index = 0;
    mem_ptr = (float*)0x10000000;
    for(int rows = 0; rows<refOrder; rows++){
        for(int cols = 0; cols < nVars; cols++){
            ref_spectra[rows][cols] = mem_ptr[index++];
        }
    }

    mem_ptr = (float*)0x10001000;
    for(int i = 0; i<nVars; i++){
        mean_v[i] = mem_ptr[i];
    }
    //calculate mean of ref_spectra


    //Start the EMSC
    xil_printf("EMSC Starting!\n");
    EMSC(raw, ref_spectra, mean_v, nVars, nObs, refOrder);



    xil_printf("Done");
        return 0;
}
```

B Tutorials

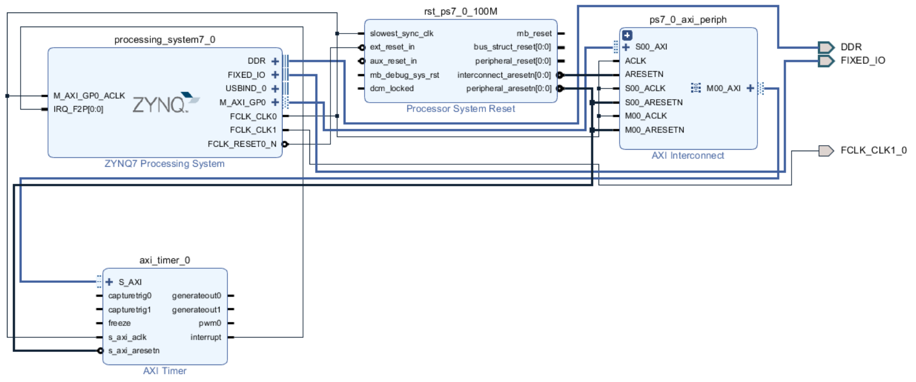
B.1 EMSC software implementation on Zedboard

This tutorial shows how to run the C++ software implementation of the EMSC algorithm presented in this paper. First a look on how to implement the required hardware platform in Vivado.

B.1.1 Building the Hardware in Vivado

Create a new project and a block diagram in Vivado. Figure 92 shows what hardware platform was used to run the EMSC. First, add the *ZYNQ7 Processing System* and run the block automation. Then add the *AXI-Timer* and connect it as shown in the figure. The *FCLK_CLK1_0* can be ignored as this was used for another application.

Figure 92: HW platform Vivado



When the clock diagram are completed, run synthesis, implementation and generate bitstream. When this is completed, open the implemented design and export the hardware. This is done by first pressing the *File* in the upper left corner, then: Export → Export Hardware. **Include Bitstream!**. Then press *Launch SDK* from the same menu (*File*).

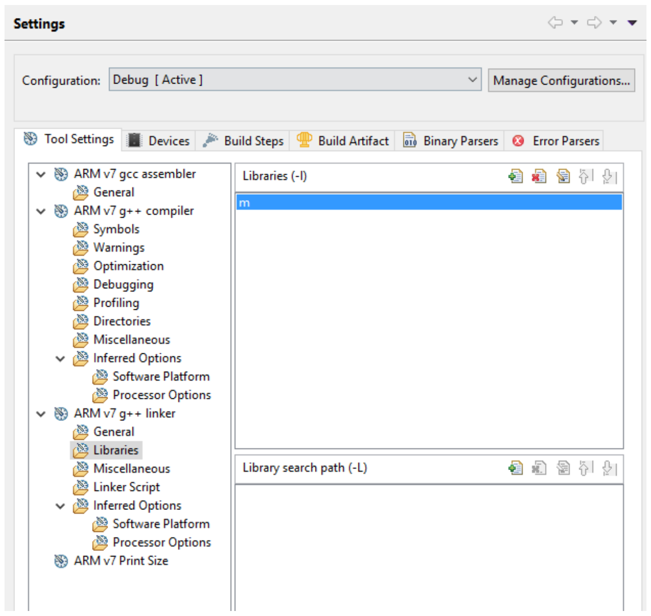
B.1.2 Setting up the SDK Environment

Inside the SDK we first need to create a new *Board Support Package* and *Application Project*. Press *File* → *New* → *Board Support Package*. Just keep the defaults and press *Finish*. Now press *File* → *New* → *Application Project*. Choose an appropriate name, make sure *C++* is checked, and under *Board Support Package* **Use existing** and choose the one just created. Press *Next* → *Empty Application* → *Finish*.

Now an application project should be visible in the *Project Explorer*. Open this folder and right click *src*, *(*new) → *Source File*. Name this *main.cpp* and press *Finish*.

Copy the EMSC source code inside *main.cpp*. Before this can compile some libraries has to be included. This is *math* and the *Eigen* library. Right click the top folder for the application project that was created before and press the *C/C++ Build Settings*. Under *"ARM v7 g++ linker"* → *Libraries*, add *m* as shown in figure 93.

Figure 93: Adding math library



Under *"ARM v7 g++ compiler"* → *Directories* add the include path for the top folder where the *Eigen* library is located. This library has to be downloaded, and can be found here: `http://eigen.tuxfamily.org/index.php?title=Main_Page#Download`. Also before the code can be launched the *heap* and *stack* sizes in the linker script will have to be updated. Right click the application project and press *Generate Linker Script*. Here change the heap to 255MB and stack to 64KB, this is a know working setting. Having to low values will give strange or no behaviour. Now the code should compile and be ready to execute.

B.1.3 Launching EMSC

This algorithm needs data to be able to produce some sensible result. This is done by laying the *raw* matrix in memory. There are multiple ways of doing this but in this tutorial it will be presented by using the TCL tool that comes with Vivado. First of all, program the FPGA and execute an empty main(Just comment everyting inside main). This will setup the board and make it possible reading and writing to the memory through TCL. Use the MATLAB script *Generate_Cube_Data* with the *Hico_Canary_Volcano* cube and go through the sections of the script. When an image shows, one has to choose 4 pixels. The first pixel should be from dark water and the second from the snow in the image. The last two pixels can be arbitrary. Continue through the sections until you get to the one called *Subset*. Because the entire cube is time consuming to load in and out of the memory of the Zynq, use the small data version called *x_small*. Then run the section *Write cube to file*. Now the *raw* matrix has been written to a binary file saved in the same directory as the MATLAB Script. Now open the TCL window and write *xsdb*. Followed by *connect* and *targets 2*. Navigate inside the TCL window so you are in the same directory as the saved binary file. Then write *mwr -force -bin -file cube_test.bin 0x10000000 520000*. This means will transfer the binary file into location 0x10000000 of the Zynq DDR memory and across the next 520000 addresses. When this is done, the EMSC algorithm can be launched. After this has completed (This can be seen from messages over UART if this has been configured) the corrected spectra can be read out. As can be seen from the code, the corrected spectra is saved at memory location 0x13197508. Using the TCL and writing *mrd -force -bin -file zynq_final.bin 0x13197508 520000*. This will read out the values from the memory and store them in a binary file *zynq_final.bin* in the same directory as the MATLAB script. Use the MATLAB script to read the data from Zynq and compare it with the corrected file that is calculated using MATLAB (produce this corrected by running the section *Calculate Corrected*).

B.2 EMSC Software/Hardware co-design on Zedboard

This tutorial will show ho to execute the Software/Hardware Co-design implementation on a Zedboard. First we will start with building the Hardware in Vivado.

B.2.1 Building the Hardware in Vivado

Figure 94 shows an overview of how the hardware should be build in Vivado Block Diagram. Together with this the *Cube DMA* and the *EMSC Accelerator* has to be configured. Double click the *cubedma_top_0* and make sure the configurations for you application is correct.

For this tutorial the following settings are used.

1. C Mm2s Axis Width = 16
2. C Mm2s Comp Width = 16
3. C Mm2s Num Comp = 1
4. C S2mm Axis Width = 64

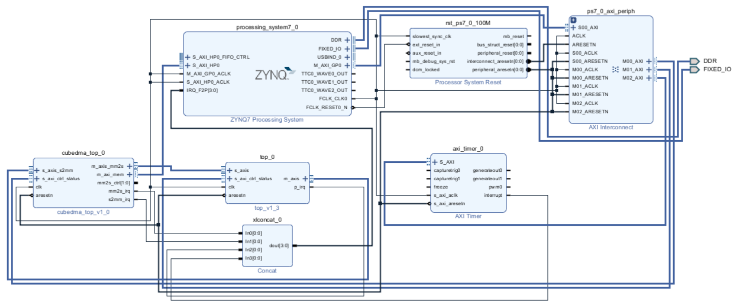


Figure 94: Overview of hardware in Vivado.

5. C S2mm Comp Width = 64

6. V S2mm Num Comp = 1

Double click the *top_0* and configure this as well. For this tutorial the following settings are used.

1. B Ram Size = 400

2. C S Axi Addr Width = 32

3. C S Axi Data Width = 32

4. G Bit Width = 32

5. Num B Ram = 16

6. P Bit Width = 64

7. Raw Bit Width = 16

When all modules are connected as shown above and the configurations are correct. Press the *Tools* → *Validate Design* to make sure the design contains no errors. Then save the block diagram and run synthesis.

B.2.2 Setting up debug cores

When the synthesis is completed we can setup debug cores which are handy when executing the design to see that it works as it should. Press the *Open Synthesized Design* in the *SYNTHESIS* menu on the left side in Vivado and press the *Schematic* option. This will open a schematic of the synthesized design. Press the darker blue square in the top left corner of the design to expand its contents. Find the *Cube DMA* and mark following signals for debug by right clicking and pressing *Mark Debug*. *m_axis_mm2s_tready*, *m_axis_mm2s_tdata*, *m_axis_mm2s_tlast*, *m_axis_mm2s_tvalid*, *mm2s_irq*, *s2mm_irq*, *s_axis_s2mm_tdata*, *s_axis_s2mm_tlast*, *s_axis_s2mm_tvalid* and *s_axis_s2mm_tready*. Find the accelerator and mark *p_irq* for debug. Press ctrl+s and save the debug constraints with a suiting name, for example, *debug*. Press the *Set Up Debug* in the *SYNTHESIS* menu and follow the wizard. If any of the signals does not have a clock domain choose one. The rest can be let to the defaults. When it completes, run synthesis, implementation and generate bit stream.

B.2.3 Setting up the Xilinx SDK

When bitstream has been generated, open the implemented design. Then press *File* → *Export* → *Export Hardware* and make sure *Include Bitstream* is enabled. Press *File* → *Launch SDK* and the Xilinx SDK will open. Inside the SDK, press *New* → *Board Support Package* and add this. Then *New* → *Application Project*. Find a suiting name, choose C++ and press *Use existing* for the *Board Support Package*. Navigate to main.cc and add the code. Then comment everything in main and execute the code, this initialises the board. This requires the Zedboard to be connected. Go back to Vivado and open the Hardware Manager. Press the *Open Target* and then *Program Device*. This will open the debug core interface. Drag the signals *m_axis_mm2s_TREADY* and *m_axix_mm2s_TVALID* down to the trigger setup and set their value to 1. This is what decides when the debug core should trigger. Then press the play button to set the trigger. Now we switch back to the SDK. Right click on the *Application Project* and press *C/C++ Build Settings*. Under *Arm v7 g++ linker* select *Libraries*. Press the add button and write *m* and press *OK*. This will make the math library available in the project. Then under *ARM v7 g++ compiler* select *Directories* and add the path of the folder where the *Eigen* library is located. Press *Apply* and *OK*. Right click the application project again and select *Generate Linker Script*. For this tutorial add 200 MB (209715200) in the *Heap Size* and 100KB (102400) in the *Stack Size* and press *Generate*.

B.2.4 Uploading data to memory

The data to execute the EMSC on has to be uploaded to the memory of the Zynq. Open Vivado TCL and write *xsdb* in the command line. Then write *connect* and *targets 2*. Navigate to where the binary file of the raw is contained and write the following command *mwr -force -bin -file raw.bin 0x100010E0 size*. Size is the number of elements to transfer. For a 500x500x52 cube stored as int32 in the binary file has a size of 13000000. Navigate to the reference spectra and write the following command *mwr -force -bin -file reference_spectra.bin 0x10000000 size*. Navigate to the mean you want to input and write *mwr -force -bin -file mean.bin 0x10001000 size*. Now the required data is stored in memory and the code is ready to be executed.

B.2.5 Executing the EMSC code

When the code is executed the debug core should trigger. Figure 95 shows the waveform from this Debug Core. It can be seen that the first component fetched from the *Cube DMA* has a value of 839 which is correct. It can also be seen that the *Cube DMA* has a bubble right before 600 and that the accelerator handles this by holdin the output until the *Cube DMA* is ready. Also it can be seen that the output from the accelerator is happening in bursts. When the execution has completed the results can be downloaded as binary files from memory using the TCL. **P** is stored in address 0x0F0BDBF0 and corrected spectra is stored in address 0x19CD1534. Reading can be done with the following command *mrd -force -bin -file name_of_file_to_store.bin size.*

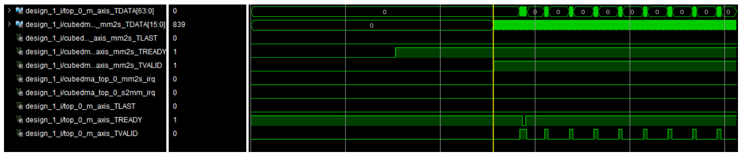
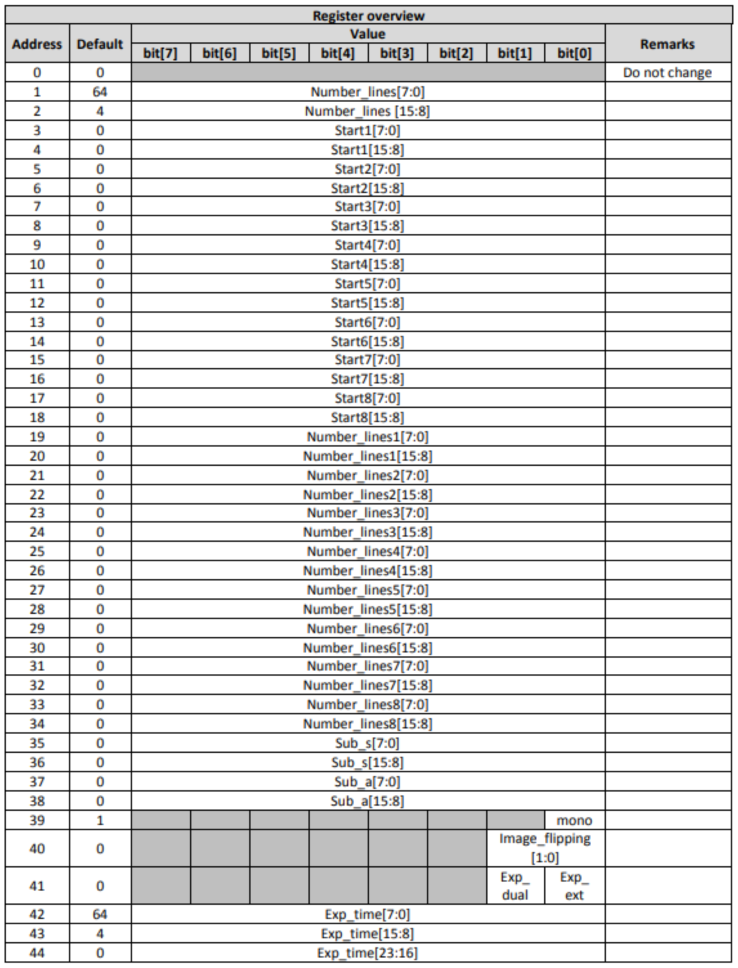

design_1_i/top_0_m_axis_TDATA[63:0] 0
design_1_i/cubedm..._mm2s_TDATA[15:0] 839
design_1_i/cubed..._axis_mm2s_TLAST 0
design_1_i/cubedm...axis_mm2s_TREADY 1
design_1_i/cubedm...axis_mm2s_TVALID 1
design_1_i/cubedma_top_0_mm2s_irq 0
design_1_i/cubedma_top_0_s2mm_irq 0
design_1_i/top_0_m_axis_TLAST 0
design_1_i/top_0_m_axis_TREADY 1
design_1_i/top_0_m_axis_TVALID 0

Figure 95: Waveform of Debug Core

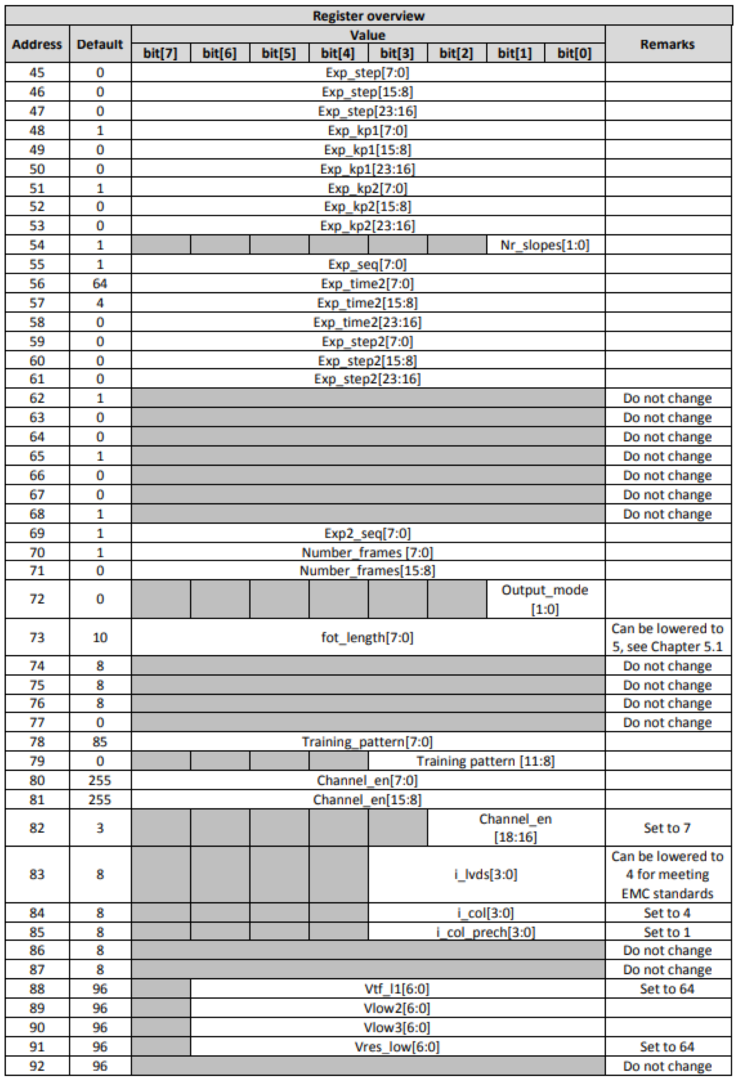
C CMV2000 Register Overview

This section shows an overview over some of the relevant registers that can be changed by the user. (For an full overview [4])



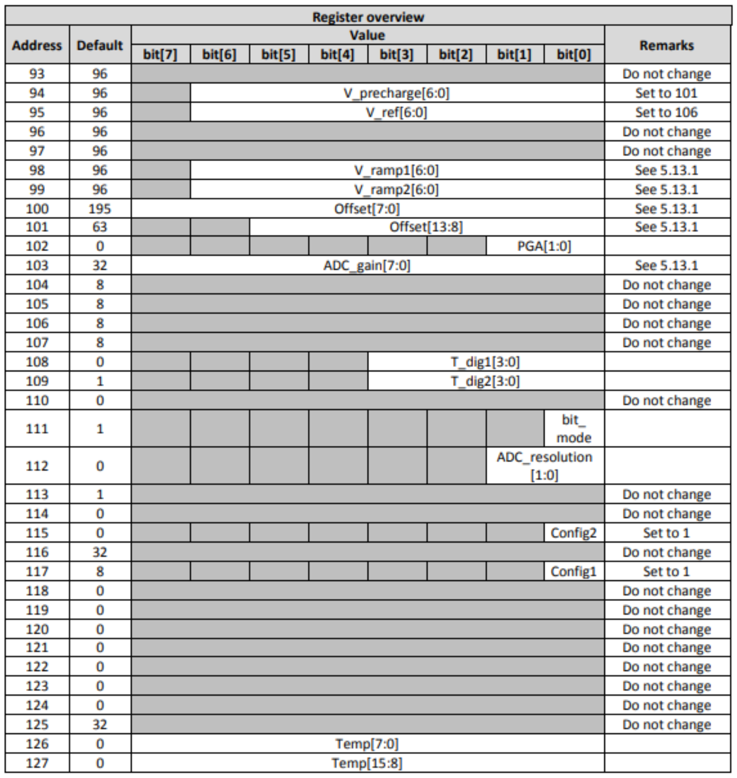
Register overview										
Address	Default	Value								Remarks
		bit[7]	bit[6]	bit[5]	bit[4]	bit[3]	bit[2]	bit[1]	bit[0]	
0	0	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
1	64	Number_lines[7:0]								
2	4	Number_lines [15:8]								
3	0	Start1[7:0]								
4	0	Start1[15:8]								
5	0	Start2[7:0]								
6	0	Start2[15:8]								
7	0	Start3[7:0]								
8	0	Start3[15:8]								
9	0	Start4[7:0]								
10	0	Start4[15:8]								
11	0	Start5[7:0]								
12	0	Start5[15:8]								
13	0	Start6[7:0]								
14	0	Start6[15:8]								
15	0	Start7[7:0]								
16	0	Start7[15:8]								
17	0	Start8[7:0]								
18	0	Start8[15:8]								
19	0	Number_lines1[7:0]								
20	0	Number_lines1[15:8]								
21	0	Number_lines2[7:0]								
22	0	Number_lines2[15:8]								
23	0	Number_lines3[7:0]								
24	0	Number_lines3[15:8]								
25	0	Number_lines4[7:0]								
26	0	Number_lines4[15:8]								
27	0	Number_lines5[7:0]								
28	0	Number_lines5[15:8]								
29	0	Number_lines6[7:0]								
30	0	Number_lines6[15:8]								
31	0	Number_lines7[7:0]								
32	0	Number_lines7[15:8]								
33	0	Number_lines8[7:0]								
34	0	Number_lines8[15:8]								
35	0	Sub_s[7:0]								
36	0	Sub_s[15:8]								
37	0	Sub_a[7:0]								
38	0	Sub_a[15:8]								
39	1	▨	▨	▨	▨	▨	▨	▨	mono	
40	0	▨	▨	▨	▨	▨	▨	Image_flipping [1:0]		
41	0	▨	▨	▨	▨	▨	▨	Exp_ dual	Exp_ ext	
42	64	Exp_time[7:0]								
43	4	Exp_time[15:8]								
44	0	Exp_time[23:16]								

Figure 96: Register overview [4]



Register overview										
Address	Default	Value								Remarks
		bit[7]	bit[6]	bit[5]	bit[4]	bit[3]	bit[2]	bit[1]	bit[0]	
45	0	Exp_step[7:0]								
46	0	Exp_step[15:8]								
47	0	Exp_step[23:16]								
48	1	Exp_kp1[7:0]								
49	0	Exp_kp1[15:8]								
50	0	Exp_kp1[23:16]								
51	1	Exp_kp2[7:0]								
52	0	Exp_kp2[15:8]								
53	0	Exp_kp2[23:16]								
54	1	▨	▨	▨	▨	▨	▨	Nr_slopes[1:0]		
55	1	Exp_seq[7:0]								
56	64	Exp_time2[7:0]								
57	4	Exp_time2[15:8]								
58	0	Exp_time2[23:16]								
59	0	Exp_step2[7:0]								
60	0	Exp_step2[15:8]								
61	0	Exp_step2[23:16]								
62	1	▨								Do not change
63	0	▨								Do not change
64	0	▨								Do not change
65	1	▨								Do not change
66	0	▨								Do not change
67	0	▨								Do not change
68	1	▨								Do not change
69	1	Exp2_seq[7:0]								
70	1	Number_frames [7:0]								
71	0	Number_frames[15:8]								
72	0	▨	▨	▨	▨	▨	▨	Output_mode [1:0]		
73	10	fot_length[7:0]								Can be lowered to 5, see Chapter 5.1
74	8	▨								Do not change
75	8	▨								Do not change
76	8	▨								Do not change
77	0	▨								Do not change
78	85	Training_pattern[7:0]								
79	0	▨	▨	▨	▨	Training pattern [11:8]				
80	255	Channel_en[7:0]								
81	255	Channel_en[15:8]								
82	3	▨	▨	▨	▨	▨	Channel_en [18:16]			Set to 7
83	8	▨	▨	▨	▨	i_lvds[3:0]				Can be lowered to 4 for meeting EMC standards
84	8	▨	▨	▨	▨	i_col[3:0]				Set to 4
85	8	▨	▨	▨	▨	i_col_prech[3:0]				Set to 1
86	8	▨								Do not change
87	8	▨								Do not change
88	96	▨	Vtf_l1[6:0]							Set to 64
89	96	▨	Vlow2[6:0]							
90	96	▨	Vlow3[6:0]							
91	96	▨	Vres_low[6:0]							Set to 64
92	96	▨								Do not change

Figure 97: Register overview [4]



Register overview										
Address	Default	Value								Remarks
		bit[7]	bit[6]	bit[5]	bit[4]	bit[3]	bit[2]	bit[1]	bit[0]	
93	96	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
94	96	▨	V_precharge[6:0]							Set to 101
95	96	▨	V_ref[6:0]							Set to 106
96	96	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
97	96	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
98	96	▨	V_ramp1[6:0]							See 5.13.1
99	96	▨	V_ramp2[6:0]							See 5.13.1
100	195	Offset[7:0]								See 5.13.1
101	63	▨	▨	Offset[13:8]						See 5.13.1
102	0	▨	▨	▨	▨	▨	▨	PGA[1:0]		
103	32	ADC_gain[7:0]								See 5.13.1
104	8	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
105	8	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
106	8	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
107	8	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
108	0	▨	▨	▨	▨	T_dig1[3:0]				
109	1	▨	▨	▨	▨	T_dig2[3:0]				
110	0	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
111	1	▨	▨	▨	▨	▨	▨	▨	bit_ mode	
112	0	▨	▨	▨	▨	▨	▨	ADC_resolution [1:0]		
113	1	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
114	0	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
115	0	▨	▨	▨	▨	▨	▨	▨	Config2	Set to 1
116	32	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
117	8	▨	▨	▨	▨	▨	▨	▨	Config1	Set to 1
118	0	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
119	0	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
120	0	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
121	0	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
122	0	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
123	0	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
124	0	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
125	32	▨	▨	▨	▨	▨	▨	▨	▨	Do not change
126	0	Temp[7:0]								
127	0	Temp[15:8]								

Figure 98: Register overview [4]

D Cube DMA register map

Field	**Description**	**Bits**
Control register (0x00)		
Start	Initiates the transfer	0
Blockwise mode	Cube is read in blocks	2
Planewise mode	Cube is read in planes	3
Error IRQ enable	IRQ trigger on error	4
Completion IRQ enable	IRQ trigger on completion	5
Status register (0x04)		
Transfer done	Indicates transfer completed	0
Error code	Indicate error conditions	3:1
Error IRQ flag	IRQ trigger due to error	4
Completion IRQ flag	IRQ trigger due to completion	5
Base address register (0x08)		
Base Address	Address of first component in HSI cube	31:0
Cube dimension register (0x0C)		
Width	Widht of HSI cube	11:0
Height	Height of HSI cube	23:12
Depth	Depth of HSI cube	31:24
Row size register (0x14)		
Row size	Total components in one row of the cube	19:0

Table 19: Cube DMA register map MM2S. [6]

Field	**Description**	**Bits**
Control register (0x20)		
Start	Initiates the transfer	0
Error IRQ enable	IRQ trigger on error	4
Completion IRQ enable	IRQ trigger on completion	5
Status register (0x24)		
Transfer done	Indicates transfer completed	0
Error code	Indicate error conditions	3:1
Error IRQ flag	IRQ trigger due to error	4
Completion IRQ flag	IRQ trigger due to completion	5
Base address register (0x28)		
Base Address	Address of first component in HSI cube	31:0

Table 20: Cube DMA register map S2MM. [6]

E AXI Dependencies

E.1 Read transaction dependencies

1. Master may assert ARVALID before slave asserts ARREADY.
2. Slave may wait for master to assert ARVALID before asserting ARREADY.
3. Slave is allowed to assert ARREADY before master asserts ARVALID
4. Slave cannot assert RVALID before both ARVALID and ARREADY has been asserted.
5. Slave can assert RVALID before master asserts RREADY.
6. Master can wait for slave to assert RVALID before asserting RREADY.
7. Master is allowed to assert RREADY before RVALID has been asserted.

E.2 Write transaction dependencies

1. Master is allowed to assert AWVALID or WVALID before slave asserts AWREADY or WREADY.
2. Slave may wait for master to assert AWVALID or WVALID, or both before it asserts AWREADY.
3. Slave is allowed to assert AWREADY before AWVALID or WVALID is asserted.
4. Slave may wait for AWVALID or WVALID before asserting WREADY.
5. Slave is allowed to assert WREADY before AWVALID or WVALID, or both is asserted.
6. Slave is not allowed to assert BVALID before both WVALID and WREADY is asserted by the master.
7. Slave may assert BVALID before master asserts BREADY.
8. Master can wait for slave to assert BVALID before asserting BREADY.
9. Master may assert BREADY before slave assert BVALID.

F LVDS Receiver Interface

F.1 Deserializer

F.1.1 Design File

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library UNISIM;
use UNISIM.VComponents.all;

entity lvds_deserializer is
    Generic(
        PIXEL_BIT_WIDTH : positive := 10;
        NUM_LVDS_PAIRS  : positive := 16;
        BIT_CLK_DELAY_TAP_VALUE : integer := 0
    );
    Port (
        refclk       : in std_logic;
        rst          : in std_logic;
        enable       : in std_logic;
        clk_p_n      : in std_logic_vector(1 downto 0);
        data_p_n     : in std_logic_vector(NUM_LVDS_PAIRS*2-1 downto 0);
        ctrl_p_n     : in std_logic_vector(1 downto 0);
        training_data : in std_logic_vector(11 downto 0);
        sensor_idle  : in std_logic;
        valid_out    : out std_logic;
        in_sync      : out std_logic;

        --IDELAYE2
        CE           : in std_logic_vector(NUM_LVDS_PAIRS*2+1  downto 0);
        INC          : in std_logic_vector(NUM_LVDS_PAIRS*2+1 downto 0);

        --ISERDESE2
        Q_out        : out std_logic_vector(PIXEL_BIT_WIDTH*NUM_LVDS_PAIRS-1 downto 0);
        CLKDIV_PROBE : out std_logic
    );
end lvds_deserializer;

architecture Behavioral of lvds_deserializer is
--IDELAYE Signals
signal d_e_o, dly_e_o : std_logic_vector(NUM_LVDS_PAIRS*2+1 downto 0);
signal clk, dly_clk : std_logic;

--ISERDESE Signals
signal BITSLIP : std_logic_vector(NUM_LVDS_PAIRS*2+1  downto 0) := (others => '0');
signal Q_inv : std_logic_vector(NUM_LVDS_PAIRS*(PIXEL_BIT_WIDTH/2)+PIXEL_BIT_WIDTH/2 - 1 downto 0);
signal Q : std_logic_vector(0 to PIXEL_BIT_WIDTH+PIXEL_BIT_WIDTH*NUM_LVDS_PAIRS-1);
signal RXCLK,ISE_CE : std_logic;
signal CLK_even, CLKB_even, CLK_odd, CLKB_odd, CLKDIV, RDY:  std_logic;

--Control Channel
signal control_word : std_logic_vector(PIXEL_BIT_WIDTH-1 downto 0);
signal ctrl_e_o, ctrl_dly_e_o : std_logic_vector(1 downto 0);
signal DVAL, LVAL, FVAL, SLOT, ROW, FOT, INTE1, INTE2 : std_logic;


--State machine
TYPE State_type IS (idle,training_ctrl_ch,training, transfer);  -- Define the states
SIGNAL state : State_Type;    -- Create a signal that uses
signal trained : std_logic;
signal data_ctrl_p_n : std_logic_vector(2+NUM_LVDS_PAIRS*2-1 downto 0);

begin
data_ctrl_p_n(NUM_LVDS_PAIRS*2-1 downto 0) <= data_p_n; -- assigning first NUM_LVDS_PAIRS*2-1 downto 0 bits
      to data channels
data_ctrl_p_n(2+NUM_LVDS_PAIRS*2-1 downto 2+NUM_LVDS_PAIRS*2-2) <= ctrl_p_n; --assigning last two bits to
      ctrl channel
control_word <= Q(PIXEL_BIT_WIDTH*NUM_LVDS_PAIRS to PIXEL_BIT_WIDTH*NUM_LVDS_PAIRS+PIXEL_BIT_WIDTH-1); --
      Fetching control bits from result.
Q_out <= Q(0 to PIXEL_BIT_WIDTH*NUM_LVDS_PAIRS-1); --Setting data bits on Q_out
ISE_CE <= '1' when rst = '0' else '0'; --enable iserdes at once rst = '0'
CLKDIV_PROBE <= CLKDIV; --Output a version of CLKDIV


--Control Channel
DVAL <= control_word(0);
LVAL <= control_word(1);
FVAL <= control_word(2);
SLOT <= control_word(3);
ROW  <= control_word(4);
FOT  <= control_word(5);
INTE1 <= control_word(6);
INTE2 <= control_word(7);
```

```

--Synchronise data channels
process(CLKDIV, rst)
        variable counter : integer := 0;
        variable flag : std_logic;
        variable lvds_channel : integer range 0 to 16 := 0;
    begin
        if(rst = '1') then
            counter := 0;
            flag := '0';
            in_sync <= '0';
            state <= idle;
            lvds_channel := 0;
        elsif(rising_edge(CLKDIV)) then
            case state is
                when idle =>
                    if(enable = '1') then
                        state <= training_ctrl_ch;
                    end if;
                when training_ctrl_ch => --train control channel
                    --check if ctr_channel outputs training data.
                    if(to_integer(unsigned(control_word)) = to_integer(unsigned(training_data))) then
                        state <= training; --training completed
                    else
                        -- wrong output, do a bitslip.
                        if(counter = 2) then
                            BITSLIP(2*NUM_LVDS_PAIRS) <= '0';
                            BITSLIP(2*NUM_LVDS_PAIRS+1) <= '0';
                        end if;
                        if(counter = 4) then
                            if(flag = '0') then
                                    BITSLIP(2*NUM_LVDS_PAIRS) <= '1';
                                    counter := 0;
                                    flag := '1';
                            elsif(flag = '1') then
                                    BITSLIP(2*NUM_LVDS_PAIRS+1) <= '1';
                                    counter := 0;
                                    flag := '0';
                            end if;
                        end if;
                        counter := counter + 1;
                    end if;
                --train data channels.
                when training =>
                    --reset bitslip bits
                    if(counter = 2) then
                        BITSLIP(2*lvds_channel) <= '0';
                        BITSLIP(2*lvds_channel+1) <= '0';
                    end if;
                    --check if output is correct for this channel
                    if(to_integer(unsigned(Q(PIXEL_BIT_WIDTH*lvds_channel to PIXEL_BIT_WIDTH*lvds_channel +
