FPGA Based Active Camera Stabilization for a Small Satellite

Designing a Digital Control

Benjamin Bissendorf Jan Bredereke Janek Brumund Joscha Knobloch
Simon Pfennig Niklas Seeliger

Hochschule Bremen
Flughafenallee 10, 28199 Bremen

March 4, 2024

Abstract

written by Jan Bredereke

We investigate how an active control can stabilize a satellite's on-board camera in an inexpensive way, disturbed by micro-vibrations coming from the reaction wheels of the Attitude Determination and Control System (ADCS), and maybe also disturbed by its orbital movement. In particular, in this project we investigate how to design the digital part of the control. We use an active control running in the FPGA of a PYNQ-Z2 board. The control is a PID controller, adjusted experimentally. It turns out that obviously it will be feasible to implement the digital part in this way. We did not have the time and human resources for a complete implementation of the digital part. But all digital components have either been implemented or at least shown to be implementable. The mechanical part of a test bed was provided to us by the VIBES project; we didn't intend to work on it. Nevertheless, we conceived several improvements, and we implemented a few of them. Finally, we identify in detail the work that needs to be done, both on the digital part and on the mechanical part, such that we can conduct an experimental proof that the control indeed reduces micro-vibrations as intended.

Acronyms

AXI Advanced eXtensible Interface

DMA Direct Memory Access

FIR Finite Impulse Response

FPGA Field Programmable Gate Array

GPIO General Purpose Input / Output

IP Intellectual Property

I²S Inter Integrated Chip Sound

LSB Least Significant Bit

PCB Printed Circuit Board

PL Programmable Logic

PS Processing System

SPI Serial Peripheral Interface

Contents

Chapter 1

Introduction

written by Jan Bredereke

1.1 Context of the Project

The project "FPGA Based Active Camera Stabilization for a Small Satellite" is part of the elective module "Embedded Systems" of Jan Bredereke. The project is conducted in the context of the research project VIBES of the City University of Applied Sciences Bremen. VIBES investigates micro-vibrations in satellites. Such micro-vibrations degrade the quality of photos of an on-board camera used for scientific Earth observation. The micro-vibrations come from the reaction wheels that are part of the Attitude Determination and Control System (ADCS), in particular.

The project VIBES is headed by Antonio García. VIBES plans to investigate the micro-vibrations using CubeSat small satellites. It also aims to try out solutions practically in this way. VIBES will launch an entire series of CubeSats. The first CubeSat is scheduled to launch in the end of the year 2024.

1.2 Research Question of the Project

The development of the CubeSats encompasses a wide palette of aspects and tasks. The project of the elective module "Embedded Systems" of Jan Bredereke concentrates on one of these aspects, which is the research question:

> How can an active control stabilize a satellite's on-board camera in an inexpensive way,
> disturbed by micro-vibrations coming from the reaction wheels of the ADCS,
> and maybe also disturbed by its orbital movement?

1.3 Related Work

Little detailed information is publicly available on micro-vibrations. Companies building satellites probably have no interest in sharing their results with their competition. A notable exception are handbooks by the European space agency ESA ([ECSS13; ESSB11]) and at least one conference paper ([SVP19]).

An internship report ([Ger23]), written in the context of VIBES, selects an optical system which can be used to visualize the degrading effect of micro-vibration on images. It analyzes the effects of micro-vibrations and of orbital motion. It does not investigate counter-measures.

Chapter 2

Our Approach to Reducing Disturbances

written by Jan Bredereke

2.1 Reducing Micro-Vibrations by the Attitude Determination and Control System

Several ways of reducing disturbances by micro-vibrations are conceivable:

Passive isolation is probably not sufficient.

Active control with an actor must be fast. It can be done either with a **microcontroller** or with an **FPGA**.

Active compensation using a counter-vibration, with a (slowly) controlled amplitude and phase, is less efficient, rather more difficult to realize, and therefore not necessary.