                         PIXEL_BIT_WIDTH-1))) = to_integer(unsigned(training_data)) ) then
                        lvds_channel := lvds_channel + 1; --output correct, procees with next channel
                        if(lvds_channel = NUM_LVDS_PAIRS) then
                            state <= transfer;
                            counter := 0;
                            lvds_channel := 0;
                            in_sync <= '1';
                        end if;
                    else
                        if(counter = 4) then
                            if(flag = '0') then
                                BITSLIP(2*lvds_channel) <= '1';
                                counter := 0;
                                flag := '1';
                            elsif(flag = '1') then
                                BITSLIP(2*lvds_channel+1) <= '1';
                                counter := 0;
                                flag := '0';
                            end if;
                        end if;
                        counter := counter + 1;
                    end if;
                when transfer =>
                    --if DVAL is high, the module produces valid output
                    if(DVAL = '1') then
                        valid_out <= '1';
                    elsif(DVAL = '0') then
                        valid_out <= '0';
                     end if;
            end case;
        end if;
end process;


--Declaration of buffers
BUFR_inst : BUFR
generic map (
  BUFR_DIVIDE => integer'image(PIXEL_BIT_WIDTH/2),   -- Values: "BYPASS, 1, 2, 3, 4, 5, 6, 7, 8"
  SIM_DEVICE => "7SERIES"  -- Must be set to "7SERIES"
)
```

```
port map (
  O => CLKDIV,     -- 1-bit output: Clock output port
  CE => '1',   -- 1-bit input: Active high, clock enable (Divided modes only)
  CLR => '0', -- 1-bit input: Active high, asynchronous clear (Divided modes only)
  I => dly_clk      -- 1-bit input: Clock buffer input driven by an IBUF, MMCM or local interconnect
);

BUFIO_inst : BUFIO
   port map (
      O => RXCLK, -- 1-bit output: Clock output (connect to I/O clock loads).
      I => dly_clk  -- 1-bit input: Clock input (connect to an IBUF or BUFMR).
   );

IBUFDS_inst : IBUFDS
generic map (
   DIFF_TERM => FALSE, -- Differential Termination
   IBUF_LOW_PWR => TRUE, -- Low power (TRUE) vs. performance (FALSE) setting for referenced I/O standards
   IOSTANDARD => "LVDS_25")
port map (
   O => clk,  -- Buffer output
   I => clk_p_n(0),  -- Diff_p buffer input (connect directly to top-level port)
   IB => clk_p_n(1) -- Diff_n buffer input (connect directly to top-level port)
);

IBUFDS_DIFF_gen: for i in 0 to NUM_LVDS_PAIRS generate
IBUFDS_DIFF_OUT_inst : IBUFDS_DIFF_OUT
    generic map (
        DIFF_TERM => FALSE, -- Differential Termination
        IBUF_LOW_PWR => TRUE, -- Low power (TRUE) vs. performance (FALSE) setting for referenced I/O
              standards
        IOSTANDARD => "LVDS_25") -- Specify the input I/O standard
    port map (
        O => d_e_o(2*i),      -- Buffer diff_p output
        OB =>  d_e_o(2*i+1),     -- Buffer diff_n output
        I => data_ctrl_p_n(2*i), -- Diff_p buffer input (connect directly to top-level port)
        IB => data_ctrl_p_n(2*i+1) -- Diff_n buffer input (connect directly to top-level port)
    );
end generate IBUFDS_DIFF_gen;


--Instantiation of IDELAYCTRL and IDELAYE primitives
IDELAYCTRL_inst : IDELAYCTRL
   port map (
      RDY => RDY,       -- 1-bit output: Ready output
      REFCLK => refclk, -- 1-bit input: Reference clock input
      RST => RST        -- 1-bit input: Active high reset input
   );


delay_gen : for i in 0 to 2*NUM_LVDS_PAIRS+1 generate
   -- IDELAYE2: Input Fixed or Variable Delay Element
   --           Virtex-7
   -- Xilinx HDL Language Template, version 2017.4
   IDELAYE2_inst : IDELAYE2
   generic map (
      CINVCTRL_SEL => "FALSE",          -- Enable dynamic clock inversion (FALSE, TRUE)
      DELAY_SRC => "IDATAIN",           -- Delay input (IDATAIN, DATAIN)
      HIGH_PERFORMANCE_MODE => "FALSE", -- Reduced jitter ("TRUE"), Reduced power ("FALSE")
      IDELAY_TYPE => "VARIABLE",           -- FIXED, VARIABLE, VAR_LOAD, VAR_LOAD_PIPE
      IDELAY_VALUE => 0,                -- Input delay tap setting (0-31)
      PIPE_SEL => "FALSE",              -- Select pipelined mode, FALSE, TRUE
      REFCLK_FREQUENCY => 200.0,        -- IDELAYCTRL clock input frequency in MHz (190.0-210.0,
            290.0-310.0).
      SIGNAL_PATTERN => "DATA"          -- DATA, CLOCK input signal
   )
   port map (
      CNTVALUEOUT => open, -- 5-bit output: Counter value output
      DATAOUT => dly_e_o(i),         -- 1-bit output: Delayed data output
      C => RXCLK,                     -- 1-bit input: Clock input
      CE => CE(i),                   -- 1-bit input: Active high enable increment/decrement input
      CINVCTRL => '0',       -- 1-bit input: Dynamic clock inversion input
      CNTVALUEIN => b"00000",   -- 5-bit input: Counter value input
      DATAIN => '0',           -- 1-bit input: Internal delay data input
      IDATAIN => d_e_o(i),         -- 1-bit input: Data input from the I/O
      INC => INC(i),                 -- 1-bit input: Increment / Decrement tap delay input
      LD => '0',                   -- 1-bit input: Load IDELAY_VALUE input
      LDPIPEEN => '0',       -- 1-bit input: Enable PIPELINE register to load data input
      REGRST => RST               -- 1-bit input: Active-high reset tap-delay input
   );
   -- End of IDELAYE2_inst instantiation
end generate delay_gen;

clock_delay_inst : IDELAYE2
   generic map (
      CINVCTRL_SEL => "FALSE",          -- Enable dynamic clock inversion (FALSE, TRUE)
      DELAY_SRC => "IDATAIN",           -- Delay input (IDATAIN, DATAIN)
      HIGH_PERFORMANCE_MODE => "FALSE", -- Reduced jitter ("TRUE"), Reduced power ("FALSE")
      IDELAY_TYPE => "FIXED",           -- FIXED, VARIABLE, VAR_LOAD, VAR_LOAD_PIPE
      IDELAY_VALUE => BIT_CLK_DELAY_TAP_VALUE, -- Input delay tap setting (0-31)
      PIPE_SEL => "FALSE",              -- Select pipelined mode, FALSE, TRUE
      REFCLK_FREQUENCY => 200.0,        -- IDELAYCTRL clock input frequency in MHz (190.0-210.0,
            290.0-310.0).
      SIGNAL_PATTERN => "CLOCK"          -- DATA, CLOCK input signal
   )
   port map (
      CNTVALUEOUT => open, -- 5-bit output: Counter value output
      DATAOUT => dly_clk,          -- 1-bit output: Delayed data output
      C => '0',                      -- 1-bit input: Clock input
      CE => '0',                    -- 1-bit input: Active high enable increment/decrement input
      CINVCTRL => '0',        -- 1-bit input: Dynamic clock inversion input
      CNTVALUEIN => B"00000",   -- 5-bit input: Counter value input
      DATAIN => '0',            -- 1-bit input: Internal delay data input
      IDATAIN => clk,          -- 1-bit input: Data input from the I/O
      INC => '0',                  -- 1-bit input: Increment / Decrement tap delay input
      LD => '0',                    -- 1-bit input: Load IDELAY_VALUE input
      LDPIPEEN => '0',        -- 1-bit input: Enable PIPELINE register to load data input
      REGRST => '0'                -- 1-bit input: Active-high reset tap-delay input
   );


--Map the ISERDESE2 clocks
CLK_even <= RXCLK;
CLKB_odd <= RXCLK;
CLK_odd <= not RXCLK;
CLKB_even <= not RXCLK;

--Instantiation of the ISERDESE primitives
ISERDES_gen: for i in 0 to NUM_LVDS_PAIRS generate
twelve_gen: if(PIXEL_BIT_WIDTH = 12) generate --12-bit
    ISERDESE2_even : ISERDESE2 --even bits
        generic map (
           DATA_RATE => "SDR",           -- DDR, SDR
           DATA_WIDTH => PIXEL_BIT_WIDTH/2,              -- Parallel data width (2-8,10,14)
           DYN_CLKDIV_INV_EN => "FALSE", -- Enable DYNCLKDIVINVSEL inversion (FALSE, TRUE)
           DYN_CLK_INV_EN => "FALSE",    -- Enable DYNCLKINVSEL inversion (FALSE, TRUE)
           -- INIT_Q1 - INIT_Q4: Initial value on the Q outputs (0/1)
           INIT_Q1 => '0',
           INIT_Q2 => '0',
           INIT_Q3 => '0',
           INIT_Q4 => '0',
           INTERFACE_TYPE => "NETWORKING",   -- MEMORY, MEMORY_DDR3, MEMORY_QDR, NETWORKING, OVERSAMPLE
           IOBDELAY => "BOTH",           -- NONE, BOTH, IBUF, IFD
           NUM_CE => 1,                  -- Number of clock enables (1,2)
           OFB_USED => "FALSE",          -- Select OFB path (FALSE, TRUE)
           SERDES_MODE => "MASTER",      -- MASTER, SLAVE
           -- SRVAL_Q1 - SRVAL_Q4: Q output values when SR is used (0/1)
           SRVAL_Q1 => '0',
           SRVAL_Q2 => '0',
           SRVAL_Q3 => '0',
           SRVAL_Q4 => '0'
        )
        port map (
           O => open,                       -- 1-bit output: Combinatorial output
           -- Q1 - Q8: 1-bit (each) output: Registered data outputs
           Q1 => Q(PIXEL_BIT_WIDTH*i),
           Q2 => Q(PIXEL_BIT_WIDTH*i +2),
           Q3 => Q(PIXEL_BIT_WIDTH*i +4),
           Q4 => Q(PIXEL_BIT_WIDTH*i +6),
           Q5 => Q(PIXEL_BIT_WIDTH*i +8),
           Q6 => Q(PIXEL_BIT_WIDTH*i +10),
           Q7 => open,
           Q8 => open,
           -- SHIFTOUT1, SHIFTOUT2: 1-bit (each) output: Data width expansion output ports
           SHIFTOUT1 => open,
           SHIFTOUT2 => open,
           BITSLIP => BITSLIP(2*i),        -- 1-bit input: The BITSLIP pin performs a Bitslip operation
                 synchronous to
                                          -- CLKDIV when asserted (active High). Subsequently, the data seen
                                                on the
                                          -- Q1 to Q8 output ports will shift, as in a barrel-shifter
                                                operation, one
                                          -- position every time Bitslip is invoked (DDR operation is
                                                different from
                                          -- SDR).
                                          -- CE1, CE2: 1-bit (each) input: Data register clock enable inputs
           CE1 => ISE_CE,
           CE2 => ISE_CE,
           CLKDIVP => '0',           -- 1-bit input: TBD
           -- Clocks: 1-bit (each) input: ISERDESE2 clock input ports
           CLK => CLK_even,                   -- 1-bit input: High-speed clock
           CLKB => CLKB_even,                 -- 1-bit input: High-speed secondary clock
           CLKDIV => CLKDIV,             -- 1-bit input: Divided clock
           OCLK => '0',                 -- 1-bit input: High speed output clock used when INTERFACE_TYPE="
                 MEMORY"
           -- Dynamic Clock Inversions: 1-bit (each) input: Dynamic clock inversion pins to switch clock
                 polarity
           DYNCLKDIVSEL => '0', -- 1-bit input: Dynamic CLKDIV inversion
           DYNCLKSEL => '0',       -- 1-bit input: Dynamic CLK/CLKB inversion
           -- Input Data: 1-bit (each) input: ISERDESE2 data input ports
           D => '0',                       -- 1-bit input: Data input
           DDLY => dly_e_o(2*i),                 -- 1-bit input: Serial data from IDELAYE2
```

```
        OFB => '0',                    -- 1-bit input: Data feedback from OSERDESE2
        OCLKB => '0',                -- 1-bit input: High speed negative edge output clock
        RST => RST,                    -- 1-bit input: Active high asynchronous reset
        -- SHIFTIN1, SHIFTIN2: 1-bit (each) input: Data width expansion input ports
        SHIFTIN1 => '0',
        SHIFTIN2 => '0'
     );

     ISERDESE2_odd : ISERDESE2 --odd bits
         generic map (
            DATA_RATE => "SDR",            -- DDR, SDR
            DATA_WIDTH => PIXEL_BIT_WIDTH/2,               -- Parallel data width (2-8,10,14)
            DYN_CLKDIV_INV_EN => "FALSE", -- Enable DYNCLKDIVINVSEL inversion (FALSE, TRUE)
            DYN_CLK_INV_EN => "FALSE",    -- Enable DYNCLKINVSEL inversion (FALSE, TRUE)
            -- INIT_Q1 - INIT_Q4: Initial value on the Q outputs (0/1)
            INIT_Q1 => '0',
            INIT_Q2 => '0',
            INIT_Q3 => '0',
            INIT_Q4 => '0',
            INTERFACE_TYPE => "NETWORKING",   -- MEMORY, MEMORY_DDR3, MEMORY_QDR, NETWORKING, OVERSAMPLE
            IOBDELAY => "BOTH",            -- NONE, BOTH, IBUF, IFD
            NUM_CE => 1,                   -- Number of clock enables (1,2)
            OFB_USED => "FALSE",           -- Select OFB path (FALSE, TRUE)
            SERDES_MODE => "MASTER",       -- MASTER, SLAVE
            -- SRVAL_Q1 - SRVAL_Q4: Q output values when SR is used (0/1)
            SRVAL_Q1 => '0',
            SRVAL_Q2 => '0',
            SRVAL_Q3 => '0',
            SRVAL_Q4 => '0'
         )
         port map (
            O => open,                       -- 1-bit output: Combinatorial output
            -- Q1 - Q8: 1-bit (each) output: Registered data outputs
            Q1 => Q_inv(i*(PIXEL_BIT_WIDTH/2)),
            Q2 => Q_inv(i*(PIXEL_BIT_WIDTH/2) +1),
            Q3 => Q_inv(i*(PIXEL_BIT_WIDTH/2) +2),
            Q4 => Q_inv(i*(PIXEL_BIT_WIDTH/2) +3),
            Q5 => Q_inv(i*(PIXEL_BIT_WIDTH/2) +4),
            Q6 => Q_inv(i*(PIXEL_BIT_WIDTH/2) +5),
            Q7 => open,
            Q8 => open,
            -- SHIFTOUT1, SHIFTOUT2: 1-bit (each) output: Data width expansion output ports
            SHIFTOUT1 => open,
            SHIFTOUT2 => open,
            BITSLIP => BITSLIP(2*i+1),           -- 1-bit input: The BITSLIP pin performs a Bitslip
                  operation synchronous to
                                         -- CLKDIV when asserted (active High). Subsequently, the data
                                               seen on the
                                         -- Q1 to Q8 output ports will shift, as in a barrel-shifter
                                               operation, one
                                         -- position every time Bitslip is invoked (DDR operation is
                                               different from
                                         -- SDR).

            -- CE1, CE2: 1-bit (each) input: Data register clock enable inputs
            CE1 => ISE_CE,
            CE2 => ISE_CE,
            CLKDIVP => '0',           -- 1-bit input: TBD
            -- Clocks: 1-bit (each) input: ISERDESE2 clock input ports
            CLK => CLK_odd,                   -- 1-bit input: High-speed clock
            CLKB => CLKB_odd,                 -- 1-bit input: High-speed secondary clock
            CLKDIV => CLKDIV,             -- 1-bit input: Divided clock
            OCLK => '0',                 -- 1-bit input: High speed output clock used when INTERFACE_TYPE
                  ="MEMORY"
            -- Dynamic Clock Inversions: 1-bit (each) input: Dynamic clock inversion pins to switch clock
                   polarity
            DYNCLKDIVSEL => '0', -- 1-bit input: Dynamic CLKDIV inversion
            DYNCLKSEL => '0',       -- 1-bit input: Dynamic CLK/CLKB inversion
            -- Input Data: 1-bit (each) input: ISERDESE2 data input ports
            D => '0',                       -- 1-bit input: Data input
            DDLY => dly_e_o(2*i+1),                 -- 1-bit input: Serial data from IDELAYE2
            OFB => '0',                    -- 1-bit input: Data feedback from OSERDESE2
            OCLKB => '0',                -- 1-bit input: High speed negative edge output clock
            RST => RST,                    -- 1-bit input: Active high asynchronous reset
            -- SHIFTIN1, SHIFTIN2: 1-bit (each) input: Data width expansion input ports
            SHIFTIN1 => '0',
            SHIFTIN2 => '0'
         );
 end generate twelve_gen;

 ten_gen: if(PIXEL_BIT_WIDTH = 10) generate --10-bit
     ISERDESE2_even : ISERDESE2 -- even bits
         generic map (
            DATA_RATE => "SDR",            -- DDR, SDR
            DATA_WIDTH => PIXEL_BIT_WIDTH/2,               -- Parallel data width (2-8,10,14)
            DYN_CLKDIV_INV_EN => "FALSE", -- Enable DYNCLKDIVINVSEL inversion (FALSE, TRUE)
            DYN_CLK_INV_EN => "FALSE",    -- Enable DYNCLKINVSEL inversion (FALSE, TRUE)
            -- INIT_Q1 - INIT_Q4: Initial value on the Q outputs (0/1)
            INIT_Q1 => '0',
            INIT_Q2 => '0',
            INIT_Q3 => '0',
            INIT_Q4 => '0',
            INTERFACE_TYPE => "NETWORKING",   -- MEMORY, MEMORY_DDR3, MEMORY_QDR, NETWORKING, OVERSAMPLE
            IOBDELAY => "BOTH",           -- NONE, BOTH, IBUF, IFD
            NUM_CE => 1,                  -- Number of clock enables (1,2)
            OFB_USED => "FALSE",          -- Select OFB path (FALSE, TRUE)
            SERDES_MODE => "MASTER",      -- MASTER, SLAVE
            -- SRVAL_Q1 - SRVAL_Q4: Q output values when SR is used (0/1)
            SRVAL_Q1 => '0',
            SRVAL_Q2 => '0',
            SRVAL_Q3 => '0',
            SRVAL_Q4 => '0'
         )
         port map (
            O => open,                       -- 1-bit output: Combinatorial output
            -- Q1 - Q8: 1-bit (each) output: Registered data outputs
            Q1 => Q(PIXEL_BIT_WIDTH*i),
            Q2 => Q(PIXEL_BIT_WIDTH*i +2),
            Q3 => Q(PIXEL_BIT_WIDTH*i +4),
            Q4 => Q(PIXEL_BIT_WIDTH*i +6),
            Q5 => Q(PIXEL_BIT_WIDTH*i +8),
            Q6 => open,
            Q7 => open,
            Q8 => open,
            -- SHIFTOUT1, SHIFTOUT2: 1-bit (each) output: Data width expansion output ports
            SHIFTOUT1 => open,
            SHIFTOUT2 => open,
            BITSLIP => BITSLIP(2*i),        -- 1-bit input: The BITSLIP pin performs a Bitslip operation
                 synchronous to
                                         -- CLKDIV when asserted (active High). Subsequently, the data
                                              seen on the
                                         -- Q1 to Q8 output ports will shift, as in a barrel-shifter
                                              operation, one
                                         -- position every time Bitslip is invoked (DDR operation is
                                              different from
                                         -- SDR).
                                         -- CE1, CE2: 1-bit (each) input: Data register clock enable
                                              inputs
            CE1 => ISE_CE,
            CE2 => ISE_CE,
            CLKDIVP => '0',           -- 1-bit input: TBD
            -- Clocks: 1-bit (each) input: ISERDESE2 clock input ports
            CLK => CLK_even,                   -- 1-bit input: High-speed clock
            CLKB => CLKB_even,                 -- 1-bit input: High-speed secondary clock
            CLKDIV => CLKDIV,            -- 1-bit input: Divided clock
            OCLK => '0',                -- 1-bit input: High speed output clock used when INTERFACE_TYPE
                 ="MEMORY"
            -- Dynamic Clock Inversions: 1-bit (each) input: Dynamic clock inversion pins to switch clock
                  polarity
            DYNCLKDIVSEL => '0', -- 1-bit input: Dynamic CLKDIV inversion
            DYNCLKSEL => '0',       -- 1-bit input: Dynamic CLK/CLKB inversion
            -- Input Data: 1-bit (each) input: ISERDESE2 data input ports
            D => '0',                      -- 1-bit input: Data input
            DDLY => dly_e_o(2*i),                -- 1-bit input: Serial data from IDELAYE2
            OFB => '0',                  -- 1-bit input: Data feedback from OSERDESE2
            OCLKB => '0',               -- 1-bit input: High speed negative edge output clock
            RST => RST,                  -- 1-bit input: Active high asynchronous reset
            -- SHIFTIN1, SHIFTIN2: 1-bit (each) input: Data width expansion input ports
            SHIFTIN1 => '0',
            SHIFTIN2 => '0'
         );

         ISERDESE2_odd : ISERDESE2 --odd bits
             generic map (
                DATA_RATE => "SDR",           -- DDR, SDR
                DATA_WIDTH => PIXEL_BIT_WIDTH/2,              -- Parallel data width (2-8,10,14)
                DYN_CLKDIV_INV_EN => "FALSE", -- Enable DYNCLKDIVINVSEL inversion (FALSE, TRUE)
                DYN_CLK_INV_EN => "FALSE",    -- Enable DYNCLKINVSEL inversion (FALSE, TRUE)
                -- INIT_Q1 - INIT_Q4: Initial value on the Q outputs (0/1)
                INIT_Q1 => '0',
                INIT_Q2 => '0',
                INIT_Q3 => '0',
                INIT_Q4 => '0',
                INTERFACE_TYPE => "NETWORKING",   -- MEMORY, MEMORY_DDR3, MEMORY_QDR, NETWORKING,
                     OVERSAMPLE
                IOBDELAY => "BOTH",           -- NONE, BOTH, IBUF, IFD
                NUM_CE => 1,                  -- Number of clock enables (1,2)
                OFB_USED => "FALSE",          -- Select OFB path (FALSE, TRUE)
                SERDES_MODE => "MASTER",      -- MASTER, SLAVE
                -- SRVAL_Q1 - SRVAL_Q4: Q output values when SR is used (0/1)
                SRVAL_Q1 => '0',
                SRVAL_Q2 => '0',
                SRVAL_Q3 => '0',
                SRVAL_Q4 => '0'
             )
             port map (
                O => open,                       -- 1-bit output: Combinatorial output
                -- Q1 - Q8: 1-bit (each) output: Registered data outputs
                Q1 => Q_inv(i*(PIXEL_BIT_WIDTH/2)),
                Q2 => Q_inv(i*(PIXEL_BIT_WIDTH/2) +1),
                Q3 => Q_inv(i*(PIXEL_BIT_WIDTH/2) +2),
                Q4 => Q_inv(i*(PIXEL_BIT_WIDTH/2) +3),
                   Q5 => Q_inv(i*(PIXEL_BIT_WIDTH/2) +4),
                   Q6 => open,
                   Q7 => open,
                   Q8 => open,
                   -- SHIFTOUT1, SHIFTOUT2: 1-bit (each) output: Data width expansion output ports
                   SHIFTOUT1 => open,
                   SHIFTOUT2 => open,
                   BITSLIP => BITSLIP(2*i+1),           -- 1-bit input: The BITSLIP pin performs a Bitslip
                         operation synchronous to
                                             -- CLKDIV when asserted (active High). Subsequently, the
                                                   data seen on the
                                             -- Q1 to Q8 output ports will shift, as in a barrel-shifter
                                                    operation, one
                                             -- position every time Bitslip is invoked (DDR operation is
                                                    different from
                                             -- SDR).

                   -- CE1, CE2: 1-bit (each) input: Data register clock enable inputs
                   CE1 => ISE_CE,
                   CE2 => ISE_CE,
                   CLKDIVP => '0',           -- 1-bit input: TBD
                   -- Clocks: 1-bit (each) input: ISERDESE2 clock input ports
                   CLK => CLK_odd,                    -- 1-bit input: High-speed clock
                   CLKB => CLKB_odd,                  -- 1-bit input: High-speed secondary clock
                   CLKDIV => CLKDIV,             -- 1-bit input: Divided clock
                   OCLK => '0',                 -- 1-bit input: High speed output clock used when
                         INTERFACE_TYPE="MEMORY"
                   -- Dynamic Clock Inversions: 1-bit (each) input: Dynamic clock inversion pins to switch
                         clock polarity
                   DYNCLKDIVSEL => '0', -- 1-bit input: Dynamic CLKDIV inversion
                   DYNCLKSEL => '0',       -- 1-bit input: Dynamic CLK/CLKB inversion
                   -- Input Data: 1-bit (each) input: ISERDESE2 data input ports
                   D => '0',                       -- 1-bit input: Data input
                   DDLY => dly_e_o(2*i+1),                 -- 1-bit input: Serial data from IDELAYE2
                   OFB => '0',                   -- 1-bit input: Data feedback from OSERDESE2
                   OCLKB => '0',               -- 1-bit input: High speed negative edge output clock
                   RST => RST,                   -- 1-bit input: Active high asynchronous reset
                   -- SHIFTIN1, SHIFTIN2: 1-bit (each) input: Data width expansion input ports
                   SHIFTIN1 => '0',
                   SHIFTIN2 => '0'
                );
        end generate ten_gen;
end generate ISERDES_gen;

-- Invert output from odd-bits ISERDESE2
inv_gen: for i in 0 to (NUM_LVDS_PAIRS*(PIXEL_BIT_WIDTH/2)+(PIXEL_BIT_WIDTH/2)-1) generate
        Q(i*2+1) <= not Q_inv(i);
end generate inv_gen;
end Behavioral;
```

F.1.2 Test Bench

```
`timescale 1ns / 1ps

module lvds_tb;
parameter PIXEL_BIT_WIDTH = 12;
parameter NUM_LVDS_PAIRS = 4;
parameter BIT_CLK_DELAY_TAP_VALUE = 0;
parameter PERIOD = 5;
reg [NUM_LVDS_PAIRS*2+1:0]CE;
reg refclk;
reg rst;
reg enable;
reg [1:0] clk_p_n;
reg [NUM_LVDS_PAIRS*2-1:0] data_p_n;
reg [1:0] ctrl_p_n;
wire [2*PIXEL_BIT_WIDTH-1:0] pixel_word;
reg[PIXEL_BIT_WIDTH-1:0] training_data;
reg[PIXEL_BIT_WIDTH-1:0] pixel_data;
reg sensor_idle;
wire in_sync;

//IDELAYE2
reg[NUM_LVDS_PAIRS*2-1:0] INC;
wire [NUM_LVDS_PAIRS*2-1:0] data_out;
wire dly_clk_probe;

//ISERDESE2
wire[PIXEL_BIT_WIDTH*NUM_LVDS_PAIRS-1:0] Q_out;
wire CLKDIV_PROBE;

 lvds_deserializer
    #(.PIXEL_BIT_WIDTH(PIXEL_BIT_WIDTH),
      .NUM_LVDS_PAIRS(NUM_LVDS_PAIRS),
      .BIT_CLK_DELAY_TAP_VALUE(BIT_CLK_DELAY_TAP_VALUE))
 DUT
    (.refclk(refclk),
     .rst(rst),
     .enable(enable),
     .clk_p_n(clk_p_n),
     .data_p_n(data_p_n),
     .ctrl_p_n(ctrl_p_n),
     .training_data(training_data),
     .sensor_idle(sensor_idle),
     .in_sync(in_sync),
     .CE(CE),
     .INC(INC),
     .Q_out(Q_out),
     .CLKDIV_PROBE(CLKDIV_PROBE));


integer          iter, i, times;
reg [PIXEL_BIT_WIDTH-1:0] ctrl_chn_training;


always #(2.5) refclk = ~refclk;

always #(PERIOD/2) begin
    clk_p_n[0] = ~clk_p_n[0];
    clk_p_n[1] = ~clk_p_n[0];
end

initial begin
    clk_p_n[0] = 1'b0;
    refclk = 1'b0;
    data_p_n = 0;
    enable = 1'b0;
    CE <= 0;
    INC <= 1'b0;
    sensor_idle = 1'b0;
    rst = 1'b1;
    ctrl_chn_training = 10'b1011010010; //722
    repeat(5) @(posedge clk_p_n[0]);
    rst = 1'b0;
    training_data = 722;
    pixel_data = 0;
    repeat(6) @(posedge clk_p_n[0]);
    enable = 1'b1;
    while(in_sync == 1'b0) begin
        if(enable == 1'b1) begin
            write_data(training_data);
        end
    end
    for (times=0; times < 150; times = times + 1) begin
        write_data(pixel_data);
        pixel_data = pixel_data + 1;
    end
end


task write_data;
    input [PIXEL_BIT_WIDTH-1:0] pattern;
    begin
        for (iter = 0; iter < PIXEL_BIT_WIDTH; iter = iter + 1) begin
            for(i = 0; i < NUM_LVDS_PAIRS; i = i + 1) begin
                data_p_n[i*2] = pattern[iter];
                data_p_n[i*2+1] = ~data_p_n[i*2];
            end
                ctrl_p_n[0] = ctrl_chn_training[iter];
                ctrl_p_n[1] = ~ctrl_p_n[0];
            @(posedge clk_p_n[0] or negedge clk_p_n[0]);
        end
    end
 endtask

endmodule
```

F.2 Pixel Order Alignment Module

F.2.1 Design File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library UNISIM;
use UNISIM.VComponents.all;
Library xpm;
use xpm.vcomponents.all;

```

```vhdl
entity pixel_alignment is
    Generic(
        PIXEL_BIT_WIDTH : positive := 12;
        NUM_LVDS_PAIRS : positive := 4;
        PIXEL_ROW_SIZE : positive := 32;
        NUM_ROWS    : positive := 600;
        NUM_FRAMES  : positive := 8400
    );
    Port (
        wr_clk : in std_logic;
        rd_clk : in std_logic;
        rd_rst : in std_logic;
        rst    : in std_logic;
        valid_in : in std_logic;
        pixel_words : in std_logic_vector(PIXEL_BIT_WIDTH*NUM_LVDS_PAIRS-1 downto 0);

        m_axis_tdata  : out std_logic_vector(47 downto 0);
        --Tell DMA data is valid.
        m_axis_tvalid : out std_logic;
        --DMA is ready to receive data
        m_axis_tready : in  std_logic;
        --Tell DMA this is last data
        m_axis_tlast  : out std_logic

    );
end pixel_alignment;

architecture Behavioral of pixel_alignment is

        constant num_out : integer := PIXEL_ROW_SIZE/4;

        signal full, rd_en, empty : std_logic_vector(NUM_LVDS_PAIRS-1 downto 0);
        signal rd_data_count : std_logic_vector(8*4-1 downto 0);
        signal wr_data_count : std_logic_vector(10*4-1 downto 0);
        signal data_out : std_logic_vector(NUM_LVDS_PAIRS*48-1 downto 0);
        signal out_valid, out_handshake : std_logic;

        signal wr_in_all : std_logic_vector(10*4-1 downto 0);
        signal fifo : integer;

        signal component_cnt : integer := 0;
        signal row_cnt    : integer := 0;
        signal frame_cnt  : integer := 0;

        TYPE State_type IS (idle, output);  -- Define the states
        SIGNAL state : State_Type;
begin

    m_axis_tvalid <= out_valid;

    --Process for write clock domain
    process(wr_clk,rst)
    begin
        if(rst = '1') then
            wr_in_all <= (others => '0');
        elsif(rising_edge(wr_clk))then
            --Count number of written elements in all FIFOs
            if(valid_in = '1') then
                wr_in_all(9 downto 0) <= std_logic_vector( unsigned(wr_in_all(9 downto 0)) + 1 );
                wr_in_all(19 downto 10) <= std_logic_vector( unsigned(wr_in_all(19 downto 10)) + 1 );
                wr_in_all(29 downto 20) <= std_logic_vector( unsigned(wr_in_all(29 downto 20)) + 1 );
                wr_in_all(39 downto 30) <= std_logic_vector( unsigned(wr_in_all(39 downto 30)) + 1 );
            end if;
            --Reset counters when row is reached.
            if(to_integer(unsigned(wr_in_all(9 downto 0))) = PIXEL_ROW_SIZE) then
                wr_in_all(9 downto 0)   <= "0000000001";
                wr_in_all(19 downto 10) <= "0000000001";
                wr_in_all(29 downto 20) <= "0000000001";
                wr_in_all(39 downto 30) <= "0000000001";
            end if;
        end if;
    end process;

    --Process for read clock domain
    process(rd_clk,rd_rst)
            variable value_cnt : integer := 0;
            variable last_flag : std_logic;
            variable fwft_flag : std_logic := '0';
        begin
            if(rd_rst = '1') then
                fifo <= 0;
                state <= idle;
                rd_en <= (others => '0');
                out_valid <= '0';
                fwft_flag := '0';
                component_cnt <= 0;
                row_cnt <= 0;
                frame_cnt <= 0;
                m_axis_tlast <= '0';
                last_flag := '0';
            elsif(rising_edge(rd_clk))then
```

```
                case state is
                    when idle =>
                        m_axis_tlast <= '0';
                        out_valid <= '0';
                        --Wait to all FIFOs are filled
                        if(to_integer(unsigned(wr_in_all(9 downto 0))) >= PIXEL_ROW_SIZE) then
                            state <= output;
                            if(fwft_flag = '0') then -- If first time, read out first value of FIFOs
                                rd_en <= (others => '1');
                                fwft_flag := '1';
                            end if;
                        end if;
                    when output =>
                        if(m_axis_tready = '1' and (valid_in = '1' or row_cnt = NUM_ROWS-1)) then
                                          --Make sure CubeDMA is ready to receive
                            out_valid <= '1';                   -- valid data on AXI-s
                            rd_en <= (others => '0');                    -- Stop inital readout
                            rd_en(fifo) <= '1';                 -- Start selected fifo readout
                            value_cnt := value_cnt + 1;
                            component_cnt <= component_cnt + 1;
                            if(value_cnt = num_out+1) then
                                rd_en(fifo) <= '0';
                                --state <= ch_2;
                                if(fifo = 3) then           -- Row is completed
                                    state <= idle;          -- Return to idle
                                    out_valid <= '0';       -- Disable AXI-valid
                                    fifo <= 0;              -- Make sure fifo cannot reach 4 to avoid error
                                    value_cnt := 0;
                                else                        -- SubRow is completed, increment selected fifo
                                    rd_en(fifo +1 ) <= '1';
                                    fifo <= fifo + 1;
                                    value_cnt := 1;
                                end if;
                            end if;
                            if(component_cnt = PIXEL_ROW_SIZE-1 and row_cnt = NUM_ROWS-1 and last_flag =
                                  '1') then --Readout is completed
                                m_axis_tlast <= '1';
                                state <= idle;
                                component_cnt <= 0;
                                row_cnt    <= 0;
                                frame_cnt  <= 0;
                                rd_en <= (others => '0');
                            end if;
                        else    -- If CubeDMA is not ready to receive, disable fifo readout and AXI valid
                            rd_en(fifo) <= '0';
                            out_valid <= '0';
                        end if;
                        -- Counters to monitor progress
                        if(component_cnt = PIXEL_ROW_SIZE) then
                            row_cnt <= row_cnt + 1;
                            component_cnt <= 0;
                        end if;
                        if(row_cnt = NUM_ROWS) then
                            frame_cnt <= frame_cnt + 1;
                            row_cnt <= 0;
                        end if;
                        if(frame_cnt = NUM_FRAMES-1) then
                            last_flag := '1';
                        end if;
                end case;
            end if;
    end process;

-- Control Mux for m_axis_tdata
m_axis_tdata <= data_out(47 downto 0) when fifo = 0 else
                data_out(95 downto 48) when fifo = 1 else
                data_out(143 downto 96) when fifo = 2 else
                data_out(191 downto 144) when fifo = 3 else
                (others => '0');



-- xpm_fifo_async: Asynchronous FIFO
-- Xilinx Parameterized Macro, Version 2017.4

fifo_gen: for i in 0 to NUM_LVDS_PAIRS-1 generate
    xpm_fifo_async_inst : xpm_fifo_async
      generic map (

        FIFO_MEMORY_TYPE        => "auto",           --string; "auto", "block", or "distributed";
        ECC_MODE                => "no_ecc",         --string; "no_ecc" or "en_ecc";
        RELATED_CLOCKS          => 0,                --positive integer; 0 or 1
        FIFO_WRITE_DEPTH        => 1024,             --positive integer
        WRITE_DATA_WIDTH        => PIXEL_BIT_WIDTH,   --positive integer
        WR_DATA_COUNT_WIDTH     => 10,               --positive integer
        PROG_FULL_THRESH        => 13,               --positive integer
        FULL_RESET_VALUE        => 0,                --positive integer; 0 or 1;
        USE_ADV_FEATURES        => "0707",           --string; "0000" to "1F1F";
        READ_MODE               => "std",            --string; "std" or "fwft";
        FIFO_READ_LATENCY       => 1,                --positive integer;
        READ_DATA_WIDTH         => 48,               --positive integer
        RD_DATA_COUNT_WIDTH     => 8,                --positive integer
        PROG_EMPTY_THRESH       => 3,                --positive integer
        DOUT_RESET_VALUE        => "0",               --string
        CDC_SYNC_STAGES         => 2,                 --positive integer
        WAKEUP_TIME             => 0                  --positive integer; 0 or 2;
      )
      port map (

        sleep            => '0',
        rst              => rst,
        wr_clk           => wr_clk,
        wr_en            => valid_in,
        din              => pixel_words(PIXEL_BIT_WIDTH*i + PIXEL_BIT_WIDTH-1 downto PIXEL_BIT_WIDTH*i),
        full             => full(i),
        overflow         => open,
        wr_rst_busy      => open,
        prog_full        => open,
        wr_data_count    => wr_data_count(10*i + 9 downto 10*i),
        almost_full      => open,
        wr_ack           => open,
        rd_clk           => rd_clk,
        rd_en            => rd_en(i),
        dout             => data_out(i*48 + 48-1 downto i*48),
        empty            => empty(i),
        underflow        => open,
        rd_rst_busy      => open,
        prog_empty       => open,
        rd_data_count    => rd_data_count(8*i + 7 downto i*8),
        almost_empty     => open,
        data_valid       => open,
        injectsbiterr    => '0',
        injectdbiterr    => '0',
        sbiterr          => open,
        dbiterr          => open
      );
end generate fifo_gen;
end Behavioral;
```

F.2.2 Testbench

```
`timescale 1ns / 1ps

module pixel_alignment_tb;
parameter NUM_LVDS_PAIRS = 4;
parameter PIXEL_BIT_WIDTH = 12;
parameter PIXEL_ROW_SIZE = 512;
parameter PERIOD = 10;

reg wr_clk, rd_clk, rd_rst, rst, valid_in, m_axis_tready, binning_enabled;
reg[PIXEL_BIT_WIDTH*NUM_LVDS_PAIRS-1:0] pixel_words;
reg[7:0] binning_factor;
wire[47:0] m_axis_tdata;
wire m_axis_tvalid,m_axis_tlast;

 pixel_alignment
    #(.NUM_LVDS_PAIRS(NUM_LVDS_PAIRS),
      .PIXEL_BIT_WIDTH(PIXEL_BIT_WIDTH),
      .PIXEL_ROW_SIZE(PIXEL_ROW_SIZE))
 DUT
    (.wr_clk(wr_clk),
     .rd_clk(rd_clk),
     .rd_rst(rd_rst),
     .rst(rst),
     .valid_in(valid_in),
     .m_axis_tready(m_axis_tready),
     .pixel_words(pixel_words),
     .m_axis_tdata(m_axis_tdata),
     .m_axis_tvalid(m_axis_tvalid),
     .m_axis_tlast(m_axis_tlast));



always #(PERIOD*1.25) begin
    wr_clk = ~wr_clk;
end

always #(PERIOD/2) begin
    rd_clk = ~rd_clk;
end


integer i, iter;

initial begin
    wr_clk = 1'b0;
    rd_clk = 1'b0;
    rst = 1'b1;
    valid_in = 1'b0;
    rd_rst = 1'b1;
    m_axis_tready = 1'b0;
    repeat(5) @(posedge wr_clk);
    rd_rst = 1'b0;
    rst = 1'b0;
    m_axis_tready = 1'b1;
    repeat(5) @(posedge wr_clk);
    valid_in = 1'b1;
    for( i = 0; i < 2560; i = i + 1) begin
        for(iter = 0; iter < 4; iter = iter + 1) begin
            pixel_words[PIXEL_BIT_WIDTH*iter +: PIXEL_BIT_WIDTH] =  i;//$urandom % 4095;
        end
        repeat(1) @(posedge wr_clk);
    end
    valid_in = 1'b0;
end

endmodule
```

F.3 Binning

F.3.1 Design file

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.MATH_REAL.ALL;
use IEEE.NUMERIC_STD.ALL;
library UNISIM;
use UNISIM.VComponents.all;

entity binning_par is
      Generic(
        NUM_LVDS_PAIRS : positive := 4;
        PIXEL_BIT_WIDTH : positive := 12;
        ACCUMULATOR_BIT_WIDTH : positive := 16;
        BINNING_FACTOR : real := 16.0
      );
      Port (
        clk : in std_logic;
        rst : in std_logic;

        s_axis_tvalid : in std_logic;
        s_axis_tready : out std_logic;
        s_axis_tlast  : in std_logic;
        s_axis_tdata  : in std_logic_vector(NUM_LVDS_PAIRS*PIXEL_BIT_WIDTH-1 downto 0);

        m_axis_tdata  : out std_logic_vector(PIXEL_BIT_WIDTH-1 downto 0);
        m_axis_tvalid : out std_logic;
        m_axis_tready : in  std_logic;
        m_axis_tlast  : out std_logic
         );
end binning_par;

architecture Behavioral of binning_par is

constant shifts : real := LOG2(BINNING_FACTOR);
constant use_shift : integer := integer(shifts);


signal input_reg : std_logic_vector(ACCUMULATOR_BIT_WIDTH-1 downto 0);
signal reg_accumulator : std_logic_vector(ACCUMULATOR_BIT_WIDTH-1 downto 0);

signal add1, add2, add3 : std_logic_vector(ACCUMULATOR_BIT_WIDTH-1 downto 0);
signal valid_in_reg : std_logic;
signal data_out : std_logic_vector(PIXEL_BIT_WIDTH-1 downto 0);
signal data_in : std_logic_vector(NUM_LVDS_PAIRS*PIXEL_BIT_WIDTH-1 downto 0);
signal counter : integer;

signal valid_out : std_logic;
signal valid_in  : std_logic;

signal in_handshake : std_logic;
signal out_handshake : std_logic;
signal in_last : std_logic;
signal in_rdy : std_logic;
begin

data_out <= reg_accumulator(15-(4-integer(shifts))downto integer(shifts)) when rst = '0' else (others =>
      '0');
data_in <= s_axis_tdata;

```

```
add1 <= std_logic_vector(resize(unsigned("00"&data_in(11 downto 0)) + unsigned("00"&data_in(23 downto 12)),
      add1'length));
add2 <= std_logic_vector(resize(unsigned("00"&data_in(47 downto 36)) + unsigned("00"&data_in(35 downto 24)),
       add2'length));
add3 <= std_logic_vector(resize(unsigned(add1) + unsigned(add2), add3'length));

in_rdy <= m_axis_tready;
s_axis_tready <= in_rdy;
m_axis_tvalid <= valid_out;
valid_in <= '1' when in_rdy='1' and s_axis_tvalid = '1' else '0';
m_axis_tdata <= data_out;


bin_4: if(use_shift = 2) generate
    process(clk, rst)
        variable init_flag : std_logic := '0';
    begin
        if(rst = '1') then
            valid_out <= '0';
            reg_accumulator <= (others => '0');
            m_axis_tlast <= '0';
            in_last <= '0';
        elsif(rising_edge(clk)) then
                valid_out <= '0';
                in_last <= s_axis_tlast;
                m_axis_tlast <= in_last;
                if(valid_in = '1' and init_flag = '1') then
                    reg_accumulator <= add3;
                    valid_out <= '1';
                else
                    init_flag := '1';
                end if;
        end if;
    end process;
end generate bin_4;

bin_8: if(use_shift = 3) generate
    process(clk, rst)

        variable last_flag : std_logic_vector(2 downto 0);
    begin
        if(rst = '1') then
            valid_out <= '0';
            reg_accumulator <= (others => '0');
            valid_in_reg <= '0';
            input_reg <= (others => '0');
            counter <= 0;
            m_axis_tlast <= '0';
            last_flag := "000";
        elsif(rising_edge(clk)) then
            input_reg <= add3;
            last_flag := last_flag(1 downto 0) & s_axis_tlast;
            m_axis_tlast <= last_flag(2);
            valid_in_reg <= valid_in;
            valid_out <= '0';
            if(valid_in_reg = '1') then
                if(counter = 0) then
                    reg_accumulator <= input_reg;
                    counter <= counter + 1;
                else
                    reg_accumulator <= std_logic_vector(resize(unsigned(input_reg) + unsigned(
                          reg_accumulator),reg_accumulator'length));
                    if(counter = use_shift-2) then
                        counter <= 0;
                        valid_out <= '1';
                    else
                        counter <= counter + 1;
                    end if;
                end if;

            end if;
        end if;
    end process;
end generate bin_8;

bin_16: if(use_shift = 4) generate
    process(clk, rst)
    --variable counter : integer := 0;
    variable last_flag : std_logic_vector(2 downto 0);
    begin
        if(rst = '1') then
            valid_out <= '0';
            reg_accumulator <= (others => '0');
            valid_in_reg <= '0';
            input_reg <= (others => '0');
            counter <= 0;
            m_axis_tlast <= '0';
            last_flag := "000";
        elsif(rising_edge(clk)) then
            input_reg <= add3;
            last_flag := last_flag(1 downto 0) & s_axis_tlast;
            m_axis_tlast <= last_flag(2);
            valid_in_reg <= valid_in;
            valid_out <= '0';
            if(valid_in_reg = '1') then
                if(counter = 0) then
                    reg_accumulator <= input_reg;
                    counter <= counter + 1;
                else
                    reg_accumulator <= std_logic_vector(resize(unsigned(input_reg) + unsigned(
                          reg_accumulator),reg_accumulator'length));
                    if(counter = use_shift-1) then
                        counter <= 0;
                        valid_out <= '1';
                    else
                        counter <= counter + 1;
                    end if;
                end if;

            end if;
        end if;
    end process;
end generate bin_16;

end Behavioral;
```

F.3.2 Test Bench

```
`timescale 1ns / 1ps

module binning_tb;


parameter NUM_LVDS_PAIRS = 4;
parameter PIXEL_BIT_WIDTH = 12;
parameter ACCUMULATOR_BIT_WIDTH = 16;
parameter BINNING_FACTOR = 4;
parameter PERIOD = 10;

reg clk;
reg rst;
reg[NUM_LVDS_PAIRS*PIXEL_BIT_WIDTH-1:0] pixel_data;
wire[NUM_LVDS_PAIRS*PIXEL_BIT_WIDTH-1:0] data_out;
wire valid_out;
reg valid_in;

 binning
    #(.NUM_LVDS_PAIRS(NUM_LVDS_PAIRS),
      .PIXEL_BIT_WIDTH(PIXEL_BIT_WIDTH),
      .ACCUMULATOR_BIT_WIDTH(ACCUMULATOR_BIT_WIDTH),
      .BINNING_FACTOR(BINNING_FACTOR))
 DUT
    (.clk(clk),
     .rst(rst),
     .pixel_data(pixel_data),
     .data_out(data_out),
     .valid_in(valid_in),
     .valid_out(valid_out));

integer i,iter;


always #(PERIOD/2) begin
    clk = ~clk;
end


initial begin
    clk = 1'b0;
    rst = 1'b1;
    pixel_data = 0;
    valid_in = 1'b0;
    repeat(6) @(posedge clk);
    rst = 1'b0;
    repeat(6) @(posedge clk);
    valid_in = 1'b1;
    for(iter = 0; iter < 12; iter = iter + 1) begin
        for (i = 0; i < NUM_LVDS_PAIRS; i = i + 1) begin
            pixel_data[PIXEL_BIT_WIDTH*i +: PIXEL_BIT_WIDTH] = $urandom % 4095;
        end
        repeat(1) @(posedge clk);
    end
end



```

```
endmodule
```

F.4 Control Interface

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library UNISIM;
use UNISIM.VComponents.all;

entity Control_Interface is
  Port (
    rst : in std_logic;
    gpio_in_1 : in std_logic_vector(31 downto 0);
    gpio_in_2 : in std_logic_vector(31 downto 0);
    gpio_out_1 : out std_logic_vector(31 downto 0);
    enable : out std_logic;
    training_pattern : out std_logic_vector(11 downto 0);
    sensor_idle :  out std_logic;
    INC         : out std_logic_vector(9 downto 0);
    CE          : out std_logic_vector(9 downto 0);
    in_synq : in std_logic;
    frame_req : out std_logic
  );
end Control_Interface;

architecture Behavioral of Control_Interface is


begin
process(rst)
begin
    if(rst = '1') then
        enable <= gpio_in_1(0);
        training_pattern <= gpio_in_1(31 downto 22) ;
        sensor_idle <= gpio_in_1(1);
        INC <= gpio_in_1(11 downto 2);
        CE <= gpio_in_1(21 downto 12);
        frame_req <= gpio_in_2(0);
        gpio_out_1(0) <= in_synq;
    elsif(rst = '0') then
        enable <= '0';
        training_pattern <=(others => '0');
        sensor_idle <= '0';
        INC <= (others => '0');
        CE <= (others => '0');
        frame_req <= '0';
        gpio_out_1(0) <= '0';
    end if;
end process;
end Behavioral;
```

G EMSC HW/SW implementation

G.1 Sequential Implementation

G.1.1 Software

Listing 2: C++ code using listings

```cpp
#include <stdio.h>
#include "xil_printf.h"     //Printf for Uart
#include "Eigen/dense"      //Eigen
#include <stdlib.h>         //atof
#include <math.h>           //Pow, sqrt
#include <float.h>
#include "xparameters.h"    //Board specific parameters
#include "xuartps.h"        //Uart
#include <string.h>
#include "xtmrctr.h"        //Axi Timer

//Interrupt
#include "xscugic.h"
#include "xil_exception.h"
```

```
//Eigen
using Eigen::MatrixXd;

//Axi timer
#define TMRCTR_DEVICE_ID  XPAR_TMRCTR_0_DEVICE_ID
#define TIMER_COUNTER_0  0
XTmrCtr TimerCounter;

//Uart
#define UART_DEVICE_ID      XPAR_PS7_UART_1_DEVICE_ID
XUartPs Uart_Ps;


//Interrupt
bool mm2s_complete = false;
bool s2mm_complete = false;
int int_counter = 0;
const u32 MM2S_INT = 61U;
const u32 S2MM_INT = 62U;
const u32 P_INT = 63U;
XScuGic_Config* scugic_config;
XScuGic scugic_inst;
u32* dma_regs = (u32*)0x43C00000;