All electronics needs to withstand space radiation, requiring large structural widths on the chips, and consequently are slow.

The following arguments influence our choice between a microcontroller and an FPGA solution:

A microcontroller requires a real-time operating system, because of the high control frequency. Other on-board software needs a conventional operating system. This might necessitate a dedicated microcontroller. Using a microcontroller for micro-vibration control thus is feasible, but needs valuable space, weight and electricity.

The software defined radio of the CubeSat already uses a PYNQ-Z2 board comprising a microcontroller and an FPGA. This microcontroller runs a conventional operating system. Only a small part of the FPGA in the PYNQ-Z2 is used, up to now.

Therefore we decided for an active control running in the FPGA of the PYNQ-Z2.

Two approaches are available for designing the control:

An analytical modelling of the controlled system requires a lot of control theory. And it requires a lot of details on the experimental setup.

A PID controller, adjusted experimentally, needs less knowledge of control theory. It is independent of the experimental setup.

Therefore we chose a PID controller, adjusted experimentally.

2.2 Mitigating Orbital Movement

The disturbances due to the orbital movement can be mitigated by making the camera follow its field of view on Earth. This can be done either by swivelling the entire satellite, or by moving the camera only. The former puts demands on the Attitude Determination and Control System. The latter might be realized by an extension to the micro-vibration control already discussed.

We do not investigate further this aspect for the time being. But we keep in mind that it may prove advantageous to have actors for micro-vibration control which also allow for movements suitable for orbital movement mitigation.

Chapter 3

Mechanical Concept of the First Version of The Test Bed

written by Jan Bredereke

The project VIBES provided a first version of a test bed to us.

3.1 Design Rationale

The approach of the project VIBES is to keep the test bed as simple as possible; the test bed thus assumes that vibrations are strictly translational and not rotational. This simplification comes from previous experiments. These experiments proved that it is possible to measure micro-vibrations at all. Since translational movements are simpler than rotational ones, the simplification was kept for the first version of the test bed here.

3.2 Geometry of the Test Bed

The first version of the test bed consists of two heavy and rigid metal plates, connected by four supporting stands (compare Fig. 3.1 and Fig. 3.2 on the facing page). The lower plate carries a motor capable to run at up to 6000 rpm. It carries an acceleration sensor close to the motor, too. The upper plate carries two audio speakers. The movable parts of these speakers are connected by a thin and light, L-shaped, 3D-printed carrier plate. The carrier plate in turn carries a camera and another acceleration sensor. Both acceleration sensors each are part of a small printed circuit board which allows to solder wires to it. A USB cable from the camera and another cable from the acceleration sensor near the camera lead away from the camera plate. Rubber feet support the entire setup. The off-the-shelf rubber feet are rather stiff.

3.3 Evaluation of the Geometry of the Test Bed

The first version of the test bed would not allow us to design a micro-vibration control system, neither for translational vibrations nor for rotational vibrations. This version is designed for the simplification to translational vibrations. But, inevitably, it will have rotational vibrations, too, interacting with the translational sensors.

Furthermore, the cabling needs improvement. The cables from the camera platform are thick and stiff. They are bound to introduce non-linearities, which additionally will change after every touching of the setup. (We keep the choice of using USB, though. Its four wires are way better than what most of the affordable RasPi cameras offer: a 15-wire flat ribbon cable.)

Consequently, we improved the concept of the test bed.