/*Function Prototypes ******************************/
void mean(double** ref_spectra, double* mean,  int nVars, int refOrder);
void EMSC(double ** ref_spectra,
          double ** corrected, int nVars,
          int nObs, int refOrder);
double ** initialize(int rows, int columns);
int init_timer(u16 DeviceId, u8 TmrCtrNumber);
u32 start_timer(u8 TmrCtrNumber);
u32 stop_timer(u8 TmrCtrNumber);
static void p_int_irq_handler(void* ref);
static void dma_irq_handler(void* ref);
/***************************************************/





double ** initialize(int rows, int columns) {
    double **temp;
    temp = (double **)malloc(rows * sizeof(double*));
    for (int row = 0; row < rows; row++) {
        temp[row] = (double*)malloc(columns * sizeof(double));
    }
    return temp;}

static void p_int_irq_handler(void* ref){
    int_counter += 1;
}

static void dma_irq_handler(void* ref) {
    int instance = (int)ref;
    int status_reg;
    u32 mask = 0;

    if (instance == 0) {
        status_reg = 1;
        mm2s_complete = true;
    }
    else{
        status_reg = 9;
        s2mm_complete = true;
    }
    //else{
    //  int_counter = int_counter + 1;
    //}
    mask = dma_regs[status_reg];
    dma_regs[status_reg] = (1 << 5);
}

int init_interrupt_system(){

     //Initialize Interrupt system
    //-------------------------------------------------------------------------

    int ret;


    scugic_config = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
    if(NULL == scugic_config){
        return XST_FAILURE;
    }
    ret = XScuGic_CfgInitialize(&scugic_inst, scugic_config, scugic_config->CpuBaseAddress);
    if (ret != XST_SUCCESS) {
        print("Failed to initialize GIC\n");
        return ret;
    }



    u32 id_full = XScuGic_CPUReadReg(&scugic_inst, XSCUGIC_INT_ACK_OFFSET);
    XScuGic_CPUWriteReg(&scugic_inst, XSCUGIC_EOI_OFFSET, id_full);

    ret = XScuGic_SelfTest(&scugic_inst);
    if (ret != XST_SUCCESS) {
        return XST_FAILURE;
    }


    ret = XScuGic_Connect(&scugic_inst, MM2S_INT, (Xil_InterruptHandler)dma_irq_handler, (void*)0);
       if (ret != XST_SUCCESS)
           return ret;

   ret = XScuGic_Connect(&scugic_inst, S2MM_INT, (Xil_InterruptHandler)dma_irq_handler, (void*)1);
       if (ret != XST_SUCCESS)
           return ret;

    //ret = XScuGic_Connect(&scugic_inst, P_INT, (Xil_InterruptHandler)p_int_irq_handler, (void*)2);
    if (ret != XST_SUCCESS){
            print("Failed to initialize GIC 3\n");
            return ret;
    }

    XScuGic_SetPriorityTriggerType(&scugic_inst, MM2S_INT, 0xA0, 0x3);
    XScuGic_SetPriorityTriggerType(&scugic_inst, S2MM_INT, 0xA0, 0x3);
    //XScuGic_SetPriorityTriggerType(&scugic_inst, P_INT, 0xA0, 0x3);

    XScuGic_Enable(&scugic_inst, MM2S_INT);
    XScuGic_Enable(&scugic_inst, S2MM_INT);
    //XScuGic_Enable(&scugic_inst, P_INT);


    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                (Xil_ExceptionHandler)XScuGic_InterruptHandler,
                                 &scugic_inst);
    Xil_ExceptionEnable();

    //-------------------------------------------------------------------------
    return XST_SUCCESS;
}




void EMSC(double ** ref_spectra, double * mean_spectra, double ** corrected,
          int nVars, int nObs, int refOrder){

    //DECLARATIONS---------------------
    MatrixXd M(refOrder + 4, nVars);
    //double ** G = initialize(nVars, refOrder+4);
    double ** corr_M = initialize(2, nVars);
    double num = 0;
    double multiplier = pow(2.0, 25.0);
    //---------------------------------


    for (int i = 0; i < nVars; i++) {

        //Add 1 in first row
        M(0,i) = 1;

        //Add linspace and linspace squared
        M(1,i) = num;
        corr_M[0][i] = num;

        M(2,i) = pow(num, 2);
        corr_M[1][i] = pow(num,2);
        num += (1.0 / (nVars - 1));

        //Add reference spectra
        for (int y = 0; y < refOrder; y++) {
            M(y + 3,i) = ref_spectra[y][i];
        }

        //Add mean in last row
        M(refOrder+3,i) = mean_spectra[i];
    }



    //Initiate Block Ram
    //----------------------------------------------
    //Create pointer to Block Ram base address
    u32 * init = (u32*)0x43c10000;
    //Creates a pointer to the address to write G
    u32 * in_G = (u32*)0x43c10004;
    u32 * num_pixels = (u32*)0x43c10008;

    //
    *init = 0x22034;
    *num_pixels = 0x3D090;
    //Execute pseudo-inverse of M
    MatrixXd M_M = M*M.transpose();
    MatrixXd p_inv = M.transpose() * M_M.completeOrthogonalDecomposition().pseudoInverse();

    for(int y = 0; y<refOrder+4; y++){
        for(int i = 0; i<nVars; i++){
            *in_G  =(int) floor(p_inv(i,y)* multiplier);
            //save_G[index++] = floor(p_inv(i,y)* multiplier);
        }

    }
    *init = 0x20034;
    *init = 0x21034;
    //-----------------------------------------------







    //Initiate and enable Cube DMA
    //-----------------------------------------------
    u32* mm2s = (u32*)0x43c00000;
    u32* s2mm = (u32*)0x43c00020;

    // Program S2MM DMA

    s2mm[0] = 0x0;
    s2mm[2] = 0x0F0BDBF0;
    s2mm[0] = (1 << 5) | 1;


    // Program MM2S DMA
    mm2s[0] = 0;
    mm2s[2] = 0x100010E0;

    //mm2s[3] = 0x1001001; //liten kube
    //mm2s[5] = 520000; //liten kube

    mm2s[3] = 0x341F41F4; //stor kube  |52|500|500
    mm2s[5] = 0x6590;//stor kube       |26000| 52*500



    mm2s[0] = (1 << 8) | (1 << 5) | 1;

    while (!s2mm_complete || !mm2s_complete); //comment this if non-blocking is used
    //while ((mm2s[1] != 0x1) || (s2mm[1] != 0x3D090001));
    //-----------------------------------------------



    //Calculate the corrected spectra
    //-----------------------------------------
    int64_t * test_ptr = (int64_t*)0x0F0BDBF0;
    u16 * raw_ptr = (u16*)0x100010E0;
    double p_st[8];
    u16 ah;
    int counter = 0;
    //-----------------------------------------
    while(counter < nObs){
        //if((int_counter - counter > 10) || (nObs - int_counter < 10)){ //Uncomment for non-blocking
            for(int i = 0; i < 8; i++){
                p_st[i] = test_ptr[i+counter*8]/multiplier;
            }
            for(int cols = 0; cols < nVars; cols++){
                ah = raw_ptr[counter*52+cols];
                corrected[counter][cols] = (ah - (p_st[0] + p_st[1]*corr_M[0][cols] + p_st[2]*corr_M[1][cols
                      ]))/p_st[7];
            }
            counter++;
        //} //Uncomment for non-blocking
    }


    return;
}

//Axi-Timer
//----------------------------------------
int init_timer(u16 DeviceId, u8 TmrCtrNumber){
    int Status;
        XTmrCtr *TmrCtrInstancePtr = &TimerCounter;
        /*
         * Initialize the timer counter so that it's ready to use,
         * specify the device ID that is generated in xparameters.h
         */
        Status = XTmrCtr_Initialize(TmrCtrInstancePtr, DeviceId);
        if (Status != XST_SUCCESS) {
            return XST_FAILURE;
        }
        /*
         * Perform a self-test to ensure that the hardware was built
         * correctly, use the 1st timer in the device (0)
         */
        Status = XTmrCtr_SelfTest(TmrCtrInstancePtr, TmrCtrNumber);
        if (Status != XST_SUCCESS) {
            return XST_FAILURE;
        }
        /*
        * Enable the Autoreload mode of the timer counters.
        */
        return XST_SUCCESS;}

u32 start_timer(u8 TmrCtrNumber){
    XTmrCtr *TmrCtrInstancePtr = &TimerCounter;
    XTmrCtr_SetOptions(TmrCtrInstancePtr, TmrCtrNumber,
                            XTC_AUTO_RELOAD_OPTION);
    u32 val = XTmrCtr_GetValue(TmrCtrInstancePtr, TmrCtrNumber);
    XTmrCtr_Start(TmrCtrInstancePtr, TmrCtrNumber);
    return val;}

u32 stop_timer(u8 TmrCtrNumber){
    XTmrCtr *TmrCtrInstancePtr = &TimerCounter;
    u32 val = XTmrCtr_GetValue(TmrCtrInstancePtr, TmrCtrNumber);
    XTmrCtr_SetOptions(TmrCtrInstancePtr, TmrCtrNumber, 0);
    return val;
}
//----------------------------------------
int main(){
    //RAW: 0x100010E0;
    //REF_SPECTRA: 0x10000000;
    //MEAN: 0x10001000;
    //CORRECTED: 0x19CD1534;
    //Initialize interrupts

    init_interrupt_system();

    //Adding pointer to location of stored cube.
    float * mem_ptr = (float*)0x10000000;
    int cubeDepth = 52;
    int cubeHeigth = 500;
    int cubeWidth = 500;
    int nObs = cubeHeigth*cubeWidth; //total number of pixels
    int nVars = cubeDepth;
    int refOrder = 4; //numbers of species in spectra

    double ** ref_spectra = initialize(refOrder, nVars);
    double ** corrected = initialize(nObs, nVars);
    double * mean_v =  (double*)malloc(nVars * sizeof(double));

    //Fill raw matrix1
    int index = 0;
    //Fetch ref_spectra from memory
    index = 0;
    mem_ptr = (float*)0x10000000;
    for(int rows = 0; rows<refOrder; rows++){
        for(int cols = 0; cols < nVars; cols++){
            ref_spectra[rows][cols] = mem_ptr[index++];
        }
    }
    //Fetch mean from memory
    mem_ptr = (float*)0x10001000;
    for(int i = 0; i<nVars; i++){
        mean_v[i] = mem_ptr[i];
    }


    //Start the EMSC
    xil_printf("ESMC Starting!\n");

    //Set timer

    u32 value1, value2;
    init_timer(TMRCTR_DEVICE_ID, TIMER_COUNTER_0);
    value1 = start_timer(TIMER_COUNTER_0);

    EMSC(ref_spectra, mean_v, corrected, nVars, nObs, refOrder);

    //Stop timer
    value2 = stop_timer(TIMER_COUNTER_0);
    xil_printf("Timer: %d\n", value2-value1);
    xil_printf("Counter: %d\n", int_counter);


    //Store result in memory
    mem_ptr = (float*)0x19CD1534;
            index = 0;
            for(int i = 0; i<nObs; i++){
                for(int y = 0; y<nVars; y++){
                    mem_ptr[index++] = (float)corrected[i][y];
                }
            }


    xil_printf("Done");

    XScuGic_Disable(&scugic_inst, MM2S_INT);
    XScuGic_Disable(&scugic_inst, S2MM_INT);
    //XScuGic_Disable(&scugic_inst, P_INT);

    XScuGic_Disconnect(&scugic_inst, MM2S_INT);
    XScuGic_Disconnect(&scugic_inst, S2MM_INT);
    //XScuGic_Disconnect(&scugic_inst, P_INT);

    return 0;
}
```

G.1.2 Top module design file

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity top is
    Generic(
        B_RAM_SIZE      : integer := 400;
        NUM_B_RAM       : integer := 16;
        RAW_BIT_WIDTH : positive := 16;
        G_BIT_WIDTH   : positive := 32;
        P_BIT_WIDTH   : positive := 64;
        C_S_AXI_DATA_WIDTH : integer := 32;
        C_S_AXI_ADDR_WIDTH : integer := 6
    );
   Port (
        clk     :   in std_logic;
        aresetn :   in std_logic;
        p_irq   :   out std_logic;

        --AXI in-stream
        s_axis_tdata  : in  std_logic_vector(RAW_BIT_WIDTH-1 downto 0);
        --DMA is ready to send data
        s_axis_tvalid : in  std_logic;
        --EMSC is ready to receive data
        s_axis_tready : out std_logic;
        --DMA say this is last data
        s_axis_tlast  : in  std_logic;

        --AXI out-stream
        m_axis_tdata  : out std_logic_vector(63 downto 0);
        --EMSC is ready to send to DMA.
        m_axis_tvalid : out std_logic;
        --DMA is ready to receive data
        m_axis_tready : in  std_logic;
        --Tell DMA this is last data
        m_axis_tlast  : out std_logic;


        -- Register interface
        s_axi_ctrl_status_awaddr  : in  std_logic_vector(5 downto 0);
        s_axi_ctrl_status_awprot  : in  std_logic_vector(2 downto 0);
        s_axi_ctrl_status_awvalid : in  std_logic;
        s_axi_ctrl_status_awready : out std_logic;
        s_axi_ctrl_status_wdata   : in  std_logic_vector(31 downto 0);
        s_axi_ctrl_status_wstrb   : in  std_logic_vector(3 downto 0);
        s_axi_ctrl_status_wvalid  : in  std_logic;
        s_axi_ctrl_status_wready  : out std_logic;
        s_axi_ctrl_status_bresp   : out std_logic_vector(1 downto 0);
        s_axi_ctrl_status_bvalid  : out std_logic;
        s_axi_ctrl_status_bready  : in  std_logic;
        s_axi_ctrl_status_araddr  : in  std_logic_vector(5 downto 0);
        s_axi_ctrl_status_arprot  : in  std_logic_vector(2 downto 0);
        s_axi_ctrl_status_arvalid : in  std_logic;
        s_axi_ctrl_status_arready : out std_logic;
        s_axi_ctrl_status_rdata   : out std_logic_vector(31 downto 0);
        s_axi_ctrl_status_rresp   : out std_logic_vector(1 downto 0);
        s_axi_ctrl_status_rvalid  : out std_logic;
        s_axi_ctrl_status_rready  : in  std_logic
   );
end top;

```

```
architecture Behavioral of top is

-- AXI in-stream signals
signal in_stream_data  : std_logic_vector(RAW_BIT_WIDTH-1 downto 0);
signal in_stream_valid : std_logic;
signal in_stream_ready : std_logic;
signal in_stream_last  : std_logic;
signal in_stream_handshake : std_logic;
signal in_stream_handshake_delay : std_logic;
signal in_raw_delay :   std_logic_vector(RAW_BIT_WIDTH-1 downto 0);


-- AXI out-stream signals
signal out_stream_data  : std_logic_vector(63 downto 0);
signal out_stream_valid : std_logic;
signal out_stream_ready : std_logic;
signal out_stream_last  : std_logic;
signal out_stream_handshake : std_logic;

-- Signals from/to b_ram_bank
signal read_enable : std_logic;
signal b_ram_out   : std_logic_vector(G_BIT_WIDTH*NUM_B_RAM-1 downto 0);
signal enable      : std_logic;
signal v_len       : std_logic_vector(11 downto 0);
signal Ref_order   : std_logic_vector(5 downto 0);
signal num_pixels  : std_logic_vector(31 downto 0);
signal initialized : std_logic;

-- Signals from/to dot_product_module
signal p_rdy_w  :   std_logic_vector(NUM_B_RAM-1 downto 0);
signal p_rdy    :   std_logic;
signal p_out    :   std_logic_vector(NUM_B_RAM*P_BIT_WIDTH-1 downto 0);
signal dp_extend_end : std_logic;
signal dp_enable : std_logic;


-- Signals from/to AXI gear box
signal last_p : std_logic;

begin
--Connections
in_stream_data  <= s_axis_tdata;
in_stream_valid <= s_axis_tvalid;
s_axis_tready   <= in_stream_ready;
in_stream_last  <= s_axis_tlast;


--Helper signal
in_stream_handshake <= '1' when (in_stream_valid = '1' and in_stream_ready = '1') else '0';

--Output is ready
p_rdy <= '1' when p_rdy_w = (p_rdy_w'range => '1') else '0';
--Dot product module ready
dp_enable <= '1' when (in_stream_handshake_delay = '1' and enable = '1' and initialized = '1') or
       dp_extend_end = '1' else '0';
--B_ram bank read enable
read_enable <= '1' when (in_stream_valid = '1' and enable = '1') and initialized = '1' else '0';

process(clk, aresetn)
    variable counter : integer := 0;
begin
    if (aresetn = '0') then
        in_stream_ready <= '0';
        in_stream_handshake_delay <= '0';
        last_p <= '0';
        counter := 0;
        in_raw_delay <= (others => '0');
    elsif(rising_edge(clk)) then
        last_p <= '0';
        in_raw_delay <= in_stream_data;
        in_stream_handshake_delay <= in_stream_handshake;
        if(enable = '1' and initialized = '1') then
            in_stream_ready <= '1';
        end if;
        if(in_stream_last = '1') then
            dp_extend_end <= '1';
        elsif(dp_extend_end = '1') then
            counter := counter + 1;
            if(counter = 3) then
                last_p <= '1';
                dp_extend_end <= '0';
                counter := 0;
            end if;
        end if;
    end if;
end process;



--B_ram bank declaration
b_ram: entity work.b_ram_bank
```

```
  Generic map(
          B_RAM_SIZE         =>  B_RAM_SIZE,
          B_RAM_BIT_WIDTH    =>  G_BIT_WIDTH,
          NUM_B_RAM          =>  NUM_B_RAM,
          C_S_AXI_DATA_WIDTH =>  C_S_AXI_DATA_WIDTH,
          C_S_AXI_ADDR_WIDTH =>  C_S_AXI_ADDR_WIDTH
       )
  Port map(
        clk             =>  clk,
        aresetn         =>  aresetn,

        -- B_ram interface
        read_enable     =>  read_enable,

        data_out        =>  b_ram_out,
        v_len           =>  v_len,
        R_order         =>  Ref_Order,
        num_pixels      =>  num_pixels,
        init_flag       =>  initialized,


        -- Register interface
        enable          =>  enable,
        s_axi_ctrl_status_awaddr    =>  s_axi_ctrl_status_awaddr,
        s_axi_ctrl_status_awprot    =>  s_axi_ctrl_status_awprot,
        s_axi_ctrl_status_awvalid   =>  s_axi_ctrl_status_awvalid,
        s_axi_ctrl_status_awready   =>  s_axi_ctrl_status_awready,
        s_axi_ctrl_status_wdata     =>  s_axi_ctrl_status_wdata,
        s_axi_ctrl_status_wstrb     =>  s_axi_ctrl_status_wstrb,
        s_axi_ctrl_status_wvalid    =>  s_axi_ctrl_status_wvalid,
        s_axi_ctrl_status_wready    =>  s_axi_ctrl_status_wready,
        s_axi_ctrl_status_bresp     =>  s_axi_ctrl_status_bresp,
        s_axi_ctrl_status_bvalid    =>  s_axi_ctrl_status_bvalid,
        s_axi_ctrl_status_bready    =>  s_axi_ctrl_status_bready,
        s_axi_ctrl_status_araddr    =>  s_axi_ctrl_status_araddr,
        s_axi_ctrl_status_arprot    =>  s_axi_ctrl_status_arprot,
        s_axi_ctrl_status_arvalid   =>  s_axi_ctrl_status_arvalid,
        s_axi_ctrl_status_arready   =>  s_axi_ctrl_status_arready,
        s_axi_ctrl_status_rdata     =>  s_axi_ctrl_status_rdata,
        s_axi_ctrl_status_rresp     =>  s_axi_ctrl_status_rresp,
        s_axi_ctrl_status_rvalid    =>  s_axi_ctrl_status_rvalid,
        s_axi_ctrl_status_rready    =>  s_axi_ctrl_status_rready
  );


--Dot product module declaration
dp: entity work.dot_product_module
    generic map(
        RAW_BIT_WIDTH   =>  RAW_BIT_WIDTH,
        G_BIT_WIDTH     =>  G_BIT_WIDTH,
        NUM_B_RAM       =>  NUM_B_RAM,
        P_BIT_WIDTH     =>  P_BIT_WIDTH
    )
    port map(
        clk             =>  clk,
        aresetn         =>  aresetn,
        en              =>  dp_enable,
        in_G            =>  b_ram_out,
        in_raw          =>  in_raw_delay,
        v_len           =>  v_len,
        p_rdy           =>  p_rdy_w,
        p_out           =>  p_out
    );


--Output Module/Axi_gearbox declaration
gb: entity work.axi_gearbox
    generic map(
        B_RAM_SIZE  =>  B_RAM_SIZE,
        B_RAM_BIT_WIDTH => G_BIT_WIDTH,
        NUM_B_RAM   =>  NUM_B_RAM,
        RAW_BIT_WIDTH   =>  RAW_BIT_WIDTH,
        G_BIT_WIDTH     =>  G_BIT_WIDTH,
        P_BIT_WIDTH     =>  P_BIT_WIDTH,
        C_S_AXI_DATA_WIDTH  =>  C_S_AXI_DATA_WIDTH,
        C_S_AXI_ADDR_WIDTH  =>  C_S_AXI_ADDR_WIDTH
    )
    port map(
        clk         =>      clk,
        aresetn     =>      aresetn,
        p_out       =>      p_out,
        p_rdy       =>      p_rdy,
        p_int       =>      p_irq,
        Ref_Order   =>    Ref_Order,
        enable      =>      enable,
        last_p      =>      last_p,
        num_pixels  =>   num_pixels,
        m_axis_tdata    =>  m_axis_tdata,
        m_axis_tvalid   =>  m_axis_tvalid,
        m_axis_tready   =>  m_axis_tready,
        m_axis_tlast    =>  m_axis_tlast
```

```
   );

end Behavioral;
```

G.1.3 Top module testbench

```
`timescale 1ns / 1ps

module top_tb;
   parameter C_S_AXI_DATA_WIDTH = 32;
   parameter C_S_AXI_ADDR_WIDTH = 6;
   parameter B_RAM_SIZE = 400;
   parameter NUM_B_RAM = 16;
   parameter RAW_BIT_WIDTH = 16;
   parameter G_BIT_WIDTH = 32;
   parameter P_BIT_WIDTH = 64;
   parameter PERIOD = 10;

   reg clk, aresetn;
   reg[15:0] s_axis_tdata;
   reg s_axis_tvalid, s_axis_tlast;
   wire s_axis_tready, p_irq;

   reg[32:0]counter;
   wire[63:0] m_axis_tdata;
   wire m_axis_tvalid, m_axis_tlast;
   reg m_axis_tready;


   reg[5:0]   s_axi_ctrl_status_awaddr;
   reg[2:0]   s_axi_ctrl_status_awprot;
   reg        s_axi_ctrl_status_awvalid;
   reg[31:0]  s_axi_ctrl_status_wdata;
   reg[3:0]   s_axi_ctrl_status_wstrb;
   reg        s_axi_ctrl_status_wvalid;
   reg        s_axi_ctrl_status_bready;
   reg[5:0]   s_axi_ctrl_status_araddr;
   reg[2:0]   s_axi_ctrl_status_arprot;
   reg        s_axi_ctrl_status_arvalid;
   reg        s_axi_ctrl_status_rready;
   wire       s_axi_ctrl_status_awready;
   wire       s_axi_ctrl_status_wready;
   wire[1:0]  s_axi_ctrl_status_bresp;
   wire       s_axi_ctrl_status_bvalid;
   wire       s_axi_ctrl_status_arready;
   wire[31:0] s_axi_ctrl_status_rdata;
   wire[1:0]  s_axi_ctrl_status_rresp;
   wire       s_axi_ctrl_status_rvalid;


   top
      #(.C_S_AXI_DATA_WIDTH(C_S_AXI_DATA_WIDTH),
        .C_S_AXI_ADDR_WIDTH(C_S_AXI_ADDR_WIDTH),
        .B_RAM_SIZE(B_RAM_SIZE),
        .NUM_B_RAM(NUM_B_RAM),
        .RAW_BIT_WIDTH(RAW_BIT_WIDTH),
        .G_BIT_WIDTH(G_BIT_WIDTH),
        .P_BIT_WIDTH(P_BIT_WIDTH))
   DUT
       (.clk(clk),
        .aresetn(aresetn),
        .p_irq(p_irq),
        //IN-STREAM
        .s_axis_tdata(s_axis_tdata),
        .s_axis_tvalid(s_axis_tvalid),
        .s_axis_tready(s_axis_tready),
        .s_axis_tlast(s_axis_tlast),

        //OUT-STREAM
        .m_axis_tdata(m_axis_tdata),
        .m_axis_tvalid(m_axis_tvalid),
        .m_axis_tready(m_axis_tready),
        .m_axis_tlast(m_axis_tlast),

        //REGISTER-INTERFACE
        .s_axi_ctrl_status_awaddr(s_axi_ctrl_status_awaddr),
        .s_axi_ctrl_status_awprot(s_axi_ctrl_status_awprot),
        .s_axi_ctrl_status_awvalid(s_axi_ctrl_status_awvalid),
        .s_axi_ctrl_status_wdata(s_axi_ctrl_status_wdata),
        .s_axi_ctrl_status_wstrb(s_axi_ctrl_status_wstrb),
        .s_axi_ctrl_status_wvalid(s_axi_ctrl_status_wvalid),
        .s_axi_ctrl_status_bready(s_axi_ctrl_status_bready),
        .s_axi_ctrl_status_araddr(s_axi_ctrl_status_araddr),
        .s_axi_ctrl_status_arprot(s_axi_ctrl_status_arprot),
        .s_axi_ctrl_status_arvalid(s_axi_ctrl_status_arvalid),
        .s_axi_ctrl_status_rready(s_axi_ctrl_status_rready),
        .s_axi_ctrl_status_awready(s_axi_ctrl_status_awready),
        .s_axi_ctrl_status_wready(s_axi_ctrl_status_wready),
        .s_axi_ctrl_status_bresp(s_axi_ctrl_status_bresp),
        .s_axi_ctrl_status_bvalid(s_axi_ctrl_status_bvalid),
        .s_axi_ctrl_status_arready(s_axi_ctrl_status_arready),
        .s_axi_ctrl_status_rdata(s_axi_ctrl_status_rdata),
        .s_axi_ctrl_status_rresp(s_axi_ctrl_status_rresp),
        .s_axi_ctrl_status_rvalid(s_axi_ctrl_status_rvalid)
       );

    always #(PERIOD/2) clk = ~clk;

    integer          f_in_G, f_in_raw, f_out_P;
    integer          iter, i;
    reg[31:0] in_G_temp;
    reg[31:0] in_raw_temp;
    reg[31:0] read_holder;
    reg flagg;

    initial begin
        clk = 1'b0;
        aresetn = 1'b0;
        counter = 32'b0;
        s_axis_tlast <= 1'b0;
        s_axi_ctrl_status_awprot = 'b0;
        s_axi_ctrl_status_bready = 1'b0;
        s_axi_ctrl_status_wstrb = 4'hF;
        s_axi_ctrl_status_arprot = 'b0;




        f_in_G = $fopen("D:/MasterOppgave/smallsat_prototype/EMSC/Test/Hardware/in_G.bin", "rb");

        if (f_in_G == 0) begin
            $display("Failed to open input file %s", "D:/MasterOppgave/smallsat_prototype/EMSC/Test/Hardware/in_G.bin");
            $finish;
        end

        f_in_raw = $fopen("D:/MasterOppgave/smallsat_prototype/EMSC/Test/Hardware/raw_large.bin", "rb");

        if (f_in_raw == 0) begin
            $display("Failed to open input file %s", "D:/MasterOppgave/smallsat_prototype/EMSC/Test/Hardware/raw_large.bin");
            $finish;
        end

        f_out_P = $fopen("D:/MasterOppgave/smallsat_prototype/EMSC/Test/P_out_tb.bin", "wb");

        if (f_out_P == 0) begin
            $display("Failed to open input file %s", "D:/MasterOppgave/smallsat_prototype/EMSC/Test/P_out_tb.bin");
            $finish;
        end

        repeat(2) @(posedge clk);
        aresetn = 1'b1;
        write_to_reg(6'h8, 32'h2);
        write_to_reg(6'h0, 32'h22034);
        repeat(3) @(posedge clk);

        for (iter = 0; iter < 416; iter = iter + 1) begin      //|
            for( i = 0; i < 4; i = i + 1) begin      //|
                in_G_temp[i*8  +: 8] = $fgetc(f_in_G);      //|
            end
            write_to_reg(6'h4, in_G_temp);      //|
            @(posedge clk);      //|
        end

        write_to_reg(6'h0, 32'h20034);
        repeat(2) @(posedge clk);

        write_to_reg(6'h0, 32'h21034);
        repeat(10) @(posedge clk);


        s_axis_tvalid <= 1'b1;

        for (iter = 0; iter < 104; iter = iter + 1) begin      //|
            for( i = 0; i < 2; i = i + 1) begin      //|
                in_raw_temp[i*8  +: 8] = $fgetc(f_in_raw);
                                                                                    //|
            end
            s_axis_tdata <= in_raw_temp[15:0];
            if(iter == 103) begin
                s_axis_tlast <= 1'b1;
            end
                  //|
            @(posedge clk);
            counter = counter + 1;

                  //|
         end
        s_axis_tlast <= 1'b0;
        s_axis_tvalid <= 1'b0;



        $fclose(f_in_G);
        $fclose(f_in_raw);
        repeat(100) @(posedge clk);
        $fclose(f_out_P);
    end


    always @(posedge clk) begin
        if($urandom % 3 == 0) begin
            m_axis_tready <= 1'b0;
        end
        else
            m_axis_tready <= 1'b1;
        end

   integer byte_idx, j;

    always @(posedge clk) begin
            if(m_axis_tready == 1'b1 && m_axis_tvalid == 1'b1) begin
                for (byte_idx = 0; byte_idx < 8; byte_idx = byte_idx + 1) begin
                    $fwrite(f_out_P, "%c", m_axis_tdata[byte_idx*8+:8]);
                end
            end
    end

    task write_to_reg;
        input [5:0] address;
        input [31:0] data;
        begin
            @(posedge clk);
            s_axi_ctrl_status_awaddr <= address;
            s_axi_ctrl_status_awvalid <= 1'b1;
            s_axi_ctrl_status_wvalid <= 1'b1;
            s_axi_ctrl_status_wdata <= data;

            while (!(s_axi_ctrl_status_awready == 1'b1 && s_axi_ctrl_status_wready == 1'b1)) begin
                @(posedge clk);
            end

            s_axi_ctrl_status_awvalid <= 1'b0;
            s_axi_ctrl_status_wvalid <= 1'b0;
        end
     endtask



     task read_reg;
             input  [5:0] address;
             output [31:0] data;
             begin
                 @(posedge clk);
                 s_axi_ctrl_status_araddr = address;
                 s_axi_ctrl_status_arvalid = 1'b1;
                 s_axi_ctrl_status_rready = 1'b1;
                 while (!(s_axi_ctrl_status_rvalid == 1'b1)) begin
                     @(posedge clk);
                 end

                 s_axi_ctrl_status_rready <= 1'b0;
                 s_axi_ctrl_status_arvalid <= 1'b0;
                 data = s_axi_ctrl_status_rdata;


             end
          endtask

endmodule
```

G.1.4 Block ram bank design file

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;


entity b_ram_bank is
  Generic(
          B_RAM_SIZE      : integer := 100;
          B_RAM_BIT_WIDTH : integer := 32;
          NUM_B_RAM       : integer := 16;
          C_S_AXI_DATA_WIDTH : integer := 32;
          C_S_AXI_ADDR_WIDTH : integer := 6
        );
  Port (
        clk             :       in std_logic;
        aresetn         :       in std_logic;
        read_enable     :       in std_logic;
        enable          :       out std_logic;

        v_len           :       out std_logic_vector(11 downto 0);
        R_order         :       out std_logic_vector(5 downto 0);
        init_flag       :       out std_logic;
        data_out        :       out std_logic_vector(B_RAM_BIT_WIDTH*NUM_B_RAM-1 downto 0);
        num_pixels      :       out std_logic_vector(31 downto 0);

        -- Register interface
        s_axi_ctrl_status_awaddr  : in  std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
        s_axi_ctrl_status_awprot  : in  std_logic_vector(2 downto 0);
        s_axi_ctrl_status_awvalid : in  std_logic;
        s_axi_ctrl_status_awready : out std_logic;
        s_axi_ctrl_status_wdata   : in  std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        s_axi_ctrl_status_wstrb   : in  std_logic_vector(3 downto 0);
        s_axi_ctrl_status_wvalid  : in  std_logic;
        s_axi_ctrl_status_wready  : out std_logic;
        s_axi_ctrl_status_bresp   : out std_logic_vector(1 downto 0);
        s_axi_ctrl_status_bvalid  : out std_logic;
        s_axi_ctrl_status_bready  : in  std_logic;
        s_axi_ctrl_status_araddr  : in  std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
        s_axi_ctrl_status_arprot  : in  std_logic_vector(2 downto 0);
        s_axi_ctrl_status_arvalid : in  std_logic;
        s_axi_ctrl_status_arready : out std_logic;
        s_axi_ctrl_status_rdata   : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        s_axi_ctrl_status_rresp   : out std_logic_vector(1 downto 0);
        s_axi_ctrl_status_rvalid  : out std_logic;
        s_axi_ctrl_status_rready  : in  std_logic
  );
end b_ram_bank;

architecture Behavioral of b_ram_bank is

--Control/status registers
signal emsc2cpu_register : std_logic_vector(31 downto 0);
signal cpu2emsc_register : std_logic_vector(31 downto 0);
signal in_G_register     : std_logic_vector(31 downto 0);

--Control Signals
signal init, valid_input : std_logic;
signal G_size : std_logic_vector(11 downto 0);
signal Ref_Order : std_logic_vector(5 downto 0);
signal initialized : std_logic;

--Registers
signal data_in_w    : std_logic_vector(B_RAM_BIT_WIDTH-1 downto 0);
signal read_address : integer range 0 to B_RAM_SIZE-1;
signal write_enable : std_logic_vector(NUM_B_RAM-1 downto 0);
signal b_ram_sel   : std_logic_vector(NUM_B_RAM-1 downto 0);
signal data_in     : std_logic_vector(31 downto 0);

TYPE state_type IS (idle, write, read);
SIGNAL state : state_type;

begin

data_in_w <= data_in(B_RAM_BIT_WIDTH-1 downto 0) when aresetn = '1' else (others => '0');

--Set outputs and read from AXI-register interface registers
init_flag <= initialized;
v_len <= G_size;
R_order <= Ref_Order;
G_size <= cpu2emsc_register(11 downto 0);
Ref_Order <= cpu2emsc_register(19 downto 14);
enable <= cpu2emsc_register(12) when initialized = '1' else '0';
init <= cpu2emsc_register(13);
emsc2cpu_register(0) <= initialized;
emsc2cpu_register(31 downto 1) <= (others => '0');


```

```
--AXI register interface declaration
register_interface: entity work.register_interface
    generic map(
        C_S_AXI_DATA_WIDTH => C_S_AXI_DATA_WIDTH,
        C_S_AXI_ADDR_WIDTH => C_S_AXI_ADDR_WIDTH,
        B_RAM_SIZE         => B_RAM_SIZE,
        B_RAM_BIT_WIDTH    => B_RAM_BIT_WIDTH,
        NUM_B_RAM          => NUM_B_RAM
    )
    port map(
         s_axi_aclk    => clk,
         s_axi_aresetn => aresetn,

         s_axi_awaddr  => s_axi_ctrl_status_awaddr,
         s_axi_awprot  => s_axi_ctrl_status_awprot,
         s_axi_awvalid => s_axi_ctrl_status_awvalid,
         s_axi_awready => s_axi_ctrl_status_awready,

         s_axi_wdata  => s_axi_ctrl_status_wdata,
         s_axi_wstrb  => s_axi_ctrl_status_wstrb,
         s_axi_wvalid => s_axi_ctrl_status_wvalid,
         s_axi_wready => s_axi_ctrl_status_wready,

         s_axi_bresp  => s_axi_ctrl_status_bresp,
         s_axi_bvalid => s_axi_ctrl_status_bvalid,
         s_axi_bready => s_axi_ctrl_status_bready,

         s_axi_araddr  => s_axi_ctrl_status_araddr,
         s_axi_arprot  => s_axi_ctrl_status_arprot,
         s_axi_arvalid => s_axi_ctrl_status_arvalid,
         s_axi_arready => s_axi_ctrl_status_arready,

         s_axi_rdata  => s_axi_ctrl_status_rdata,
         s_axi_rresp  => s_axi_ctrl_status_rresp,
         s_axi_rvalid => s_axi_ctrl_status_rvalid,
         s_axi_rready => s_axi_ctrl_status_rready,

         --Register Outputs
         emsc2cpu_register  => emsc2cpu_register,

         --Register Inputs
         cpu2emsc_register => cpu2emsc_register,
         in_G_register     => data_in,
         num_pixels        => num_pixels,
         valid_input       => valid_input
         --read_enable       => read_enable_w
    );

--Block ram declaration
b_ram: for i in 0 to NUM_B_RAM-1 generate
    DUT : 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         =>         aresetn,
        data_in         =>         data_in_w,
        write_enable    =>         write_enable(i),
        read_enable     =>         read_enable,
        read_address    =>         read_address,
        data_out        =>         data_out(B_RAM_BIT_WIDTH*i + B_RAM_BIT_WIDTH-1 downto B_RAM_BIT_WIDTH*i)
    );
end generate b_ram;

--State machine to fill Block rams
process(clk, aresetn)
    variable counter : integer range 0 to B_RAM_SIZE-1 := 0;
    variable b_ram_written : integer range 0 to 32 := 0;
    variable prev_b_ram_addr : std_logic_vector(NUM_B_RAM-1 downto 0);
    variable valid_prev : std_logic;
begin
    if(aresetn = '0') then
        initialized <= '0';
        b_ram_sel <= (others => '0');
        b_ram_written := 0;
        state <= idle;
    elsif(rising_edge(clk)) then
            case state is
                --Stays in idle until either a init or read should be
                --performed
                when idle =>
                    counter := 0;
                    if(init = '1' and valid_input = '1') then
                        state <= write;
                        b_ram_sel <= (0 => '1', others => '0');
                        counter := counter + 1;
                    elsif(read_enable = '1' and initialized = '1') then
                        read_address <= read_address + 1;
                        state <= read;
                    end if;

                --Stays in write until initialization is completed.
                --Has to take care of bubbles in input data
                when write =>
                    if(valid_input = '1') then
                        counter := counter + 1;
                        if(counter >= to_integer(unsigned(G_size))) then
                            if(write_enable(to_integer(unsigned(Ref_Order))-1) = '1' or b_ram_written >=
                                  to_integer(unsigned(Ref_Order))-1) then
                                state <= idle;
                                initialized <= '1';
                            else
                                b_ram_sel <= b_ram_sel(NUM_B_RAM-2 downto 0) & '0';
                                b_ram_written := b_ram_written + 1;
                                counter := 0;
                            end if;
                        end if;
                    end if;

                --The read state should simply read out 1 value from each B_ram
                --each cycle.
                when read =>
                    if(read_enable = '1') then
                        read_address <= read_address + 1;
                        if(read_address >= to_integer(unsigned(G_size))-1) then
                            state <= idle;
                            read_address <= 0;
                        end if;
                    end if;
            end case;
    end if;
end process;


process(b_ram_sel, state, init, valid_input)
begin
    if(state = write and valid_input = '1') then
        write_enable <= b_ram_sel;
    elsif(state = idle and init = '1' and valid_input = '1') then
        write_enable <= (0 => '1', others =>'0');
    else
        write_enable <= (others => '0');
    end if;
end process;

end Behavioral;
```

G.1.5 Block ram bank testbench

```
`timescale 1ns / 1ps

module b_ram_bank_tb;
   parameter C_S_AXI_DATA_WIDTH = 32;
   parameter C_S_AXI_ADDR_WIDTH = 6;
   parameter B_RAM_SIZE = 100;
   parameter B_RAM_BIT_WIDTH = 32;
   parameter NUM_B_RAM = 8;
   parameter PERIOD = 10;

   reg[31:0] in_G_temp;
   reg[31:0] read_holder;

   reg clk, aresetn, read_enable;
   wire enable;
   wire[B_RAM_BIT_WIDTH*NUM_B_RAM-1:0] data_out;

   reg[5:0]   s_axi_ctrl_status_awaddr;
   reg[2:0]   s_axi_ctrl_status_awprot;
   reg        s_axi_ctrl_status_awvalid;
   reg[31:0]  s_axi_ctrl_status_wdata;
   reg[3:0]   s_axi_ctrl_status_wstrb;
   reg        s_axi_ctrl_status_wvalid;
   reg        s_axi_ctrl_status_bready;
   reg[5:0]   s_axi_ctrl_status_araddr;
   reg[2:0]   s_axi_ctrl_status_arprot;
   reg        s_axi_ctrl_status_arvalid;
   reg        s_axi_ctrl_status_rready;
   wire       s_axi_ctrl_status_awready;
   wire       s_axi_ctrl_status_wready;
   wire[1:0]  s_axi_ctrl_status_bresp;
   wire       s_axi_ctrl_status_bvalid;
   wire       s_axi_ctrl_status_arready;
   wire[31:0] s_axi_ctrl_status_rdata;
   wire[1:0]  s_axi_ctrl_status_rresp;
```

```
    wire       s_axi_ctrl_status_rvalid;

    b_ram_bank
         #(.C_S_AXI_DATA_WIDTH(C_S_AXI_DATA_WIDTH),
           .C_S_AXI_ADDR_WIDTH(C_S_AXI_ADDR_WIDTH),
           .B_RAM_SIZE(B_RAM_SIZE),
           .B_RAM_BIT_WIDTH(B_RAM_BIT_WIDTH),
           .NUM_B_RAM(NUM_B_RAM))
    DUT
         (.clk(clk),
          .aresetn(aresetn),
          .read_enable(read_enable),
          .enable(enable),
          .data_out(data_out),
          .s_axi_ctrl_status_awaddr(s_axi_ctrl_status_awaddr),
          .s_axi_ctrl_status_awprot(s_axi_ctrl_status_awprot),
          .s_axi_ctrl_status_awvalid(s_axi_ctrl_status_awvalid),
          .s_axi_ctrl_status_wdata(s_axi_ctrl_status_wdata),
          .s_axi_ctrl_status_wstrb(s_axi_ctrl_status_wstrb),
          .s_axi_ctrl_status_wvalid(s_axi_ctrl_status_wvalid),
          .s_axi_ctrl_status_bready(s_axi_ctrl_status_bready),
          .s_axi_ctrl_status_araddr(s_axi_ctrl_status_araddr),
          .s_axi_ctrl_status_arprot(s_axi_ctrl_status_arprot),
          .s_axi_ctrl_status_arvalid(s_axi_ctrl_status_arvalid),
          .s_axi_ctrl_status_rready(s_axi_ctrl_status_rready),
          .s_axi_ctrl_status_awready(s_axi_ctrl_status_awready),
          .s_axi_ctrl_status_wready(s_axi_ctrl_status_wready),
          .s_axi_ctrl_status_bresp(s_axi_ctrl_status_bresp),
          .s_axi_ctrl_status_bvalid(s_axi_ctrl_status_bvalid),
          .s_axi_ctrl_status_arready(s_axi_ctrl_status_arready),
          .s_axi_ctrl_status_rdata(s_axi_ctrl_status_rdata),
          .s_axi_ctrl_status_rresp(s_axi_ctrl_status_rresp),
          .s_axi_ctrl_status_rvalid(s_axi_ctrl_status_rvalid)
         );


     integer          f_in_G, f_in_raw;
     integer          iter, i;

     always #(PERIOD/2) clk = ~clk;

     initial begin
         clk = 1'b0;
         aresetn = 1'b0;
         read_enable <= 1'b0;

         s_axi_ctrl_status_awprot = 'b0;
         s_axi_ctrl_status_bready = 1'b0;
         s_axi_ctrl_status_wstrb = 4'hF;
         s_axi_ctrl_status_arprot = 'b0;




         f_in_G = $fopen("D:/MasterOppgave/smallsat_prototype/EMSC/in_G.bin", "rb");

         if (f_in_G == 0) begin
             $display("Failed to open input file %s", "D:/MasterOppgave/smallsat_prototype/EMSC/in_G.bin");
             $finish;
         end


         repeat(2) @(posedge clk);
         aresetn = 1'b1;
         write_to_reg(6'h0, 32'h22034);

         repeat(3) @(posedge clk);

         for (iter = 0; iter < 416; iter = iter + 1) begin
               //|
             for( i = 0; i < 4; i = i + 1) begin
                   //|
                 in_G_temp[i*8  +: 8] = $fgetc(f_in_G);
                                                                                 //|
             end
             write_to_reg(6'h4, in_G_temp);

                   //|
             @(posedge clk);

                   //|
         end

         repeat(2)@(posedge clk);
         read_reg(6'h8, read_holder);

         repeat(2)@(posedge clk);
         read_reg(6'h8, read_holder);

         read_enable <= 1'b1;

```

```
        repeat(3)@(posedge clk);

        read_enable <= 1'b0;
    end



     task write_to_reg;
        input [5:0] address;
        input [31:0] data;
        begin
            @(posedge clk);
            s_axi_ctrl_status_awaddr <= address;
            s_axi_ctrl_status_awvalid <= 1'b1;
            s_axi_ctrl_status_wvalid <= 1'b1;
            s_axi_ctrl_status_wdata <= data;

            while (!(s_axi_ctrl_status_awready == 1'b1 && s_axi_ctrl_status_wready == 1'b1)) begin
                @(posedge clk);
            end

            s_axi_ctrl_status_awvalid <= 1'b0;
            s_axi_ctrl_status_wvalid <= 1'b0;
        end
     endtask



     task read_reg;
             input  [5:0] address;
             output [31:0] data;
             begin
                 @(posedge clk);
                 s_axi_ctrl_status_araddr = address;
                 s_axi_ctrl_status_arvalid = 1'b1;
                 s_axi_ctrl_status_rready = 1'b1;
                 while (!(s_axi_ctrl_status_rvalid == 1'b1)) begin
                     @(posedge clk);
                 end

                 s_axi_ctrl_status_rready <= 1'b0;
                 s_axi_ctrl_status_arvalid <= 1'b0;
                 data = s_axi_ctrl_status_rdata;


             end
          endtask




endmodule
```

G.1.6 Block ram design file

```
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;
        data_out        : out std_logic_vector(B_RAM_BIT_WIDTH-1 downto 0)
    );
end block_ram;

architecture Behavioral of block_ram is

signal count_i : integer range 0 to B_RAM_SIZE-1;

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(count_i) <= data_in;
        end if;
    end if;
end process;


process(clk)
begin
    if(rising_edge(clk)) then
        if(aresetn = '0') then
            data_out <= (others => '0');
        elsif(read_enable = '1') then
            data_out <= b_ram_data(read_address);
        end if;
    end if;

end process;

process(clk)
begin
    if(rising_edge(clk)) then
        if(aresetn = '0') then
            count_i <= 0;
        elsif(write_enable = '1') then
            count_i <= count_i + 1;
        end if;
    end if;
end process;
end Behavioral;
```

G.1.7 Dot product module design file

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity dot_product_module is
      Generic(
            RAW_BIT_WIDTH : positive := 12;
            G_BIT_WIDTH   : positive := 32;
            NUM_B_RAM     : positive := 5;
            P_BIT_WIDTH   : positive := 48
      );
      Port (
            clk     :   in  std_logic;
            aresetn :   in  std_logic;
            en      :   in  std_logic;
            in_G    :   in  std_logic_vector(G_BIT_WIDTH*NUM_B_RAM-1 downto 0);
            in_raw  :   in  std_logic_vector(RAW_BIT_WIDTH-1 downto 0);
            v_len   :   in  std_logic_vector(11 downto 0);
            p_rdy   :   out std_logic_vector(NUM_B_RAM-1 downto 0);
            p_out   :   out std_logic_vector(NUM_B_RAM*P_BIT_WIDTH-1 downto 0)
      );
end dot_product_module;

architecture Behavioral of dot_product_module is

begin
--Declaration of dot product cores
dot_prod: for i in 0 to NUM_B_RAM-1 generate
dp: entity work.dot_product
        generic map(
            bit_depth_raw => RAW_BIT_WIDTH,
            bit_depth_G   => G_BIT_WIDTH,
            P_BIT_WIDTH   => P_BIT_WIDTH
        )
        port map(
            clk         =>          clk,
            en          =>          en,
            reset_n     =>          aresetn,
            in_raw      =>          in_raw,
            in_G        =>          in_G(G_BIT_WIDTH*i + G_BIT_WIDTH-1 downto G_BIT_WIDTH*i),
            v_len       =>          v_len,
            p_rdy       =>          p_rdy(i),
            p           =>          p_out(P_BIT_WIDTH*i + P_BIT_WIDTH-1 downto P_BIT_WIDTH*i)
        );
end generate dot_prod;
end Behavioral;
```

G.1.8 Dot product core design file

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

entity dot_product is
    generic(
        bit_depth_raw : positive := 12;
        bit_depth_G   : positive := 32;
        P_BIT_WIDTH   : positive := 48
    );
    Port (
        clk         : in    std_logic;
        en          : in    std_logic;
        reset_n     : in    std_logic;
        in_raw      : in    std_logic_vector(bit_depth_raw-1 downto 0);
        in_G        : in    std_logic_vector(bit_depth_G-1 downto 0);
        v_len       : in    std_logic_vector(11 downto 0);
        p_rdy       : out   std_logic;
        p           : out   std_logic_vector(P_bit_width-1 downto 0)
    );
end dot_product;

architecture Behavioral of dot_product is

signal mul_r  : std_logic_vector((bit_depth_raw + bit_depth_G - 1) downto 0);
signal add_r  : std_logic_vector((bit_depth_raw + bit_depth_G - 1) downto 0);
signal counter : integer range 0 to 400;
signal out_rdy : std_logic;

begin

out_rdy <=  '1' when ((counter = to_integer(unsigned(v_len))+1) and en = '1') else '0';

p <= std_logic_vector(resize(signed(add_r),p'length));
p_rdy <= out_rdy;

process(clk, reset_n)
begin
    if (rising_edge(clk))then
       if(reset_n = '0') then
           mul_r <= (others => '0');
           add_r <= (others => '0');
           counter <= 0;
        elsif(en = '1') then
          counter <= counter + 1;
          --Calculate multiplication between RAW and G.
          mul_r <= std_logic_vector(signed(in_raw)*signed(in_G));
          if(counter = (to_integer(unsigned(v_len)) + 1)) then
              --Initially sett accumulator reg to first multiplication
              --between RAW and G
              add_r <= std_logic_vector(signed(mul_r));
              counter <= 2;
          else
              --Accumulutator reg set to current multiplication
              --between RAW and G added with acummulated result
              add_r <= std_logic_vector(resize(signed(mul_r)+signed(add_r),add_r'length));
          end if;
        end if;
       end if;
end process;
end Behavioral;
```

G.1.9 Output Module design file

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity axi_gearbox is
      Generic(
        B_RAM_SIZE      : integer := 100;
        B_RAM_BIT_WIDTH : integer := 32;
        NUM_B_RAM       : integer := 8;
        RAW_BIT_WIDTH : positive := 16;
        G_BIT_WIDTH   : positive := 32;
        P_BIT_WIDTH   : positive := 48;
        C_S_AXI_DATA_WIDTH : integer := 32;
        C_S_AXI_ADDR_WIDTH : integer := 6
     );
    Port (
        clk     :   in  std_logic;
        aresetn :   in  std_logic;
```

```vhdl
        p_out   :   in  std_logic_vector(P_BIT_WIDTH*NUM_B_RAM-1 downto 0);
        p_rdy   :   in  std_logic;
        enable  :   in  std_logic;
        p_int   :   out std_logic;
        last_p  :   in  std_logic;
        num_pixels : in std_logic_vector(31 downto 0);
        Ref_order : in  std_logic_vector(5 downto 0);

        m_axis_tdata  : out std_logic_vector(63 downto 0);
        --EMSC is ready to send to DMA.
        m_axis_tvalid : out std_logic;
        --DMA is ready to receive data
        m_axis_tready : in  std_logic;
        --Tell DMA this is last data
        m_axis_tlast  : out std_logic
    );
end axi_gearbox;

architecture Behavioral of axi_gearbox is
signal res_mem : std_logic_vector(P_BIT_WIDTH*NUM_B_RAM-1 downto 0);
signal start : std_logic;
signal t_valid_flag : std_logic;
signal out_handshake : std_logic;
signal last_t_valid : std_logic;
begin

--Process to handle AXI-stream output
process(clk,aresetn)
    variable counter : integer range 0 to 50 := 0;
begin
    if(aresetn = '0') then
        m_axis_tdata  <= (others => '0');
        m_axis_tvalid <= '0';
        t_valid_flag <= '0';
        counter := 0;
    elsif(rising_edge(clk)) then
        if(p_rdy = '1') then
            res_mem <= p_out;
            start <= '1';
        elsif(start = '1') then
            if(m_axis_tready = '1' and t_valid_flag = '1') then
                counter := counter + 1;
            end if;
            if(counter >= to_integer(unsigned(Ref_order))) then
                m_axis_tvalid <= '0';
                t_valid_flag <= '0';
                counter := 0;
                start <= '0';
                m_axis_tdata <= (others => '0');
            else
                m_axis_tdata <= std_logic_vector(resize(signed(res_mem((P_BIT_WIDTH*counter + P_BIT_WIDTH-1)
                        downto (P_BIT_WIDTH*counter))),m_axis_tdata'length));
                m_axis_tvalid <= '1';
                t_valid_flag <= '1';
            end if;
        else
            counter := 0;
        end if;

    end if;
end process;

--Process to handle t_last signal
process(clk, aresetn)
    variable counter : integer;
    variable p_last_flag : std_logic;
begin
    if(aresetn = '0') then
        counter := 0;
        m_axis_tlast <= '0';
        p_last_flag := '0';
        last_t_valid <= '0';
    elsif(rising_edge(clk)) then
        last_t_valid <= t_valid_flag;
        if(p_last_flag = '0' and enable = '1') then
            if(t_valid_flag = '1' and m_axis_tready = '1') then
                m_axis_tlast <= '0';
            end if;
            if(counter >= to_integer(unsigned(num_pixels))) then
                p_last_flag := '1';
                counter := 0;
            elsif(p_rdy = '1') then
                counter := counter + 1;
            end if;
        elsif(p_last_flag = '1' and enable = '1') then
            counter := counter + 1;
            if(counter = to_integer(unsigned(Ref_order))-1) then
                m_axis_tlast <= '1';
                counter := 0;
                P_last_flag := '0';
            end if;
        end if;
    end if;
end process;

p_int <= '1' when (last_t_valid = '1' and t_valid_flag = '0') else '0';
end Behavioral;
```

G.1.10 AXI-register interface design file

This is a module designed by Xilinx and found in *LogiCORE IP AXI4-Lite IPIF v2.0* [18]. Some changes was made to make it work with the EMSC application.

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

package B_RAM_BANK_pkg is
    type bus_array is array(natural range <>) of std_logic_vector(31 downto 0);
end package B_RAM_BANK_pkg;

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

entity register_interface is
  generic (
    -- Users to add parameters here
    B_RAM_SIZE : integer := 100;
    B_RAM_BIT_WIDTH : integer := 32;
    NUM_B_RAM       : integer := 5;
    -- User parameters ends
    -- Do not modify the parameters beyond this line

    -- Width of S_AXI data bus
    C_S_AXI_DATA_WIDTH : integer := 32;
    -- Width of S_AXI address bus
    C_S_AXI_ADDR_WIDTH : integer := 6
    );
  port (
    -- Users to add ports here
    emsc2cpu_register : in std_logic_vector(31 downto 0);
    cpu2emsc_register : out std_logic_vector(31 downto 0);
    num_pixels        : out std_logic_vector(31 downto 0);
    in_G_register     : out std_logic_vector(31 downto 0);
    valid_input       : out std_logic;
--    read_enable       : out std_logic;

    -- User ports ends
    -- Do not modify the ports beyond this line

    -- Global Clock Signal
    S_AXI_ACLK    : in  std_logic;
    -- Global Reset Signal. This Signal is Active LOW
    S_AXI_ARESETN : in  std_logic;
    -- Write address (issued by master, acceped by Slave)
    S_AXI_AWADDR  : in  std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
    -- Write channel Protection type. This signal indicates the
    -- privilege and security level of the transaction, and whether
    -- the transaction is a data access or an instruction access.
    S_AXI_AWPROT  : in  std_logic_vector(2 downto 0);
    -- Write address valid. This signal indicates that the master signaling
    -- valid write address and control information.
    S_AXI_AWVALID : in  std_logic;
    -- Write address ready. This signal indicates that the slave is ready
    -- to accept an address and associated control signals.
    S_AXI_AWREADY : out std_logic;
    -- Write data (issued by master, acceped by Slave)
    S_AXI_WDATA   : in  std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    -- Write strobes. This signal indicates which byte lanes hold
    -- valid data. There is one write strobe bit for each eight
    -- bits of the write data bus.
    S_AXI_WSTRB   : in  std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1 downto 0);
    -- Write valid. This signal indicates that valid write
    -- data and strobes are available.
    S_AXI_WVALID  : in  std_logic;
    -- Write ready. This signal indicates that the slave
    -- can accept the write data.
    S_AXI_WREADY  : out std_logic;
    -- Write response. This signal indicates the status
    -- of the write transaction.
    S_AXI_BRESP   : out std_logic_vector(1 downto 0);
    -- Write response valid. This signal indicates that the channel
    -- is signaling a valid write response.
    S_AXI_BVALID  : out std_logic;
    -- Response ready. This signal indicates that the master
    -- can accept a write response.
    S_AXI_BREADY  : in  std_logic;
    -- Read address (issued by master, acceped by Slave)
    S_AXI_ARADDR  : in  std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
    -- Protection type. This signal indicates the privilege
    -- and security level of the transaction, and whether the
    -- transaction is a data access or an instruction access.
    S_AXI_ARPROT  : in  std_logic_vector(2 downto 0);
    -- Read address valid. This signal indicates that the channel
    -- is signaling valid read address and control information.
    S_AXI_ARVALID : in  std_logic;
    -- Read address ready. This signal indicates that the slave is
    -- ready to accept an address and associated control signals.
    S_AXI_ARREADY : out std_logic;
    -- Read data (issued by slave)
    S_AXI_RDATA   : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    -- Read response. This signal indicates the status of the
    -- read transfer.
    S_AXI_RRESP   : out std_logic_vector(1 downto 0);
    -- Read valid. This signal indicates that the channel is
    -- signaling the required read data.
    S_AXI_RVALID  : out std_logic;
    -- Read ready. This signal indicates that the master can
    -- accept the read data and response information.
    S_AXI_RREADY  : in  std_logic
    );
end register_interface;

architecture arch_imp of register_interface is

  -- AXI4LITE signals
  signal axi_awaddr  : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
  signal axi_awready : std_logic;
  signal axi_wready  : std_logic;
  signal axi_bresp   : std_logic_vector(1 downto 0);
  signal axi_bvalid  : std_logic;
  signal axi_araddr  : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
  signal axi_arready : std_logic;
  signal axi_rdata   : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
  signal axi_rresp   : std_logic_vector(1 downto 0);
  signal axi_rvalid  : std_logic;

  -- Example-specific design signals
  -- local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
  -- ADDR_LSB is used for addressing 32/64 bit registers/memories
  -- ADDR_LSB = 2 for 32 bits (n downto 2)
  -- ADDR_LSB = 3 for 64 bits (n downto 3)
  constant ADDR_LSB          : integer := (C_S_AXI_DATA_WIDTH/32)+ 1;
  constant OPT_MEM_ADDR_BITS : integer := 3;
  constant C_NUM_REGS        : integer := 16;
  ------------------------------------------------
  ---- Signals for user logic register space example
  --------------------------------------------------
  ---- Number of Slave Registers 16
  type reg_arr_t is array (0 to C_NUM_REGS) of std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
  signal slv_regs  : reg_arr_t;
  signal read_data : reg_arr_t;

  signal slv_reg_rden : std_logic;
  signal slv_reg_wren : std_logic;
  signal reg_data_out : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
  signal byte_index   : integer;
  signal aw_en        : std_logic;

begin
  -- I/O Connections assignments

  S_AXI_AWREADY <= axi_awready;
  S_AXI_WREADY  <= axi_wready;
  S_AXI_BRESP   <= axi_bresp;
  S_AXI_BVALID  <= axi_bvalid;
  S_AXI_ARREADY <= axi_arready;
  S_AXI_RDATA   <= axi_rdata;
  S_AXI_RRESP   <= axi_rresp;
  S_AXI_RVALID  <= axi_rvalid;
  -- Implement axi_awready generation
  -- axi_awready is asserted for one S_AXI_ACLK clock cycle when both
  -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
  -- de-asserted when reset is low.