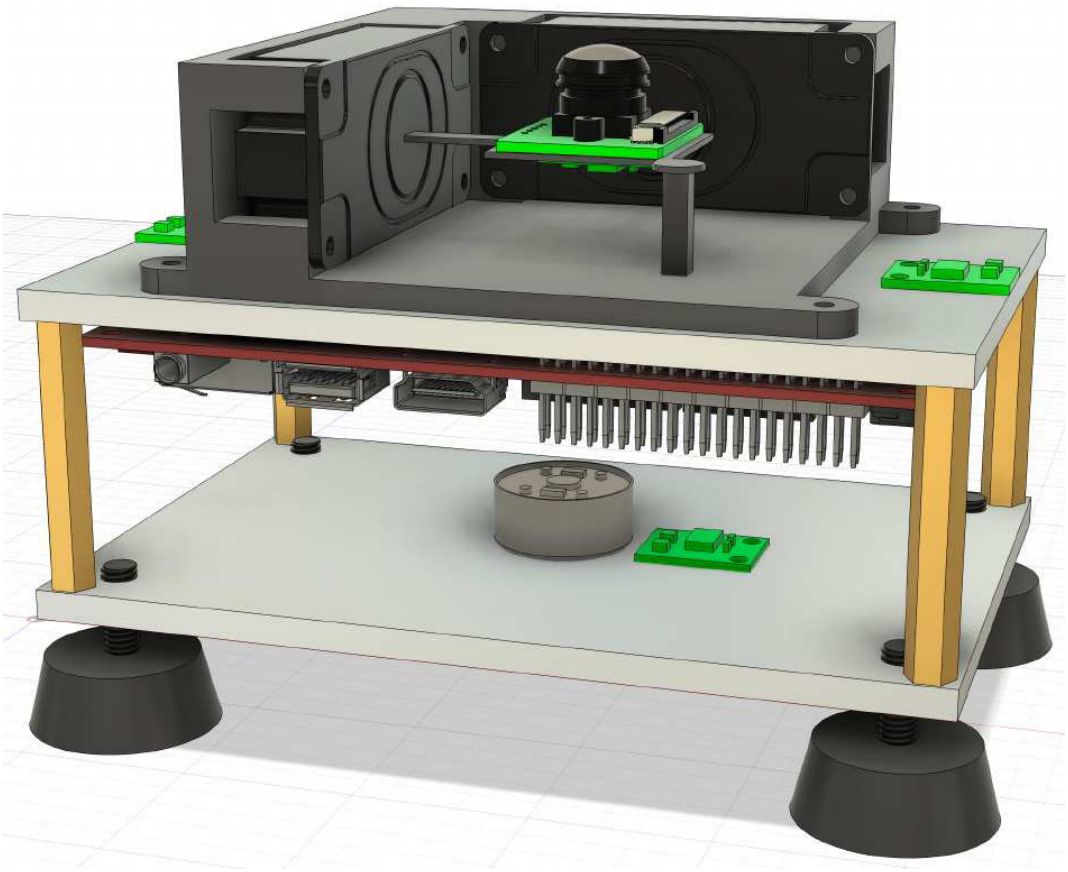

Figure 3.1: Construction drawing of the first version of the test bed. (source: [Wit23])