  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_awready <= '0';
        aw_en       <= '1';
      else
        if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '1' and aw_en = '1') then
          -- slave is ready to accept write address when
          -- there is a valid write address and write data
          -- on the write address and data bus. This design
          -- expects no outstanding transactions.
          axi_awready <= '1';
        elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then
          aw_en       <= '1';
          axi_awready <= '0';
        else
          axi_awready <= '0';
        end if;
      end if;
    end if;
  end process;

  -- Implement axi_awaddr latching
  -- This process is used to latch the address when both
  -- S_AXI_AWVALID and S_AXI_WVALID are valid.

  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_awaddr <= (others => '0');
      else
        if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '1' and aw_en = '1') then
          -- Write Address latching
          axi_awaddr <= S_AXI_AWADDR;
        end if;
      end if;
    end if;
  end process;

  -- Implement axi_wready generation
  -- axi_wready is asserted for one S_AXI_ACLK clock cycle when both
  -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
  -- de-asserted when reset is low.

  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_wready <= '0';
      else
        if (axi_wready = '0' and S_AXI_WVALID = '1' and S_AXI_AWVALID = '1' and aw_en = '1') then
          -- slave is ready to accept write data when
          -- there is a valid write address and write data
          -- on the write address and data bus. This design
          -- expects no outstanding transactions.
          axi_wready <= '1';
        else
          axi_wready <= '0';
        end if;
      end if;
    end if;
  end process;

  -- Implement memory mapped register select and write logic generation
  -- The write data is accepted and written to memory mapped registers when
  -- axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write strobes are used to
  -- select byte enables of slave registers while writing.
  -- These registers are cleared when reset (active low) is applied.
  -- Slave register write enable is asserted when valid address and data are available
  -- and the slave is ready to accept the write address and write data.
  slv_reg_wren <= axi_wready and S_AXI_WVALID and axi_awready and S_AXI_AWVALID;

  process (S_AXI_ACLK)
    variable loc_addr     : integer range 0 to 2**(OPT_MEM_ADDR_BITS+1)-1;
    variable slv_regs_nxt : reg_arr_t;
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        slv_regs <= (others => (others => '0'));
        valid_input <= '0';
      else
        loc_addr := to_integer(unsigned(axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB)));
        valid_input <= '0';
        if (slv_reg_wren = '1') then
            if(loc_addr = 1) then
                valid_input <= '1';
            end if;
          if (loc_addr < C_NUM_REGS) then
            for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
              if (S_AXI_WSTRB(byte_index) = '1') then
                -- Respective byte enables are asserted as per write strobes
                -- slave registor 0
                slv_regs(loc_addr)(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7 downto
                      byte_index*8);
              end if;
            end loop;
          end if;
        end if;
      end if;
    end if;
  end process;

  -- Implement write response logic generation
  -- The write response and response valid signals are asserted by the slave
  -- when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
  -- This marks the acceptance of address and indicates the status of
  -- write transaction.

  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_bvalid <= '0';
        axi_bresp  <= "00";              --need to work more on the responses
      else
        if (axi_awready = '1' and S_AXI_AWVALID = '1' and axi_wready = '1' and S_AXI_WVALID = '1' and
              axi_bvalid = '0') then
          axi_bvalid <= '1';
          axi_bresp  <= "00";
        elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then  --check if bready is asserted while bvalid is
              high)
          axi_bvalid <= '0';  -- (there is a possibility that bready is always asserted high)
        end if;
      end if;
    end if;
  end process;

  -- Implement axi_arready generation
  -- axi_arready is asserted for one S_AXI_ACLK clock cycle when
  -- S_AXI_ARVALID is asserted. axi_awready is
  -- de-asserted when reset (active low) is asserted.
  -- The read address is also latched when S_AXI_ARVALID is
  -- asserted. axi_araddr is reset to zero on reset assertion.

  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_arready <= '0';
        axi_araddr  <= (others => '1');
      else
        if (axi_arready = '0' and S_AXI_ARVALID = '1') then
          -- indicates that the slave has acceped the valid read address
          axi_arready <= '1';
          -- Read Address latching
          axi_araddr  <= S_AXI_ARADDR;
        else
          axi_arready <= '0';
        end if;
      end if;
    end if;
  end process;

  -- Implement axi_arvalid generation
  -- axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
  -- S_AXI_ARVALID and axi_arready are asserted. The slave registers
  -- data are available on the axi_rdata bus at this instance. The
  -- assertion of axi_rvalid marks the validity of read data on the
  -- bus and axi_rresp indicates the status of read transaction.axi_rvalid
  -- is deasserted on reset (active low). axi_rresp and axi_rdata are
  -- cleared to zero on reset (active low).
  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_rvalid <= '0';
        axi_rresp  <= "00";
--        read_enable <= '0';
      else
        if (axi_arready = '1' and S_AXI_ARVALID = '1' and axi_rvalid = '0') then
          -- Valid read data is available at the read data bus
          axi_rvalid <= '1';
          axi_rresp  <= "00";           -- 'OKAY' response
        elsif (axi_rvalid = '1' and S_AXI_RREADY = '1') then
          -- Read data is accepted by the master
--          read_enable <= '1';
          axi_rvalid <= '0';
--        else
--            read_enable <= '0';
        end if;
      end if;
    end if;
  end process;

  -- Implement memory mapped register select and read logic generation
  -- Slave register read enable is asserted when valid address is available
  -- and the slave is ready to accept the read address.
  slv_reg_rden <= axi_arready and S_AXI_ARVALID and (not axi_rvalid);

  process (axi_araddr, read_data)
```

```
    variable loc_addr : integer range 0 to 2**(OPT_MEM_ADDR_BITS+1)-1;
  begin
    -- Address decoding for reading registers
    loc_addr := to_integer(unsigned(axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB)));
--        read_enable <= '0';
--    if(loc_addr = 0) then
--        read_enable <= '1';
--    end if;
    if (loc_addr < C_NUM_REGS) then
      reg_data_out <= read_data(loc_addr);
    else
      reg_data_out <= (others => '0');
    end if;
  end process;

  -- Output register or memory read data
  process(S_AXI_ACLK) is
  begin
    if (rising_edge (S_AXI_ACLK)) then
      if (S_AXI_ARESETN = '0') then
        axi_rdata <= (others => '0');
      else
        if (slv_reg_rden = '1') then
          -- When there is a valid read address (S_AXI_ARVALID) with
          -- acceptance of read address by the slave (axi_arready),
          -- output the read dada
          -- Read address mux
          axi_rdata <= reg_data_out;    -- register read data
        end if;
      end if;
    end if;
  end process;

  -- Add user logic here
  cpu2emsc_register <= slv_regs(0);
  in_G_register <= slv_regs(1);
  num_pixels <= slv_regs(2);

  -- Data returned when reading is the register values -- except for the cases
  -- where we want reads to behave differently
  process (slv_regs, emsc2cpu_register)
  begin
    for i in 0 to C_NUM_REGS-1 loop
      read_data(i) <= slv_regs(i);
    end loop;
    read_data(1) <= emsc2cpu_register;
  end process;
  -- User logic ends

end arch_imp;
```

G.2 Parallel Implementation

G.2.1 Software

Listing 3: C++ code using listings

```
#include <stdio.h>
#include "xil_printf.h"     //Printf for Uart
#include "Eigen/dense"      //Eigen
#include <stdlib.h>         //atof
#include <math.h>           //Pow, sqrt
#include <float.h>
#include "xparameters.h"    //Board specific parameters
#include "xuartps.h"        //Uart
#include <string.h>
#include "xtmrctr.h"        //Axi Timer

//Interrupt
#include "xscugic.h"
#include "xil_exception.h"


//Axi Timer
#include "xtmrctr.h"

#include <string.h>


using Eigen::MatrixXd;


//Axi timer
```

```
#define TMRCTR_DEVICE_ID  XPAR_TMRCTR_0_DEVICE_ID
#define TIMER_COUNTER_0  0
XTmrCtr TimerCounter;

//Uart
#define UART_DEVICE_ID      XPAR_PS7_UART_1_DEVICE_ID
XUartPs Uart_Ps;

//Interrupt
bool mm2s_complete = false;
bool s2mm_complete = false;
const u32 MM2S_INT = 61U;
const u32 S2MM_INT = 62U;
const u32 P_INT = 63U;
XScuGic_Config* scugic_config;
XScuGic scugic_inst;
int int_counter = 0;
u32* dma_regs = (u32*)0x43C00000;




/*Function Prototypes ******************************/
void EMSC(double ** ref_spectra,
          double ** corrected, int nVars,
          int nObs, int refOrder);
double ** initialize(int rows, int columns);
int init_timer(u16 DeviceId, u8 TmrCtrNumber);
u32 start_timer(u8 TmrCtrNumber);
u32 stop_timer(u8 TmrCtrNumber);
static void dma_irq_handler(void* ref);
static void p_int_irq_handler(void* ref);

/**************************************************/

double ** initialize(int rows, int columns) {
    double **temp;
    temp = (double **)malloc(rows * sizeof(double*));
    for (int row = 0; row < rows; row++) {
        temp[row] = (double*)malloc(columns * sizeof(double));
    }
    return temp;
}


static void p_int_irq_handler(void* ref){
    int_counter += 1;
}

static void dma_irq_handler(void* ref) {
    int instance = (int)ref;
    int status_reg;
    u32 mask = 0;

    if (instance == 0) {
        status_reg = 1;
        mm2s_complete = true;
    }
    else{
        status_reg = 9;
        s2mm_complete = true;
    }
    //else{
    //  int_counter = int_counter + 1;
    //}
    mask = dma_regs[status_reg];
    dma_regs[status_reg] = (1 << 5);
}


int init_interrupt_system(){

     //Initialize Interrupt system
    //---------------------------------------------------------------------------

    int ret;


    scugic_config = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
    if(NULL == scugic_config){
        return XST_FAILURE;
    }
    ret = XScuGic_CfgInitialize(&scugic_inst, scugic_config, scugic_config->CpuBaseAddress);
    if (ret != XST_SUCCESS) {
        print("Failed to initialize GIC\n");
        return ret;
    }



    u32 id_full = XScuGic_CPUReadReg(&scugic_inst, XSCUGIC_INT_ACK_OFFSET);
    XScuGic_CPUWriteReg(&scugic_inst, XSCUGIC_EOI_OFFSET, id_full);

    ret = XScuGic_SelfTest(&scugic_inst);
    if (ret != XST_SUCCESS) {
        return XST_FAILURE;
    }


   ret = XScuGic_Connect(&scugic_inst, MM2S_INT, (Xil_InterruptHandler)dma_irq_handler, (void*)0);
       if (ret != XST_SUCCESS)
           return ret;

   ret = XScuGic_Connect(&scugic_inst, S2MM_INT, (Xil_InterruptHandler)dma_irq_handler, (void*)1);
       if (ret != XST_SUCCESS)
           return ret;

    //ret = XScuGic_Connect(&scugic_inst, P_INT, (Xil_InterruptHandler)p_int_irq_handler, (void*)2);
    if (ret != XST_SUCCESS){
            print("Failed to initialize GIC 3\n");
            return ret;
    }

    XScuGic_SetPriorityTriggerType(&scugic_inst, MM2S_INT, 0xA0, 0x3);
    XScuGic_SetPriorityTriggerType(&scugic_inst, S2MM_INT, 0xA0, 0x3);
    //XScuGic_SetPriorityTriggerType(&scugic_inst, P_INT, 0xA0, 0x3);

    XScuGic_Enable(&scugic_inst, MM2S_INT);
    XScuGic_Enable(&scugic_inst, S2MM_INT);
    //XScuGic_Enable(&scugic_inst, P_INT);


    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                (Xil_ExceptionHandler)XScuGic_InterruptHandler,
                                 &scugic_inst);
    Xil_ExceptionEnable();

    //--------------------------------------------------------------------------
    return XST_SUCCESS;
}




void EMSC(double** ref_spectra, double* mean_spectra, double** corrected, int nVars, int nObs, int refOrder)
       {
    //DECLARATIONS---------------------
    MatrixXd M(refOrder + 4, nVars);
    //double ** G = initialize(nVars, refOrder+4);
    double ** corr_M = initialize(2, nVars);
    double num = 0;
    double multiplier = pow(2.0, 20.0);
    //----------------------------------


    //----------------------------------------------------
    xil_printf("Constructing M!\n");
    for (int i = 0; i < nVars; i++) {

        //Add 1 in first row
        M(0,i) = 1;

        //Add linspace and linspace squared
        M(1,i) = num;
        corr_M[0][i] = num;

        M(2,i) = pow(num, 2);
        corr_M[1][i] = pow(num,2);
        num += (1.0 / (nVars - 1));

        //Add reference spectra
        for (int y = 0; y < refOrder; y++) {
            M(y + 3,i) = ref_spectra[y][i];
        }

        //Add mean in last row
        M(refOrder+3,i) = mean_spectra[i];
    }




    //INITIALIZE BLOCK RAM
    //  - G is loaded into the b-ram
    //------------------------------
    //u32 * mem_ptr = (u32*)0x10000000;
    u32 * init = (u32*)0x43c10000;
    u32 * in_G = (u32*)0x43c10004;
    u32 * num_pixels = (u32*)0x43c10008;

    //
    *init = 0x22034;
    *num_pixels = 0x3D090;
    //Execute pseudo-inverse of M
    MatrixXd M_M = M*M.transpose();
    MatrixXd p_inv = M.transpose() * M_M.completeOrthogonalDecomposition().pseudoInverse();
    xil_printf("Pseudo-Inverse Completed!\n");
    for(int y = 0; y<refOrder+4; y++){
        for(int i = 0; i<nVars; i++){
            *in_G  =(int) floor(p_inv(i,y)* multiplier);
            //save_G[index++] = floor(p_inv(i,y)* multiplier);
        }

    }
    //-------------------------------

    *init = 0x20034;
    *init = 0x21034;


    //Initiate and enable Cube DMA
    //----------------------------------------------
    u32* mm2s = (u32*)0x43c00000;
    u32* s2mm = (u32*)0x43c00020;

    // Program S2MM DMA

    s2mm[0] = 0x0;
    s2mm[2] = 0x0F0BDBF0;
    s2mm[0] = (1 << 5) | 1;


    // Program MM2S DMA
    mm2s[0] = 0;
    mm2s[2] = 0x100010E0;

    //mm2s[3] = 0x1001001; //small cube
    //mm2s[5] = 520000; //small cube

    mm2s[3] = 0x341F41F4; //large cube  |52|500|500
    mm2s[5] = 0x6590;//large cube      |26000| 52*500

    u32 value1, value2;
    init_timer(TMRCTR_DEVICE_ID, TIMER_COUNTER_0);
    value1 = start_timer(TIMER_COUNTER_0);

    mm2s[0] = (1 << 8) | (1 << 5) | 1;
    while (!s2mm_complete || !mm2s_complete);
    //while ((mm2s[1] != 0x1) || (s2mm[1] != 0x3D090001));
    //while ((mm2s[1] != 0x1) || (s2mm[1] != 0x3D081201));
    //----------------------------------------------

    value2 = stop_timer(TIMER_COUNTER_0);
    xil_printf("Timer: %d\n", value2-value1);
    xil_printf("Counter: %d\n", int_counter);


    //Calculate the corrected spectra
    //----------------------------------------
    int64_t * test_ptr = (int64_t*)0x0F0BDBF0;
    u16 * raw_ptr = (u16*)0x100010E0;
    double p_st[8];
    u16 ah;
    int counter = 0;
    float* mem_ptr = (float*)0x19CD1534;
    int index = 0;
    //----------------------------------------



    while(counter < nObs){
            for(int i = 0; i < 8; i++){
                p_st[i] = test_ptr[i+counter*8]/multiplier;
            }
            for(int cols = 0; cols < nVars; cols++){
                ah = raw_ptr[counter*52+cols];
                corrected[counter][cols] = (ah - (p_st[0] + p_st[1]*corr_M[0][cols] + p_st[2]*corr_M[1][cols
                      ]))/p_st[refOrder+3];
            }
            counter++;
    }
    xil_printf("Test!\n");

    return;
}



int init_timer(u16 DeviceId, u8 TmrCtrNumber){
    int Status;
        XTmrCtr *TmrCtrInstancePtr = &TimerCounter;
        /*
```

```
         * Initialize the timer counter so that it's ready to use,
         * specify the device ID that is generated in xparameters.h
         */
        Status = XTmrCtr_Initialize(TmrCtrInstancePtr, DeviceId);
        if (Status != XST_SUCCESS) {
            printf("Timer failed\n");
            return XST_FAILURE;
        }
        /*
         * Perform a self-test to ensure that the hardware was built
         * correctly, use the 1st timer in the device (0)
         */
        Status = XTmrCtr_SelfTest(TmrCtrInstancePtr, TmrCtrNumber);
        if (Status != XST_SUCCESS) {
            printf("Timer failed\n");
            return XST_FAILURE;
        }
        /*
        * Enable the Autoreload mode of the timer counters.
        */
        return XST_SUCCESS;}

u32 start_timer(u8 TmrCtrNumber){
    XTmrCtr *TmrCtrInstancePtr = &TimerCounter;
    XTmrCtr_SetOptions(TmrCtrInstancePtr, TmrCtrNumber,
                            XTC_AUTO_RELOAD_OPTION);
    u32 val = XTmrCtr_GetValue(TmrCtrInstancePtr, TmrCtrNumber);
    XTmrCtr_Start(TmrCtrInstancePtr, TmrCtrNumber);
    return val;}

u32 stop_timer(u8 TmrCtrNumber){
    XTmrCtr *TmrCtrInstancePtr = &TimerCounter;
    u32 val = XTmrCtr_GetValue(TmrCtrInstancePtr, TmrCtrNumber);
    XTmrCtr_SetOptions(TmrCtrInstancePtr, TmrCtrNumber, 0);
    return val;
}


int main(){
    init_interrupt_system();


    //Adding pointer to location of stored cube.
    float * mem_ptr = (float*)0x10010000;
    int nVars = 52; //number of wavelenghts
    int nObs  = 250000; //total number of pixels
    int refOrder = 4; //numbers of species in spectra


    double ** ref_spectra = initialize(refOrder, nVars);
    double ** corrected = initialize(nObs, nVars);
    double * mean_v =  (double*)malloc(nVars * sizeof(double));

    //Fill raw matrix
    int index = 0;


    //Construct some reference spectra
    //Just using some spectras from raw in this case
    //as an example.
    index = 0;
    mem_ptr = (float*)0x10000000;
    for(int rows = 0; rows<refOrder; rows++){
        for(int cols = 0; cols < nVars; cols++){
            ref_spectra[rows][cols] = mem_ptr[index++];
        }
    }

    mem_ptr = (float*)0x10001000;
    for(int i = 0; i<nVars; i++){
        mean_v[i] = mem_ptr[i];
    }
    //calculate mean of ref_spectra
    //mean(ref_spectra, mean_v, nVars, refOrder);


    xil_printf("ESMC Starting!\n");

    //Start the EMSC



    EMSC(ref_spectra, mean_v, corrected, nVars, nObs, refOrder);


    //Stop timer



    //Point to location for storing data
    xil_printf("Done");



    XScuGic_Disable(&scugic_inst, MM2S_INT);
    XScuGic_Disable(&scugic_inst, S2MM_INT);
    //XScuGic_Disable(&scugic_inst, P_INT);