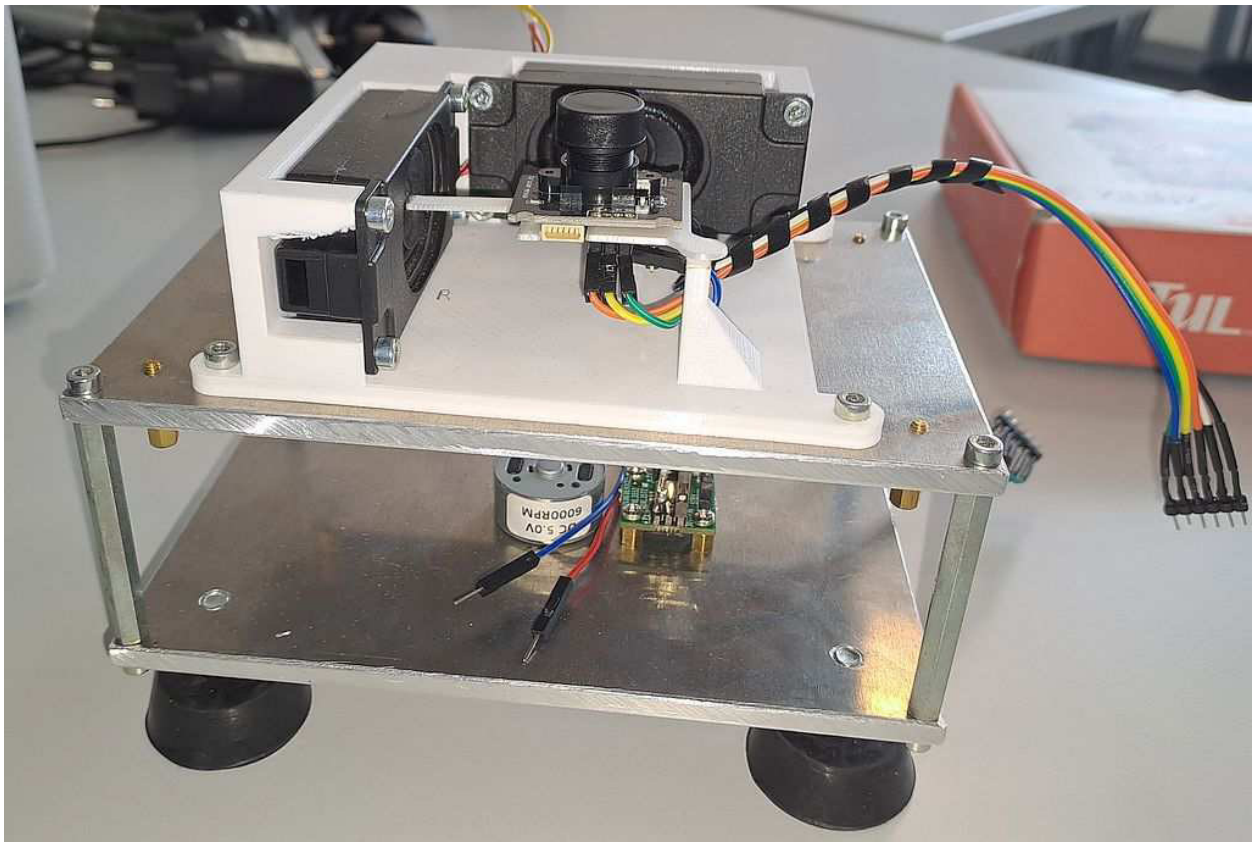

Figure 3.2: Photo of the first version of the test bed. (photo: Winfried Sembritzki)

Chapter 4

Mechanical Concept of Our Second and Third Versions of The Test Bed

written by Jan Bredereke

This section describes our changes to the test bed and the reduced research question for this iteration. Furthermore, we explain that an additional, interesting measurement becomes possible.

Due to strict time constraints, we could not implement all of our ideas from the start of the term. Therefore we describe the second version of the test bed, which we actually realized, and we describe a third version which we propose to make the test bed fit for use in testing.

4.1 Design Rationale

Our second and third version of the test bed discard the problematic simplification to translational vibrations. We now aim for reducing the rotational vibrations using our control system. This has the advantage of coming closer to the real task; only rotational vibrations are relevant in orbit.

En-route we also improve the mechanics of the cabling.

We do not expect the third version of the test bed to be the final, definitive one. Quite likely, there will be some resonances and other mechanical deficiencies.

Our goal is to demonstrate the feasibility of designing an electronic control system able to reduce micro-vibrations at frequencies of up to 1 kHz. The demonstration shall be done at at least one frequency. Our emphasis is on the electronic part. Improving the mechanical part is left to future iterations of the setup.

Similarly, we will not evaluate the images of the camera for blurring. Consequently, the second, implemented version does not comprise a camera. And the envisioned third version could use a dummy camera of similar weight and size, unless there is sufficient time for evaluating the images of a camera, too. Such an evaluation should use a camera with a narrow angle of view, requiring the cost of purchase.