    XScuGic_Disconnect(&scugic_inst, MM2S_INT);
    XScuGic_Disconnect(&scugic_inst, S2MM_INT);
    //XScuGic_Disconnect(&scugic_inst, P_INT);
    return 0;
}
```

G.2.2 Top module design file

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;




entity top is
    Generic(
        B_RAM_SIZE      : integer := 400;
        NUM_B_RAM       : integer := 16;
        RAW_BIT_WIDTH : positive := 64;
        G_BIT_WIDTH   : positive := 32;
        P_BIT_WIDTH   : positive := 64;
        C_S_AXI_DATA_WIDTH : integer := 32;
        C_S_AXI_ADDR_WIDTH : integer := 6;

        FIFO_DEPTH : integer := 512;
        FIFO_SIZE : integer := 64;
        WRITE_DATA_WIDTH : integer := 64;
        WR_DATA_COUNT_WIDTH : integer := 7;
        FIFO_MARGIN : integer := 50;
        RD_DATA_COUNT_WIDTH : integer := 7;
        READ_DATA_WIDTH : integer := 16;
        LATENCY_CYCLES  : integer := 13
    );
   Port (
        clk     :   in std_logic;
        aresetn :   in std_logic;
        p_irq   :   out std_logic;

        --AXI in-stream
        s_axis_tdata  : in  std_logic_vector(RAW_BIT_WIDTH-1 downto 0);
        --DMA is ready to send data
        s_axis_tvalid : in  std_logic;
        --EMSC is ready to receive data
        s_axis_tready : out std_logic;
        --DMA say this is last data
        s_axis_tlast  : in  std_logic;

        --AXI out-stream
        m_axis_tdata  : out std_logic_vector(63 downto 0);
        --EMSC is ready to send to DMA.
        m_axis_tvalid : out std_logic;
        --DMA is ready to receive data
        m_axis_tready : in  std_logic;
        --Tell DMA this is last data
        m_axis_tlast  : out std_logic;


        -- Register interface
        s_axi_ctrl_status_awaddr  : in  std_logic_vector(5 downto 0);
        s_axi_ctrl_status_awprot  : in  std_logic_vector(2 downto 0);
        s_axi_ctrl_status_awvalid : in  std_logic;
        s_axi_ctrl_status_awready : out std_logic;
        s_axi_ctrl_status_wdata   : in  std_logic_vector(31 downto 0);
        s_axi_ctrl_status_wstrb   : in  std_logic_vector(3 downto 0);
        s_axi_ctrl_status_wvalid  : in  std_logic;
        s_axi_ctrl_status_wready  : out std_logic;
        s_axi_ctrl_status_bresp   : out std_logic_vector(1 downto 0);
        s_axi_ctrl_status_bvalid  : out std_logic;
        s_axi_ctrl_status_bready  : in  std_logic;
        s_axi_ctrl_status_araddr  : in  std_logic_vector(5 downto 0);
        s_axi_ctrl_status_arprot  : in  std_logic_vector(2 downto 0);
        s_axi_ctrl_status_arvalid : in  std_logic;
        s_axi_ctrl_status_arready : out std_logic;
        s_axi_ctrl_status_rdata   : out std_logic_vector(31 downto 0);
        s_axi_ctrl_status_rresp   : out std_logic_vector(1 downto 0);
        s_axi_ctrl_status_rvalid  : out std_logic;
        s_axi_ctrl_status_rready  : in  std_logic
   );
end top;

architecture Behavioral of top is

-- AXI in-stream signals
signal in_stream_data  : std_logic_vector(RAW_BIT_WIDTH-1 downto 0);
signal in_stream_valid : std_logic;
signal in_stream_ready : std_logic;
signal in_stream_last  : std_logic;
signal in_stream_handshake : std_logic;
signal in_stream_valid_delay : std_logic;
signal in_stream_ready_delay : std_logic;
signal in_raw_delay :   std_logic_vector(RAW_BIT_WIDTH-1 downto 0);


-- AXI out-stream signals
signal out_stream_data  : std_logic_vector(63 downto 0);
signal out_stream_valid : std_logic;
signal out_stream_ready : std_logic;
signal out_stream_last  : std_logic;
signal out_stream_handshake : std_logic;

-- Signals from/to b_ram_bank
signal read_enable : std_logic;
signal b_ram_out   : std_logic_vector(G_BIT_WIDTH*NUM_B_RAM-1 downto 0);
signal enable      : std_logic;
signal v_len       : std_logic_vector(11 downto 0);
signal Ref_order   : std_logic_vector(5 downto 0);
signal num_pixels  : std_logic_vector(31 downto 0);
signal initialized : std_logic;

-- Signals from/to dot_product_module
signal p_rdy    :   std_logic_vector(3 downto 0);
signal p_out    :   std_logic_vector(4*NUM_B_RAM*P_BIT_WIDTH-1 downto 0);
signal dp_extend_end : std_logic;
signal dp_enable : std_logic_vector(3 downto 0);
signal dp_data_in : std_logic_vector(G_BIT_WIDTH*NUM_B_RAM*4-1 downto 0);
signal dp_enable_reg : std_logic_vector(3*LATENCY_CYCLES downto 0);
-- Signals from/to AXI gear box
signal last_p : std_logic;

--Signal for fifo
signal fifo_enable: std_logic;
signal fifo_read: std_logic_vector(3 downto 0);
signal fifo_in : std_logic_vector(63 downto 0);
signal fifo_out : std_logic_vector(16*4-1 downto 0);
signal fifo_empty, fifo_full : std_logic_vector(3 downto 0);

--Signal FIFO delay Registers
signal fifo1_reg : std_logic_vector(NUM_B_RAM*G_BIT_WIDTH*(LATENCY_CYCLES)-1 downto 0);
signal fifo2_reg : std_logic_vector(NUM_B_RAM*G_BIT_WIDTH*((2*LATENCY_CYCLES)-1)-1 downto 0);
signal fifo3_reg : std_logic_vector(NUM_B_RAM*G_BIT_WIDTH*((3*LATENCY_CYCLES)-2)-1 downto 0);

signal last_flag : std_logic;
signal valid_in : std_logic;
signal p_irq_w : std_logic;


signal b_ram_register_timing_opt : std_logic_vector(G_BIT_WIDTH*NUM_B_RAM-1 downto 0);
signal fifo_register_timing_opt  : std_logic_vector(16*4-1 downto 0);

signal fifo_init : std_logic_vector(3 downto 0);
signal prev_fifo_init : std_logic_vector(3 downto 0);
signal dp_enable_delay : std_logic_vector(3 downto 0);
signal fifo_init_delay : std_logic_vector(3 downto 0);
signal in_stream_counter : integer range 0 to 20000000;
signal fifo_full_flag : std_logic;
signal last_p_delay : std_logic_vector(3 downto 0);

TYPE State_type IS (INITIAL, CONTINOUS);  -- Define the states
SIGNAL state : State_Type;
begin

--Connections
in_stream_data  <= s_axis_tdata;
in_stream_valid <= s_axis_tvalid;
s_axis_tready   <= in_stream_ready;
in_stream_last  <= s_axis_tlast;


--Helper signal
in_stream_handshake <= '1' when (in_stream_valid = '1' and in_stream_ready = '1') else '0';

valid_in <= '1' when (in_stream_handshake = '1' and m_axis_tready = '1' and enable = '1' and initialized =
      '1') else '0';
fifo_enable <= '1' when ( in_stream_valid_delay = '1' and in_stream_ready_delay = '1' and enable = '1' and
      initialized = '1') or last_flag = '1' else '0';
```

```vhdl
fifo_full_flag <= '1' when (fifo_full(0) = '1' or fifo_full(1) = '1' or fifo_full(2) = '1'  or fifo_full(3)
       = '1') else '0';

p_irq <= p_irq_w;


--Process to control last value on
--AXI-stream
process(clk, aresetn)
    variable p_count : integer := 0;
begin
    if(aresetn = '0') then
        b_ram_register_timing_opt <= (others => '0');
        fifo_register_timing_opt <= (others => '0');
        p_count := 0;
        last_p <= '0';
        last_p_delay <= "0000";
    elsif(rising_edge(clk)) then
        last_p_delay <= p_rdy;
        b_ram_register_timing_opt <= b_ram_out;
        fifo_register_timing_opt <= fifo_out;
        if(last_p_delay(0) = '1' or last_p_delay(1) = '1' or last_p_delay(2) = '1' or last_p_delay(3) = '1')
                then
            p_count := p_count + 1;
        end if;
        if(p_count = (to_integer(unsigned(num_pixels)))+1) then
            last_p <= '1';
            p_count := 0;
        end if;
    end if;
end process;


read_enable <= '1' when fifo_init_delay(0) = '1' else '0';
fifo_read <= fifo_init_delay;

--Process to control enable signals for
--FIFO, Block ram and dot product
--modules
process(clk, aresetn)
    variable counter : integer;
    variable last_counter : integer;
    variable fifo : integer;
    variable invalid_flag : std_logic;
begin
    if(aresetn = '0') then
        counter := 0;
        last_counter := 0;
        fifo_init <= "0000";
        prev_fifo_init <= "0000";
        fifo := 0;
        last_flag <= '0';
        fifo_init_delay <= (others => '0');
        dp_enable_delay <= (others => '0');
    elsif(rising_edge(clk)) then
        fifo_init_delay <= fifo_init;
        dp_enable_delay <= fifo_init_delay;
        dp_enable <= dp_enable_delay;
        --last_p <= '0';
        if(in_stream_last  = '1') then
            last_flag <= '1';
        end if;
        if(valid_in = '1' or fifo_full_flag = '1') then
            if(invalid_flag = '1') then
                fifo_init <= prev_fifo_init;
                invalid_flag := '0';
            end if;
            counter := counter + 1;
            if(counter = LATENCY_CYCLES-1) then
                fifo_init(fifo) <= '1';
                fifo := fifo + 1;
                if(fifo > 3) then
                    fifo := 0;
                end if;
                counter := 0;
            end if;
        elsif(last_flag = '1') then
            fifo_init <= "1111";
            last_counter := last_counter + 1;
            if(last_p = '1') then
                last_flag <= '0';
                last_counter := 0;
            end if;
        else
            if(invalid_flag = '0') then
                prev_fifo_init <= fifo_init;
            end if;
            invalid_flag := '1';
            fifo_init <= (others => '0');
        end if;
    end if;
end process;

-- Controls delay register from block ram
dp_data_in(NUM_B_RAM*G_BIT_WIDTH-1 downto 0) <= b_ram_register_timing_opt;
dp_data_in(2*NUM_B_RAM*G_BIT_WIDTH-1 downto NUM_B_RAM*G_BIT_WIDTH) <= fifo1_reg(NUM_B_RAM*G_BIT_WIDTH*(
       LATENCY_CYCLES)-G_BIT_WIDTH*NUM_B_RAM-1 downto NUM_B_RAM*G_BIT_WIDTH*(LATENCY_CYCLES)-G_BIT_WIDTH*
       NUM_B_RAM-(NUM_B_RAM*G_BIT_WIDTH));
dp_data_in(3*NUM_B_RAM*G_BIT_WIDTH-1 downto 2*NUM_B_RAM*G_BIT_WIDTH) <= fifo2_reg(NUM_B_RAM*G_BIT_WIDTH*((2*
       LATENCY_CYCLES)-1)-G_BIT_WIDTH*NUM_B_RAM-1 downto NUM_B_RAM*G_BIT_WIDTH*((2*LATENCY_CYCLES)-1)-
       G_BIT_WIDTH*NUM_B_RAM-(NUM_B_RAM*G_BIT_WIDTH));
dp_data_in(4*NUM_B_RAM*G_BIT_WIDTH-1 downto 3*NUM_B_RAM*G_BIT_WIDTH) <= fifo3_reg(NUM_B_RAM*G_BIT_WIDTH*((3*
       LATENCY_CYCLES)-2)-G_BIT_WIDTH*NUM_B_RAM-1 downto NUM_B_RAM*G_BIT_WIDTH*((3*LATENCY_CYCLES)-2)-
       G_BIT_WIDTH*NUM_B_RAM-(NUM_B_RAM*G_BIT_WIDTH));
process(clk, aresetn)
begin
    if(aresetn = '0') then
        fifo1_reg <= (others => '0');
        fifo2_reg <= (others => '0');
        fifo3_reg <= (others => '0');
    elsif(rising_edge(clk)) then
        if (dp_enable(0) = '1') then
            fifo1_reg <= fifo1_reg(NUM_B_RAM*G_BIT_WIDTH*(LATENCY_CYCLES)-G_BIT_WIDTH*NUM_B_RAM-1 downto 0)
                   & b_ram_register_timing_opt;
            fifo2_reg <= fifo2_reg(NUM_B_RAM*G_BIT_WIDTH*((2*LATENCY_CYCLES)-1)-G_BIT_WIDTH*NUM_B_RAM-1
                   downto 0) & b_ram_register_timing_opt;
            fifo3_reg <= fifo3_reg(NUM_B_RAM*G_BIT_WIDTH*((3*LATENCY_CYCLES)-2)-G_BIT_WIDTH*NUM_B_RAM-1
                   downto 0) & b_ram_register_timing_opt;
        end if;
    end if;
end process;

--Process for delaying raw stream and
--valid_in signal
process(clk, aresetn)
    variable counter : integer := 0;
begin
    if (aresetn = '0') then
        in_stream_ready <= '0';
        in_stream_ready_delay <= '0';
        in_stream_valid_delay <= '0';
        counter := 0;
        in_raw_delay <= (others => '0');
    elsif(rising_edge(clk)) then
        in_raw_delay <= in_stream_data;
        in_stream_valid_delay <= in_stream_valid;
        in_stream_ready_delay <= in_stream_ready;
        if(enable = '1' and initialized = '1' and m_axis_tready = '1' and fifo_full_flag = '0') then
            in_stream_ready <= '1';
        else
            in_stream_ready <= '0';
        end if;
    end if;
end process;


--FIFO module declaration
fifo: entity work.fifo_module
    generic map(
        FIFO_DEPTH          =>  FIFO_DEPTH,
        FIFO_SIZE           =>  FIFO_SIZE,
        WRITE_DATA_WIDTH    =>  WRITE_DATA_WIDTH,
        WR_DATA_COUNT_WIDTH =>  WR_DATA_COUNT_WIDTH,
        FIFO_MARGIN         =>  FIFO_MARGIN,
        RD_DATA_COUNT_WIDTH =>  RD_DATA_COUNT_WIDTH,
        READ_DATA_WIDTH     =>  READ_DATA_WIDTH,
        LATENCY_CYCLES      =>  LATENCY_CYCLES
    )
    port map(
        clk         =>  clk,
        aresetn     =>  aresetn,
        enable      =>  fifo_enable,
        fifo_read   =>  fifo_read,
        fifo_in     =>  in_stream_data,
        fifo_out    =>  fifo_out,
        fifo_empty  =>  fifo_empty,
        fifo_full   =>  fifo_full
    );

--Block ram module declaration
b_ram: entity work.b_ram_bank
  Generic map(
          B_RAM_SIZE         =>  B_RAM_SIZE,
          B_RAM_BIT_WIDTH    =>  G_BIT_WIDTH,
          NUM_B_RAM          =>  NUM_B_RAM,
          C_S_AXI_DATA_WIDTH =>  C_S_AXI_DATA_WIDTH,
          C_S_AXI_ADDR_WIDTH =>  C_S_AXI_ADDR_WIDTH
       )
  Port map(
        clk             =>  clk,
        aresetn         =>  aresetn,

        -- B_ram interface
        read_enable     =>  read_enable,

        data_out        =>  b_ram_out,
        v_len           =>  v_len,
        R_order         =>  Ref_Order,
        num_pixels      =>  num_pixels,
        init_flag       =>  initialized,


        -- Register interface
        enable          =>  enable,
        s_axi_ctrl_status_awaddr    =>  s_axi_ctrl_status_awaddr,
        s_axi_ctrl_status_awprot    =>  s_axi_ctrl_status_awprot,
        s_axi_ctrl_status_awvalid   =>  s_axi_ctrl_status_awvalid,
        s_axi_ctrl_status_awready   =>  s_axi_ctrl_status_awready,
        s_axi_ctrl_status_wdata     =>  s_axi_ctrl_status_wdata,
        s_axi_ctrl_status_wstrb     =>  s_axi_ctrl_status_wstrb,
        s_axi_ctrl_status_wvalid    =>  s_axi_ctrl_status_wvalid,
        s_axi_ctrl_status_wready    =>  s_axi_ctrl_status_wready,
        s_axi_ctrl_status_bresp     =>  s_axi_ctrl_status_bresp,
        s_axi_ctrl_status_bvalid    =>  s_axi_ctrl_status_bvalid,
        s_axi_ctrl_status_bready    =>  s_axi_ctrl_status_bready,
        s_axi_ctrl_status_araddr    =>  s_axi_ctrl_status_araddr,
        s_axi_ctrl_status_arprot    =>  s_axi_ctrl_status_arprot,
        s_axi_ctrl_status_arvalid   =>  s_axi_ctrl_status_arvalid,
        s_axi_ctrl_status_arready   =>  s_axi_ctrl_status_arready,
        s_axi_ctrl_status_rdata     =>  s_axi_ctrl_status_rdata,
        s_axi_ctrl_status_rresp     =>  s_axi_ctrl_status_rresp,
        s_axi_ctrl_status_rvalid    =>  s_axi_ctrl_status_rvalid,
        s_axi_ctrl_status_rready    =>  s_axi_ctrl_status_rready
  );

--Dot product module declaration
gen_dp: for i in 0 to 3 generate
dp: entity work.dot_product_module
    generic map(
        RAW_BIT_WIDTH   =>  16,
        G_BIT_WIDTH     =>  G_BIT_WIDTH,
        NUM_B_RAM       =>  NUM_B_RAM,
        P_BIT_WIDTH     =>  P_BIT_WIDTH
    )
    port map(
        clk             =>  clk,
        aresetn         =>  aresetn,
        en              =>  dp_enable(i),
        in_G            =>  dp_data_in((G_BIT_WIDTH*NUM_B_RAM)*i+(G_BIT_WIDTH*NUM_B_RAM)-1 downto (
              G_BIT_WIDTH*NUM_B_RAM)*i),
        in_raw          =>  fifo_register_timing_opt(i*16+15 downto i*16),
        v_len           =>  v_len,
        p_rdy           =>  p_rdy(i),
        p_out           =>  p_out(NUM_B_RAM*P_BIT_WIDTH*i + NUM_B_RAM*P_BIT_WIDTH-1 downto NUM_B_RAM*
              P_BIT_WIDTH*i)
    );
end generate gen_dp;

--Output module/Axi Gearbox declaration
gb: entity work.axi_gearbox
    generic map(
        B_RAM_SIZE  =>  B_RAM_SIZE,
        B_RAM_BIT_WIDTH => G_BIT_WIDTH,
        NUM_B_RAM   =>  NUM_B_RAM,
        RAW_BIT_WIDTH   =>  RAW_BIT_WIDTH,
        G_BIT_WIDTH     =>  G_BIT_WIDTH,
        P_BIT_WIDTH     =>  P_BIT_WIDTH,
        C_S_AXI_DATA_WIDTH  =>  C_S_AXI_DATA_WIDTH,
        C_S_AXI_ADDR_WIDTH  =>  C_S_AXI_ADDR_WIDTH
    )
    port map(
        clk     =>      clk,
        aresetn =>      aresetn,
        p_out   =>      p_out,
        p_rdy   =>      p_rdy,
        p_int   =>      p_irq_w,
        Ref_order => Ref_Order,
        enable => enable,
        last_p  =>      last_p,
        num_pixels  =>   num_pixels,
        m_axis_tdata    =>  m_axis_tdata,
        m_axis_tvalid   =>  m_axis_tvalid,
        m_axis_tready   =>  m_axis_tready,
        m_axis_tlast    =>  m_axis_tlast,
        out_stream_handshake => out_stream_handshake
    );
end Behavioral;
```

G.2.3 Top module testbench

```verilog
`timescale 1ns / 1ps

module top_tb;
   parameter C_S_AXI_DATA_WIDTH = 32;
   parameter C_S_AXI_ADDR_WIDTH = 6;
   parameter B_RAM_SIZE = 400;
   parameter NUM_B_RAM = 16;
   parameter RAW_BIT_WIDTH = 64;
   parameter G_BIT_WIDTH = 32;
   parameter P_BIT_WIDTH = 64;
   parameter PERIOD = 10;

   parameter FIFO_DEPTH = 64;
   parameter FIFO_SIZE = 52;
   parameter WRITE_DATA_WIDTH = 64;
   parameter WR_DATA_COUNT_WIDTH = 7;
   parameter FIFO_MARGIN = 50;
   parameter RD_DATA_COUNT_WIDTH = 7;
   parameter READ_DATA_WIDTH = 16;
   parameter LATENCY_CYCLES = 13;

   reg clk, aresetn;
   reg[63:0] s_axis_tdata;
   reg s_axis_tvalid, s_axis_tlast;
   wire s_axis_tready, p_irq;

   reg[32:0]counter;
   wire[63:0] m_axis_tdata;
   wire m_axis_tvalid, m_axis_tlast;
   reg m_axis_tready;


   reg[5:0]   s_axi_ctrl_status_awaddr;
   reg[2:0]   s_axi_ctrl_status_awprot;
   reg        s_axi_ctrl_status_awvalid;
   reg[31:0]  s_axi_ctrl_status_wdata;
   reg[3:0]   s_axi_ctrl_status_wstrb;
   reg        s_axi_ctrl_status_wvalid;
   reg        s_axi_ctrl_status_bready;
   reg[5:0]   s_axi_ctrl_status_araddr;
   reg[2:0]   s_axi_ctrl_status_arprot;
   reg        s_axi_ctrl_status_arvalid;
   reg        s_axi_ctrl_status_rready;
   wire       s_axi_ctrl_status_awready;
   wire       s_axi_ctrl_status_wready;
   wire[1:0]  s_axi_ctrl_status_bresp;
   wire       s_axi_ctrl_status_bvalid;
   wire       s_axi_ctrl_status_arready;
   wire[31:0] s_axi_ctrl_status_rdata;
   wire[1:0]  s_axi_ctrl_status_rresp;
   wire       s_axi_ctrl_status_rvalid;


   top
      #(.C_S_AXI_DATA_WIDTH(C_S_AXI_DATA_WIDTH),
        .C_S_AXI_ADDR_WIDTH(C_S_AXI_ADDR_WIDTH),
        .B_RAM_SIZE(B_RAM_SIZE),
        .NUM_B_RAM(NUM_B_RAM),
        .RAW_BIT_WIDTH(RAW_BIT_WIDTH),
        .G_BIT_WIDTH(G_BIT_WIDTH),
        .P_BIT_WIDTH(P_BIT_WIDTH),
        .FIFO_DEPTH(FIFO_DEPTH),
        .FIFO_SIZE(FIFO_SIZE),
        .WRITE_DATA_WIDTH(WRITE_DATA_WIDTH),
        .WR_DATA_COUNT_WIDTH(WR_DATA_COUNT_WIDTH),
        .FIFO_MARGIN(FIFO_MARGIN),
        .RD_DATA_COUNT_WIDTH(RD_DATA_COUNT_WIDTH),
        .READ_DATA_WIDTH(READ_DATA_WIDTH),
        .LATENCY_CYCLES(LATENCY_CYCLES)
        )
   DUT
       (.clk(clk),
        .aresetn(aresetn),
        .p_irq(p_irq),
        //IN-STREAM
        .s_axis_tdata(s_axis_tdata),
        .s_axis_tvalid(s_axis_tvalid),
        .s_axis_tready(s_axis_tready),
        .s_axis_tlast(s_axis_tlast),

        //OUT-STREAM
        .m_axis_tdata(m_axis_tdata),
        .m_axis_tvalid(m_axis_tvalid),
        .m_axis_tready(m_axis_tready),
        .m_axis_tlast(m_axis_tlast),

        //REGISTER-INTERFACE
        .s_axi_ctrl_status_awaddr(s_axi_ctrl_status_awaddr),
        .s_axi_ctrl_status_awprot(s_axi_ctrl_status_awprot),
        .s_axi_ctrl_status_awvalid(s_axi_ctrl_status_awvalid),
        .s_axi_ctrl_status_wdata(s_axi_ctrl_status_wdata),
        .s_axi_ctrl_status_wstrb(s_axi_ctrl_status_wstrb),
        .s_axi_ctrl_status_wvalid(s_axi_ctrl_status_wvalid),
        .s_axi_ctrl_status_bready(s_axi_ctrl_status_bready),
        .s_axi_ctrl_status_araddr(s_axi_ctrl_status_araddr),
        .s_axi_ctrl_status_arprot(s_axi_ctrl_status_arprot),
        .s_axi_ctrl_status_arvalid(s_axi_ctrl_status_arvalid),
        .s_axi_ctrl_status_rready(s_axi_ctrl_status_rready),
        .s_axi_ctrl_status_awready(s_axi_ctrl_status_awready),
        .s_axi_ctrl_status_wready(s_axi_ctrl_status_wready),
        .s_axi_ctrl_status_bresp(s_axi_ctrl_status_bresp),
        .s_axi_ctrl_status_bvalid(s_axi_ctrl_status_bvalid),
        .s_axi_ctrl_status_arready(s_axi_ctrl_status_arready),
        .s_axi_ctrl_status_rdata(s_axi_ctrl_status_rdata),
        .s_axi_ctrl_status_rresp(s_axi_ctrl_status_rresp),
        .s_axi_ctrl_status_rvalid(s_axi_ctrl_status_rvalid)
       );

    always #(PERIOD/2) clk = ~clk;

    integer          f_in_G, f_in_raw, f_out_P;
    integer          iter, i;
    reg[31:0] in_G_temp;
    reg[63:0] in_raw_temp;
    reg[31:0] read_holder;
    reg flagg;

    initial begin
        clk = 1'b0;
        aresetn = 1'b0;
        counter = 32'b0;
        s_axis_tlast <= 1'b0;
        s_axi_ctrl_status_awprot = 'b0;
        s_axi_ctrl_status_bready = 1'b0;
        s_axi_ctrl_status_wstrb = 4'hF;
        s_axi_ctrl_status_arprot = 'b0;
        m_axis_tready <= 1'b1;




        f_in_G = $fopen("D:/MasterOppgave/smallsat_prototype/EMSC/in_G.bin", "rb");

        if (f_in_G == 0) begin
            $display("Failed to open input file %s", "D:/MasterOppgave/smallsat_prototype/EMSC/in_G.bin");
            $finish;
        end

        f_in_raw = $fopen("D:/MasterOppgave/smallsat_prototype/EMSC/Test/Hardware/raw_large.bin", "rb");

        if (f_in_raw == 0) begin
            $display("Failed to open input file %s", "D:/MasterOppgave/smallsat_prototype/EMSC/Test/Hardware
                /raw_large.bin");
            $finish;
        end

        f_out_P = $fopen("D:/MasterOppgave/smallsat_prototype/EMSC/P_out_tb.bin", "wb");

        if (f_out_P == 0) begin
            $display("Failed to open input file %s", "D:/MasterOppgave/smallsat_prototype/EMSC/P_out_tb.bin"
                );
            $finish;
        end

        repeat(2) @(posedge clk);
        aresetn = 1'b1;
        write_to_reg(6'h8, 32'h3D090);
        write_to_reg(6'h0, 32'h22034);

        repeat(3) @(posedge clk);

        for (iter = 0; iter < 416; iter = iter + 1) begin
            //|
            for( i = 0; i < 4; i = i + 1) begin
                //|
                in_G_temp[i*8  +: 8] = $fgetc(f_in_G);
                                                                    //|
            end
            write_to_reg(6'h4, in_G_temp);

                //|
            @(posedge clk);

                //|
        end

        write_to_reg(6'h0, 32'h20034);
        repeat(2) @(posedge clk);
```

```

     write_to_reg(6'h0, 32'h21034);
     repeat(10) @(posedge clk);



for (iter = 0; iter < 3250000; iter = iter + 1) begin
     while(s_axis_tready == 1'b0) begin
            @(posedge clk);
     end
                                                        //|
     for( i = 0; i < 8; i = i + 1) begin                                                                     //|
         in_raw_temp[i*8  +: 8] = $fgetc(f_in_raw);
                                                                         //|
     end


     if($urandom % 10 == 0) begin
         s_axis_tvalid = 1'b0;
         repeat(4) @(posedge clk);
     end

     s_axis_tvalid = 1'b1;
     s_axis_tdata = in_raw_temp;
     if(iter == 3249999) begin
         s_axis_tlast <= 1'b1;
     end                                                                                                     //|
     @(posedge clk);
     counter = counter + 1;
                                                                                          //|
  end
 s_axis_tlast <= 1'b0;
 s_axis_tvalid <= 1'b0;




     $fclose(f_in_G);
     $fclose(f_in_raw);
     repeat(100) @(posedge clk);
     $fclose(f_out_P);
 end


 always @(posedge clk) begin
     if($urandom % 15 == 0) begin
         m_axis_tready <= 1'b0;
         repeat(6) @(posedge clk);
     end
     else
         m_axis_tready <= 1'b1;
     end

integer byte_idx, j;




 task write_to_reg;
     input [5:0] address;
     input [31:0] data;
     begin
         @(posedge clk);
         s_axi_ctrl_status_awaddr <= address;
         s_axi_ctrl_status_awvalid <= 1'b1;
         s_axi_ctrl_status_wvalid <= 1'b1;
         s_axi_ctrl_status_wdata <= data;

         while (!(s_axi_ctrl_status_awready == 1'b1 && s_axi_ctrl_status_wready == 1'b1)) begin
             @(posedge clk);
         end

         s_axi_ctrl_status_awvalid <= 1'b0;
         s_axi_ctrl_status_wvalid <= 1'b0;
     end
  endtask



  task read_reg;
          input  [5:0] address;
          output [31:0] data;
          begin
              @(posedge clk);
              s_axi_ctrl_status_araddr = address;
              s_axi_ctrl_status_arvalid = 1'b1;
              s_axi_ctrl_status_rready = 1'b1;
              while (!(s_axi_ctrl_status_rvalid == 1'b1)) begin
                  @(posedge clk);
              end

```

```
                 s_axi_ctrl_status_rready <= 1'b0;
                 s_axi_ctrl_status_arvalid <= 1'b0;
                 data = s_axi_ctrl_status_rdata;


             end
          endtask

endmodule
```

G.2.4 FIFO module design file

```
library IEEE;
Library xpm;
use IEEE.STD_LOGIC_1164.ALL;
use xpm.vcomponents.all;

entity fifo_module is
    Generic(
            FIFO_DEPTH : integer := 16;
            FIFO_SIZE : integer := 52;
            WRITE_DATA_WIDTH : integer := 64;
            WR_DATA_COUNT_WIDTH : integer := 5;
            FIFO_MARGIN : integer := 5;
            RD_DATA_COUNT_WIDTH : integer := 5;
            READ_DATA_WIDTH : integer := 16;
            LATENCY_CYCLES  : integer := 13
    );
    Port (
            clk : in std_logic;
            aresetn : in std_logic;
            enable : in std_logic;
            fifo_read : in std_logic_vector(3 downto 0);
            fifo_in : in std_logic_vector(63 downto 0);
            fifo_out: out std_logic_vector(16*4-1 downto 0);
            fifo_empty : out std_logic_vector(3 downto 0);
            fifo_full : out std_logic_vector(3 downto 0)
    );
end fifo_module;

architecture Behavioral of fifo_module is
    signal reset : std_logic;
    signal fifo_enable : std_logic_vector(3 downto 0);
    signal fifo_sel : integer range 0 to 4;
    signal fifo_in_w : std_logic_vector(64*4-1 downto 0);
    signal rd_flag : std_logic;
    signal fifo_in_delay : std_logic_vector(63 downto 0);

    TYPE State_type IS (IDLE, FIFO1, FIFO2, FIFO3, FIFO4);  -- Define the states
    SIGNAL state : State_Type;    -- Create a signal that uses

begin

reset <= not aresetn;

--State machince to fill fifos
process(clk, aresetn)
    variable counter : integer := 0;
    variable start : std_logic := '0';
begin
    if(aresetn = '0') then
        fifo_sel <= 1;
        state <= IDLE;
        counter := 0;
        fifo_in_delay <= (others => '0');
        start := '0';
    elsif(rising_edge(clk)) then
        case state is
            when IDLE =>
                if(enable = '1') then
                    fifo_sel <= 1;
                    state <= FIFO1;
                end if;
            when FIFO1 =>
                if(enable = '1') then
                    counter := counter + 1;
                    if(start = '0') then
                        if(counter >= LATENCY_CYCLES-1) then
                            state <= FIFO2;
                            fifo_sel <= 2;
                            counter := 0;
                        end if;
                    else
                        if(counter >= LATENCY_CYCLES) then
                            state <= FIFO2;
                            fifo_sel <= 2;
                            counter := 0;
                        end if;
                    end if;
                end if;
            when FIFO2 =>
                if(enable = '1') then
                    counter := counter +1;
                    if(counter >= LATENCY_CYCLES) then
                        state <= FIFO3;
                        fifo_sel <= 3;
                        counter := 0;
                    end if;
                end if;
            when FIFO3 =>
                if(enable = '1') then
                    counter := counter +1;
                    if(counter >= LATENCY_CYCLES) then
                        state <= FIFO4;
                        fifo_sel <= 4;
                        counter := 0;
                    end if;
                end if;
            when FIFO4 =>
                if(enable = '1') then
                    counter := counter +1;
                    if(counter >= LATENCY_CYCLES) then
                        state <= FIFO1;
                        fifo_sel <= 1;
                        counter := 0;
                        start := '1';
                    end if;
                end if;
        end case;
        fifo_in_delay <= fifo_in;
    end if;
end process;

--Control fifo enables for all fifos
fifo_enable(0) <= '1' when ((state=FIFO1 or state=IDLE) and enable = '1') else '0';
fifo_enable(1) <= '1' when (state=FIFO2 and enable = '1') else '0';
fifo_enable(2) <= '1' when (state=FIFO3 and enable = '1') else '0';
fifo_enable(3) <= '1' when (state=FIFO4 and enable = '1') else '0';

--Control mux for input to fifos
process(fifo_sel, fifo_in_delay)
begin
    case fifo_sel is
        when 1 =>
            fifo_in_w(63 downto 0) <= fifo_in_delay;
            fifo_in_w(255 downto 64) <= (others => '0');
        when 2 =>
            fifo_in_w(63 downto 0) <= (others => '0');
            fifo_in_w(127 downto 64) <= fifo_in_delay;
            fifo_in_w(255 downto 128) <= (others => '0');
        when 3 =>
            fifo_in_w(127 downto 0) <= (others => '0');
            fifo_in_w(191 downto 128) <= fifo_in_delay;
            fifo_in_w(255 downto 192) <= (others => '0');
        when 4 =>
            fifo_in_w(192 downto 0) <= (others => '0');
            fifo_in_w(255 downto 192) <= fifo_in_delay;
        when others =>
            fifo_in_w <= (others => '0');
    end case;
end process;

--Declaration of FIFOs
fifo_gen: for i in 0 to 3 generate
    fifo: entity work.fifo
     Generic map(
                FIFO_DEPTH          =>  FIFO_DEPTH,
                FIFO_SIZE           =>  FIFO_SIZE,
                WRITE_DATA_WIDTH    =>  WRITE_DATA_WIDTH,
                WR_DATA_COUNT_WIDTH =>  WR_DATA_COUNT_WIDTH,
                FIFO_MARGIN         =>  FIFO_MARGIN,
                RD_DATA_COUNT_WIDTH =>  RD_DATA_COUNT_WIDTH,
                READ_DATA_WIDTH     =>  READ_DATA_WIDTH,
                LATENCY_CYCLES      =>  LATENCY_CYCLES
        )
        Port  map(
                clk                 =>  clk,
                aresetn             =>  aresetn,
                wr_en               =>  fifo_enable(i),
                rd_en               =>  fifo_read(i),
                fifo_in             =>  fifo_in_w(64*i+63 downto 64*i),
                fifo_out            =>  fifo_out(16*i+15 downto 16*i),
                fifo_empty          =>  fifo_empty(i),
                fifo_full           =>  fifo_full(i)
        );
    end generate fifo_gen;
end Behavioral;
```

G.2.5 FIFO module testbench

```
`timescale 1ns / 1ps

module fifo_module_tb;

parameter FIFO_DEPTH = 16;
parameter FIFO_SIZE = 52;
parameter WRITE_DATA_WIDTH = 64;
parameter WR_DATA_COUNT_WIDTH = 5;
parameter FIFO_MARGIN = 5;
parameter RD_DATA_COUNT_WIDTH = 5;
parameter READ_DATA_WIDTH = 16;
parameter PERIOD = 10;

reg clk, aresetn, enable;
reg[63:0] fifo_in;
wire[16*4-1:0] fifo_out;
wire[3:0] fifo_empty;
wire[3:0] fifo_full;

    fifo_module
        #(.FIFO_DEPTH(FIFO_DEPTH),
          .FIFO_SIZE(FIFO_SIZE),
          .WRITE_DATA_WIDTH(WRITE_DATA_WIDTH),
          .WR_DATA_COUNT_WIDTH(WR_DATA_COUNT_WIDTH),
          .FIFO_MARGIN(FIFO_MARGIN),
          .RD_DATA_COUNT_WIDTH(RD_DATA_COUNT_WIDTH),
          .READ_DATA_WIDTH(READ_DATA_WIDTH))
    DUT
        (.clk(clk),
         .aresetn(aresetn),
         .enable(enable),
         .fifo_in(fifo_in),
         .fifo_out(fifo_out),
         .fifo_empty(fifo_empty),
         .fifo_full(fifo_full));


    always #(PERIOD/2) clk = ~clk;

    integer f_in_raw;
    reg[63:0] in_raw_temp;
    integer iter, i;

    initial begin
        clk <= 1'b0;
        enable <= 1'b0;
        aresetn <= 1'b0;
        fifo_in <= 64'b0;
        repeat(2) @(posedge clk);

        f_in_raw = $fopen("D:/MasterOppgave/smallsat_prototype/EMSC/Test/raw_large.bin", "rb");

        if (f_in_raw == 0) begin
            $display("Failed to open input file %s", "D:/MasterOppgave/smallsat_prototype/EMSC/Test/raw_large.bin");
            $finish;
        end

        repeat(2) @(posedge clk);
            aresetn = 1'b1;

        repeat(2) @(posedge clk);

        //enable <= 1'b1;

        for (iter = 0; iter < 104; iter = iter + 1) begin                                              //|
            for( i = 0; i < 8; i = i + 1) begin                                                        //|
                in_raw_temp[i*8  +: 8] = $fgetc(f_in_raw);                                             //|
            end

            if($urandom % 3 == 0) begin
                enable <= 1'b0;
                repeat(2) @(posedge clk);
            end

            enable <= 1'b1;
            fifo_in <= in_raw_temp;
            repeat(1) @(posedge clk);                                          //|
         end



        $fclose(f_in_raw);
    end

endmodule
```

G.2.6 Block ram bank design file

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


entity b_ram_bank is
  Generic(
          B_RAM_SIZE      : integer := 100;
          B_RAM_BIT_WIDTH : integer := 32;
          NUM_B_RAM       : integer := 8;
          C_S_AXI_DATA_WIDTH : integer := 32;
          C_S_AXI_ADDR_WIDTH : integer := 6
       );
  Port (
        clk             :       in std_logic;
        aresetn         :       in std_logic;
        read_enable     :       in std_logic;
        enable          :       out std_logic;
--      valid_input     :       in std_logic;

        v_len           :       out std_logic_vector(11 downto 0);
        R_order         :       out std_logic_vector(5 downto 0);
        init_flag       :       out std_logic;
        data_out        :       out std_logic_vector(B_RAM_BIT_WIDTH*NUM_B_RAM-1 downto 0);
        num_pixels      :       out std_logic_vector(31 downto 0);


        -- Register interface
        s_axi_ctrl_status_awaddr  : in  std_logic_vector(5 downto 0);
        s_axi_ctrl_status_awprot  : in  std_logic_vector(2 downto 0);
        s_axi_ctrl_status_awvalid : in  std_logic;
        s_axi_ctrl_status_awready : out std_logic;
        s_axi_ctrl_status_wdata   : in  std_logic_vector(31 downto 0);
        s_axi_ctrl_status_wstrb   : in  std_logic_vector(3 downto 0);
        s_axi_ctrl_status_wvalid  : in  std_logic;
        s_axi_ctrl_status_wready  : out std_logic;
        s_axi_ctrl_status_bresp   : out std_logic_vector(1 downto 0);
        s_axi_ctrl_status_bvalid  : out std_logic;
        s_axi_ctrl_status_bready  : in  std_logic;
        s_axi_ctrl_status_araddr  : in  std_logic_vector(5 downto 0);
        s_axi_ctrl_status_arprot  : in  std_logic_vector(2 downto 0);
        s_axi_ctrl_status_arvalid : in  std_logic;
        s_axi_ctrl_status_arready : out std_logic;
        s_axi_ctrl_status_rdata   : out std_logic_vector(31 downto 0);
        s_axi_ctrl_status_rresp   : out std_logic_vector(1 downto 0);
        s_axi_ctrl_status_rvalid  : out std_logic;
        s_axi_ctrl_status_rready  : in  std_logic
  );
end b_ram_bank;

architecture Behavioral of b_ram_bank is

--Control/status registers
signal emsc2cpu_register : std_logic_vector(31 downto 0);
signal cpu2emsc_register : std_logic_vector(31 downto 0);
signal in_G_register     : std_logic_vector(31 downto 0);

--Control signals
signal init, valid_input : std_logic;
signal G_size : std_logic_vector(11 downto 0);
signal Ref_Order : std_logic_vector(5 downto 0);
signal initialized : std_logic;


--Registers
signal data_in_w    : std_logic_vector(B_RAM_BIT_WIDTH-1 downto 0);
signal read_address : integer range 0 to B_RAM_SIZE-1;
signal write_enable : std_logic_vector(NUM_B_RAM-1 downto 0);
signal b_ram_sel    : std_logic_vector(NUM_B_RAM-1 downto 0);
signal data_in      : std_logic_vector(31 downto 0);

TYPE state_type IS (idle, write, read);
SIGNAL state : state_type;

begin
```

```

data_in_w <= data_in(B_RAM_BIT_WIDTH-1 downto 0) when aresetn = '1' else (others => '0');

init_flag <= initialized;
v_len <= G_size;
G_size <= cpu2emsc_register(11 downto 0);
R_order <= Ref_Order;
Ref_Order <= cpu2emsc_register(19 downto 14);
enable <= cpu2emsc_register(12) when initialized = '1' else '0';
init <= cpu2emsc_register(13);
emsc2cpu_register(0) <= initialized;
emsc2cpu_register(31 downto 1) <= (others => '0');



register_interface: entity work.register_interface
    generic map(
        C_S_AXI_DATA_WIDTH => C_S_AXI_DATA_WIDTH,
        C_S_AXI_ADDR_WIDTH => C_S_AXI_ADDR_WIDTH,
        B_RAM_SIZE         => B_RAM_SIZE,
        B_RAM_BIT_WIDTH    => B_RAM_BIT_WIDTH,
        NUM_B_RAM          => NUM_B_RAM
    )
    port map(
         s_axi_aclk    => clk,
         s_axi_aresetn => aresetn,

         s_axi_awaddr  => s_axi_ctrl_status_awaddr,
         s_axi_awprot  => s_axi_ctrl_status_awprot,
         s_axi_awvalid => s_axi_ctrl_status_awvalid,
         s_axi_awready => s_axi_ctrl_status_awready,

         s_axi_wdata  => s_axi_ctrl_status_wdata,
         s_axi_wstrb  => s_axi_ctrl_status_wstrb,
         s_axi_wvalid => s_axi_ctrl_status_wvalid,
         s_axi_wready => s_axi_ctrl_status_wready,

         s_axi_bresp  => s_axi_ctrl_status_bresp,
         s_axi_bvalid => s_axi_ctrl_status_bvalid,
         s_axi_bready => s_axi_ctrl_status_bready,

         s_axi_araddr  => s_axi_ctrl_status_araddr,
         s_axi_arprot  => s_axi_ctrl_status_arprot,
         s_axi_arvalid => s_axi_ctrl_status_arvalid,
         s_axi_arready => s_axi_ctrl_status_arready,

         s_axi_rdata  => s_axi_ctrl_status_rdata,
         s_axi_rresp  => s_axi_ctrl_status_rresp,
         s_axi_rvalid => s_axi_ctrl_status_rvalid,
         s_axi_rready => s_axi_ctrl_status_rready,

         --Register Outputs
         emsc2cpu_register  => emsc2cpu_register,

         --Register Inputs
         cpu2emsc_register => cpu2emsc_register,
         in_G_register     => data_in,
         num_pixels        => num_pixels,
         valid_input       => valid_input
         --read_enable       => read_enable_w
    );


b_ram: for i in 0 to NUM_B_RAM-1 generate
    DUT : 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         =>          aresetn,
        data_in         =>          data_in_w,
        write_enable    =>          write_enable(i),
        read_enable     =>          read_enable,
        read_address    =>          read_address,
        data_out        =>          data_out(B_RAM_BIT_WIDTH*i + B_RAM_BIT_WIDTH-1 downto B_RAM_BIT_WIDTH*i)
    );
end generate b_ram;


process(clk, aresetn)
    variable counter : integer range 0 to B_RAM_SIZE-1 := 0;
    variable b_ram_written : integer range 0 to 32 := 0;
    variable prev_b_ram_addr : std_logic_vector(NUM_B_RAM-1 downto 0);
    variable valid_prev : std_logic;
begin
    if(aresetn = '0') then
        initialized <= '0';
        b_ram_sel <= (others => '0');
        b_ram_written := 0;
        state <= idle;
    elsif(rising_edge(clk)) then
            case state is
                --Stays in idle until either a init or read should be
                --performed
                when idle =>
                    counter := 0;
                    if(init = '1' and valid_input = '1') then
                        state <= write;
                        b_ram_sel <= (0 => '1', others => '0');
                        counter := counter + 1;
                    elsif(read_enable = '1' and initialized = '1') then
                        read_address <= read_address + 1;
                        state <= read;
                    end if;

                --Stays in write until initialization is completed.
                --Has to take care of bubbles in input data
                when write =>
                    if(valid_input = '1') then
                        counter := counter + 1;
                        if(counter >= to_integer(unsigned(G_size))) then
                            if(write_enable(to_integer(unsigned(Ref_Order))-1) = '1' or b_ram_written >=
                                to_integer(unsigned(Ref_Order))-1) then
                                state <= idle;
                                initialized <= '1';
                                --write_enable <= (others => '0');
                            else
                                --write_enable <= write_enable(NUM_B_RAM-2 downto 0) & '0';
                                b_ram_sel <= b_ram_sel(NUM_B_RAM-2 downto 0) & '0';
                                b_ram_written := b_ram_written + 1;
                                counter := 0;
                            end if;
                        end if;
                    end if;

                --The read state should simply read out 1 value from each B_ram
                --each cycle.
                when read =>
                    if(read_enable = '1') then
                        read_address <= read_address + 1;
                        if(read_address >= to_integer(unsigned(G_size))-1) then
                            state <= idle;
                            read_address <= 0;
                        end if;
                    end if;
            end case;
    end if;
end process;


process(b_ram_sel, state, init, valid_input)
begin
    if(state = write and valid_input = '1') then
        write_enable <= b_ram_sel;
    elsif(state = idle and init = '1' and valid_input = '1') then
        write_enable <= (0 => '1', others =>'0');
    else
        write_enable <= (others => '0');
    end if;
end process;

end Behavioral;
```

G.2.7 Block ram design file

Same as sequential desing

G.2.8 Dot product module design file

Same as sequential design.

G.2.9 Dot product core design file

Same as sequential design.

G.2.10 Output Module design file

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
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;

entity axi_gearbox is
      Generic(
        B_RAM_SIZE      : integer := 100;
        B_RAM_BIT_WIDTH : integer := 32;
        NUM_B_RAM       : integer := 8;
        RAW_BIT_WIDTH : positive := 16;
        G_BIT_WIDTH   : positive := 32;
        P_BIT_WIDTH   : positive := 48;
        C_S_AXI_DATA_WIDTH : integer := 32;
        C_S_AXI_ADDR_WIDTH : integer := 6
     );
    Port (
        clk     :   in  std_logic;
        aresetn :   in  std_logic;
        p_out   :   in  std_logic_vector(4*P_BIT_WIDTH*NUM_B_RAM-1 downto 0);
        p_rdy   :   in  std_logic_vector(3 downto 0);
        enable  :   in  std_logic;
        p_int   :   out std_logic;
        last_p  :   in  std_logic;
        num_pixels : in std_logic_vector(31 downto 0);
        Ref_order : in  std_logic_vector(5 downto 0);

        m_axis_tdata  : out std_logic_vector(63 downto 0);
        --EMSC is ready to send to DMA.
        m_axis_tvalid : out std_logic;
        --DMA is ready to receive data
        m_axis_tready : in  std_logic;
        --Tell DMA this is last data
        m_axis_tlast  : out std_logic;

        out_stream_handshake : out std_logic
    );
end axi_gearbox;

architecture Behavioral of axi_gearbox is
signal res_mem : std_logic_vector(P_BIT_WIDTH*NUM_B_RAM-1 downto 0);
signal start : std_logic;
signal t_valid_flag : std_logic;
signal out_handshake : std_logic;
signal last_t_valid : std_logic;
begin

out_stream_handshake <= out_handshake;

--Process to output data on
--AXI-stream interface
process(clk,aresetn)
    variable counter : integer range 0 to 50 := 0;
begin
    if(aresetn = '0') then
        m_axis_tdata  <= (others => '0');
        m_axis_tvalid <= '0';
        t_valid_flag <= '0';
        counter := 0;
    elsif(rising_edge(clk)) then
        if(p_rdy(0) = '1') then
            res_mem <= p_out(P_BIT_WIDTH*NUM_B_RAM-1 downto 0);
            start <= '1';
        elsif(p_rdy(1) = '1') then
            res_mem <= p_out(2*P_BIT_WIDTH*NUM_B_RAM-1 downto P_BIT_WIDTH*NUM_B_RAM);
            start <= '1';
        elsif(p_rdy(2) = '1') then
            res_mem <= p_out(3*P_BIT_WIDTH*NUM_B_RAM-1 downto 2*P_BIT_WIDTH*NUM_B_RAM);
            start <= '1';
        elsif(p_rdy(3) = '1') then
            res_mem <= p_out(4*P_BIT_WIDTH*NUM_B_RAM-1 downto 3*P_BIT_WIDTH*NUM_B_RAM);
            start <= '1';
        elsif(start = '1') then
            if(m_axis_tready = '1' and t_valid_flag = '1') then
                counter := counter + 1;
            end if;
            if(counter >= to_integer(unsigned(Ref_order))) then
                m_axis_tvalid <= '0';
                t_valid_flag <= '0';
                counter := 0;
                start <= '0';
                m_axis_tdata <= (others => '0');
            else
```

```
                m_axis_tdata <= std_logic_vector(resize(signed(res_mem((P_BIT_WIDTH*counter + P_BIT_WIDTH-1)
                        downto (P_BIT_WIDTH*counter))),m_axis_tdata'length));
                m_axis_tvalid <= '1';
                t_valid_flag <= '1';
            end if;
        else
            counter := 0;
        end if;

    end if;
end process;

--Process to assign m_axis_tlast
process(clk, aresetn)
    variable counter : integer;
    variable p_last_flag : std_logic;
begin
    if(aresetn = '0') then
        counter := 0;
        m_axis_tlast <= '0';
        p_last_flag := '0';
        last_t_valid <= '0';
    elsif(rising_edge(clk)) then
        last_t_valid <= t_valid_flag;
        if(p_last_flag = '0' and enable = '1') then
            if(t_valid_flag = '1' and m_axis_tready = '1') then
                m_axis_tlast <= '0';
            end if;
            if(counter >= to_integer(unsigned(num_pixels))) then
                p_last_flag := '1';
                counter := 0;
            elsif(p_rdy(0) = '1' or p_rdy(1) = '1' or p_rdy(2) = '1' or p_rdy(3) = '1') then
                counter := counter + 1;
            end if;
        elsif(p_last_flag = '1' and enable = '1') then
            counter := counter + 1;
            if(counter = to_integer(unsigned(Ref_order))-1) then
                m_axis_tlast <= '1';
                counter := 0;
                P_last_flag := '0';
            end if;
        end if;
    end if;
end process;

p_int <= '1' when (last_t_valid = '1' and t_valid_flag = '0') else '0';
out_handshake <= '1' when (m_axis_tready = '1' and t_valid_flag = '1') else '0';

end Behavioral;
```

G.2.11 AXI-register interface design file

This is a module designed by Xilinx and found in *LogiCORE IP AXI4-Lite IPIF v2.0* [18]. Some changes was made to make it work with the EMSC parallel application.

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

package B_RAM_BANK_pkg is
    type bus_array is array(natural range <>) of std_logic_vector(31 downto 0);
end package B_RAM_BANK_pkg;

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

entity register_interface is
  generic (
    -- Users to add parameters here
    B_RAM_SIZE : integer := 100;
    B_RAM_BIT_WIDTH : integer := 32;
    NUM_B_RAM       : integer := 5;
    -- User parameters ends
    -- Do not modify the parameters beyond this line

    -- Width of S_AXI data bus
    C_S_AXI_DATA_WIDTH : integer := 32;
    -- Width of S_AXI address bus
    C_S_AXI_ADDR_WIDTH : integer := 6
    );
  port (
    -- Users to add ports here
    emsc2cpu_register : in std_logic_vector(31 downto 0);
    cpu2emsc_register : out std_logic_vector(31 downto 0);
    num_pixels        : out std_logic_vector(31 downto 0);
    in_G_register     : out std_logic_vector(31 downto 0);
    valid_input       : out std_logic;
--    read_enable       : out std_logic;

    -- User ports ends
    -- Do not modify the ports beyond this line

    -- Global Clock Signal
    S_AXI_ACLK    : in  std_logic;
    -- Global Reset Signal. This Signal is Active LOW
    S_AXI_ARESETN : in  std_logic;
    -- Write address (issued by master, acceped by Slave)
    S_AXI_AWADDR  : in  std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
    -- Write channel Protection type. This signal indicates the
    -- privilege and security level of the transaction, and whether
    -- the transaction is a data access or an instruction access.
    S_AXI_AWPROT  : in  std_logic_vector(2 downto 0);
    -- Write address valid. This signal indicates that the master signaling
    -- valid write address and control information.
    S_AXI_AWVALID : in  std_logic;
    -- Write address ready. This signal indicates that the slave is ready
    -- to accept an address and associated control signals.
    S_AXI_AWREADY : out std_logic;
    -- Write data (issued by master, acceped by Slave)
    S_AXI_WDATA   : in  std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    -- Write strobes. This signal indicates which byte lanes hold
    -- valid data. There is one write strobe bit for each eight
    -- bits of the write data bus.
    S_AXI_WSTRB   : in  std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1 downto 0);
    -- Write valid. This signal indicates that valid write
    -- data and strobes are available.
    S_AXI_WVALID  : in  std_logic;
    -- Write ready. This signal indicates that the slave
    -- can accept the write data.
    S_AXI_WREADY  : out std_logic;
    -- Write response. This signal indicates the status
    -- of the write transaction.
    S_AXI_BRESP   : out std_logic_vector(1 downto 0);
    -- Write response valid. This signal indicates that the channel
    -- is signaling a valid write response.
    S_AXI_BVALID  : out std_logic;
    -- Response ready. This signal indicates that the master
    -- can accept a write response.
    S_AXI_BREADY  : in  std_logic;
    -- Read address (issued by master, acceped by Slave)
    S_AXI_ARADDR  : in  std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
    -- Protection type. This signal indicates the privilege
    -- and security level of the transaction, and whether the
    -- transaction is a data access or an instruction access.
    S_AXI_ARPROT  : in  std_logic_vector(2 downto 0);
    -- Read address valid. This signal indicates that the channel
    -- is signaling valid read address and control information.
    S_AXI_ARVALID : in  std_logic;
    -- Read address ready. This signal indicates that the slave is
    -- ready to accept an address and associated control signals.
    S_AXI_ARREADY : out std_logic;
    -- Read data (issued by slave)
    S_AXI_RDATA   : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    -- Read response. This signal indicates the status of the
    -- read transfer.
    S_AXI_RRESP   : out std_logic_vector(1 downto 0);
    -- Read valid. This signal indicates that the channel is
    -- signaling the required read data.
    S_AXI_RVALID  : out std_logic;
    -- Read ready. This signal indicates that the master can
    -- accept the read data and response information.
    S_AXI_RREADY  : in  std_logic
    );
end register_interface;

architecture arch_imp of register_interface is

  -- AXI4LITE signals
  signal axi_awaddr  : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
  signal axi_awready : std_logic;
  signal axi_wready  : std_logic;
  signal axi_bresp   : std_logic_vector(1 downto 0);
  signal axi_bvalid  : std_logic;
  signal axi_araddr  : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
  signal axi_arready : std_logic;
  signal axi_rdata   : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
  signal axi_rresp   : std_logic_vector(1 downto 0);
  signal axi_rvalid  : std_logic;

  -- Example-specific design signals
  -- local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
  -- ADDR_LSB is used for addressing 32/64 bit registers/memories
```

```vhdl
  -- ADDR_LSB = 2 for 32 bits (n downto 2)
  -- ADDR_LSB = 3 for 64 bits (n downto 3)
  constant ADDR_LSB          : integer := (C_S_AXI_DATA_WIDTH/32)+ 1;
  constant OPT_MEM_ADDR_BITS : integer := 3;
  constant C_NUM_REGS        : integer := 16;
  ------------------------------------------------
  ---- Signals for user logic register space example
  --------------------------------------------------
  ---- Number of Slave Registers 16
  type reg_arr_t is array (0 to C_NUM_REGS) of std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
  signal slv_regs  : reg_arr_t;
  signal read_data : reg_arr_t;

  signal slv_reg_rden : std_logic;
  signal slv_reg_wren : std_logic;
  signal reg_data_out : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
  signal byte_index   : integer;
  signal aw_en        : std_logic;

begin
  -- I/O Connections assignments

  S_AXI_AWREADY <= axi_awready;
  S_AXI_WREADY  <= axi_wready;
  S_AXI_BRESP   <= axi_bresp;
  S_AXI_BVALID  <= axi_bvalid;
  S_AXI_ARREADY <= axi_arready;
  S_AXI_RDATA   <= axi_rdata;
  S_AXI_RRESP   <= axi_rresp;
  S_AXI_RVALID  <= axi_rvalid;
  -- Implement axi_awready generation
  -- axi_awready is asserted for one S_AXI_ACLK clock cycle when both
  -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
  -- de-asserted when reset is low.

  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_awready <= '0';
        aw_en       <= '1';
      else
        if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '1' and aw_en = '1') then
          -- slave is ready to accept write address when
          -- there is a valid write address and write data
          -- on the write address and data bus. This design
          -- expects no outstanding transactions.
          axi_awready <= '1';
        elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then
          aw_en       <= '1';
          axi_awready <= '0';
        else
          axi_awready <= '0';
        end if;
      end if;
    end if;
  end process;

  -- Implement axi_awaddr latching
  -- This process is used to latch the address when both
  -- S_AXI_AWVALID and S_AXI_WVALID are valid.

  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_awaddr <= (others => '0');
      else
        if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '1' and aw_en = '1') then
          -- Write Address latching
          axi_awaddr <= S_AXI_AWADDR;
        end if;
      end if;
    end if;
  end process;

  -- Implement axi_wready generation
  -- axi_wready is asserted for one S_AXI_ACLK clock cycle when both
  -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
  -- de-asserted when reset is low.

  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_wready <= '0';
      else
        if (axi_wready = '0' and S_AXI_WVALID = '1' and S_AXI_AWVALID = '1' and aw_en = '1') then
          -- slave is ready to accept write data when
          -- there is a valid write address and write data
          -- on the write address and data bus. This design
          -- expects no outstanding transactions.
          axi_wready <= '1';
        else
          axi_wready <= '0';
        end if;
      end if;
    end if;
  end process;

  -- Implement memory mapped register select and write logic generation
  -- The write data is accepted and written to memory mapped registers when
  -- axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write strobes are used to
  -- select byte enables of slave registers while writing.
  -- These registers are cleared when reset (active low) is applied.
  -- Slave register write enable is asserted when valid address and data are available
  -- and the slave is ready to accept the write address and write data.
  slv_reg_wren <= axi_wready and S_AXI_WVALID and axi_awready and S_AXI_AWVALID;

  process (S_AXI_ACLK)
    variable loc_addr     : integer range 0 to 2**(OPT_MEM_ADDR_BITS+1)-1;
    variable slv_regs_nxt : reg_arr_t;
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        slv_regs <= (others => (others => '0'));
        valid_input <= '0';
      else
        loc_addr := to_integer(unsigned(axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB)));
        valid_input <= '0';
        if (slv_reg_wren = '1') then
            if(loc_addr = 1) then
                valid_input <= '1';
            end if;
          if (loc_addr < C_NUM_REGS) then
            for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
              if (S_AXI_WSTRB(byte_index) = '1') then
                -- Respective byte enables are asserted as per write strobes
                -- slave registor 0
                slv_regs(loc_addr)(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7 downto
                      byte_index*8);
              end if;
            end loop;
          end if;
        end if;
      end if;
    end if;
  end process;

  -- Implement write response logic generation
  -- The write response and response valid signals are asserted by the slave
  -- when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
  -- This marks the acceptance of address and indicates the status of
  -- write transaction.

  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_bvalid <= '0';
        axi_bresp  <= "00";              --need to work more on the responses
      else
        if (axi_awready = '1' and S_AXI_AWVALID = '1' and axi_wready = '1' and S_AXI_WVALID = '1' and
              axi_bvalid = '0') then
          axi_bvalid <= '1';
          axi_bresp  <= "00";
        elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then  --check if bready is asserted while bvalid is
              high)
          axi_bvalid <= '0';  -- (there is a possibility that bready is always asserted high)
        end if;
      end if;
    end if;
  end process;

  -- Implement axi_arready generation
  -- axi_arready is asserted for one S_AXI_ACLK clock cycle when
  -- S_AXI_ARVALID is asserted. axi_awready is
  -- de-asserted when reset (active low) is asserted.
  -- The read address is also latched when S_AXI_ARVALID is
  -- asserted. axi_araddr is reset to zero on reset assertion.

  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_arready <= '0';
        axi_araddr  <= (others => '1');
      else
        if (axi_arready = '0' and S_AXI_ARVALID = '1') then
          -- indicates that the slave has acceped the valid read address
          axi_arready <= '1';
          -- Read Address latching
```

```vhdl
          axi_araddr  <= S_AXI_ARADDR;
        else
          axi_arready <= '0';
        end if;
      end if;
    end if;
  end process;

  -- Implement axi_arvalid generation
  -- axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
  -- S_AXI_ARVALID and axi_arready are asserted. The slave registers
  -- data are available on the axi_rdata bus at this instance. The
  -- assertion of axi_rvalid marks the validity of read data on the
  -- bus and axi_rresp indicates the status of read transaction.axi_rvalid
  -- is deasserted on reset (active low). axi_rresp and axi_rdata are
  -- cleared to zero on reset (active low).
  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_rvalid <= '0';
        axi_rresp  <= "00";
--        read_enable <= '0';
      else
        if (axi_arready = '1' and S_AXI_ARVALID = '1' and axi_rvalid = '0') then
          -- Valid read data is available at the read data bus
          axi_rvalid <= '1';
          axi_rresp  <= "00";          -- 'OKAY' response
        elsif (axi_rvalid = '1' and S_AXI_RREADY = '1') then
          -- Read data is accepted by the master
--          read_enable <= '1';
          axi_rvalid <= '0';
--        else
--            read_enable <= '0';
        end if;
      end if;
    end if;
  end process;

  -- Implement memory mapped register select and read logic generation
  -- Slave register read enable is asserted when valid address is available
  -- and the slave is ready to accept the read address.
  slv_reg_rden <= axi_arready and S_AXI_ARVALID and (not axi_rvalid);

  process (axi_araddr, read_data)
    variable loc_addr : integer range 0 to 2**(OPT_MEM_ADDR_BITS+1)-1;
  begin
    -- Address decoding for reading registers
    loc_addr := to_integer(unsigned(axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB)));
--        read_enable <= '0';
--    if(loc_addr = 0) then
--        read_enable <= '1';
--    end if;
    if (loc_addr < C_NUM_REGS) then
      reg_data_out <= read_data(loc_addr);
    else
      reg_data_out <= (others => '0');
    end if;
  end process;

  -- Output register or memory read data
  process(S_AXI_ACLK) is
  begin
    if (rising_edge (S_AXI_ACLK)) then
      if (S_AXI_ARESETN = '0') then
        axi_rdata <= (others => '0');
      else
        if (slv_reg_rden = '1') then
          -- When there is a valid read address (S_AXI_ARVALID) with
          -- acceptance of read address by the slave (axi_arready),
          -- output the read dada
          -- Read address mux
          axi_rdata <= reg_data_out;    -- register read data
        end if;
      end if;
    end if;
  end process;

  -- Add user logic here
  cpu2emsc_register <= slv_regs(0);
  in_G_register <= slv_regs(1);
  num_pixels <= slv_regs(2);
  -- Data returned when reading is the register values -- except for the cases
  -- where we want reads to behave differently
  process (slv_regs, emsc2cpu_register)
  begin
    for i in 0 to C_NUM_REGS-1 loop
      read_data(i) <= slv_regs(i);
    end loop;
    read_data(1) <= emsc2cpu_register;
  end process;
  -- User logic ends

end arch_imp;
```