Our second and third versions require three acceleration sensors. In order to reduce the cost for them, we are prepared to increase the amplitude of the micro-vibrations, if necessary. The electronics of our vibration control system is independent of the actual amplitude of the vibrations, except for sensor noise. Sensors with less noise are more expensive. Therefore, we rather increase the amplitude of the vibrations than invest in sensors of a higher quality. The project VIBES already did prove that micro-vibrations can be measured with affordable sensors. We do not need to repeat this work here. Of course, a final version of the test bed must have both the electronics system and sensors of sufficient sensitivity.

4.2 Improved Geometry of the Test Bed

The third version of the test bed we envision keeps the main body with the motor for generating vibrations and with the PYNQ-Z2 board. We replace the upper part with the camera platform and with the actors only.

The camera is mounted on a movable plate held by a flexible rod. It thus can rotate around two axes, when the rod is bent. (In principle, it can also twist around the third axis, the line-of-sight of the camera. We will come back to this in Sect. 4.6 on page 12.) The line-of-sight of the camera is still going straight upward from the plate.

The plate now isn't pushed from two sides anymore. Instead, it is pushed from below by two actors. These actors are placed 90° apart when viewed along the axis of the rod. This allows the two actors to rotate the plate around two axes independently.

The actors are small speakers. They can handle the maximum frequency necessary easily. They are cheap. And they can be purchased off-the-shelf. Figure 4.1 shows the 3D-printed mount for the speakers. (Unfortunately, the speakers purchased do not fit this mount. Their dimensions advertised differ from their actual dimensions.)

Figure 4.1: 3D-printed mount for the speakers. (photo: Winfried Sembritzki)

The size of the plate is wide enough to reach over the speakers.[1] When they push upward, they push the middle of an edge of the plate. Having small speakers, and placing them close to each other, yields the advantage that the plate can be made smaller and thus lighter.

The speakers must be able to bear 3 W of power from the existing audio amplifier. This prevents inadvertent destruction by experimentation errors.

Beyond our third version, we envision improved designs of the actors. A custom actor can be made of a magnet and a coil. This would be much smaller and more robust. Figure 4.2 on the next page and Fig. 4.3 on page 9 show drawings of a camera mount with such actors. The coils are drawn in violet color. However, we did not have the time to iterate through several versions of such a custom actor, which are likely to be necessary. A piezo actor might be possible, too, depending on the maximum amplitude necessary. Since a single piezo actor typically achives a few micrometers only, we did not risk to follow this approach for now. If possible, a piezo actor should be preferable, due to its even more increased

[1] Actually, the speakers purchased are too large for the plate used. But since they do not fit their mount anyway (see above), a mechanical redesign is necessary here in any case.

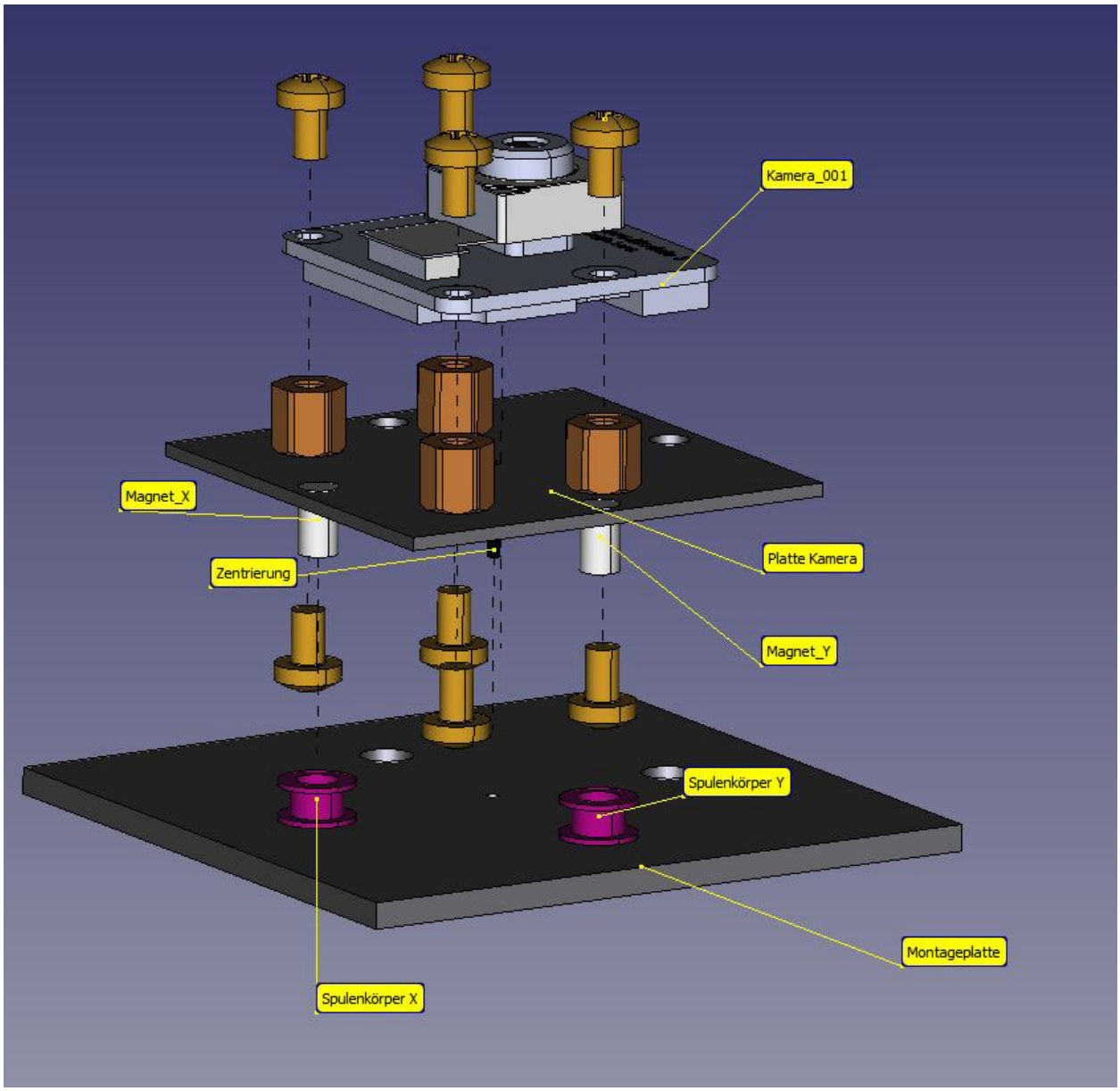

Figure 4.2: actors made of a magnet and a coil: explosion drawing. (drawing: Winfried Sembritzki)

robustness and even smaller size.

The plate with the camera carries a printed circuit board (PCB) with the acceleration sensors. They are arranged in an L shape. They are placed in three of the four courners of the quadratic plate/PCB. For measuring the rotational accelerations, we use the sensors' direction vertical to the plate only. The difference of the accelerations of two adjacent sensors describes the rotational acceleration around the axis perpendicular to their connecting line. One sensor is part of two such differential arrangements.[2] The axes of both arrangements are lined up with the axes of the actors.

We use thin, light, flexible and short individual wires for connecting the movable camera plate with the main body of the test bed. These wires replace the thick and stiff cables of the previous version. Both the sensors' SPI bus and the camera's USB are connected in this way. The short length allows to have no shielding for the USB wires. We use a stripboard mounted close to the camera plate for having a mechanically robust starting point for the conventional further cabling.

We use softer, 3D-printed feet, instead of the off-the-shelf feet of the first version of the test bed.

[2]Sensor noise can be reduced by using all three sensors for each axis: instead of computing the difference between a sensor A and a sensor B, we can compute the difference between the sensor A and the mean of sensor B and sensor C. This improvement depends on a sufficient mechanical precision of the setup, such that both axes remain decoupled.

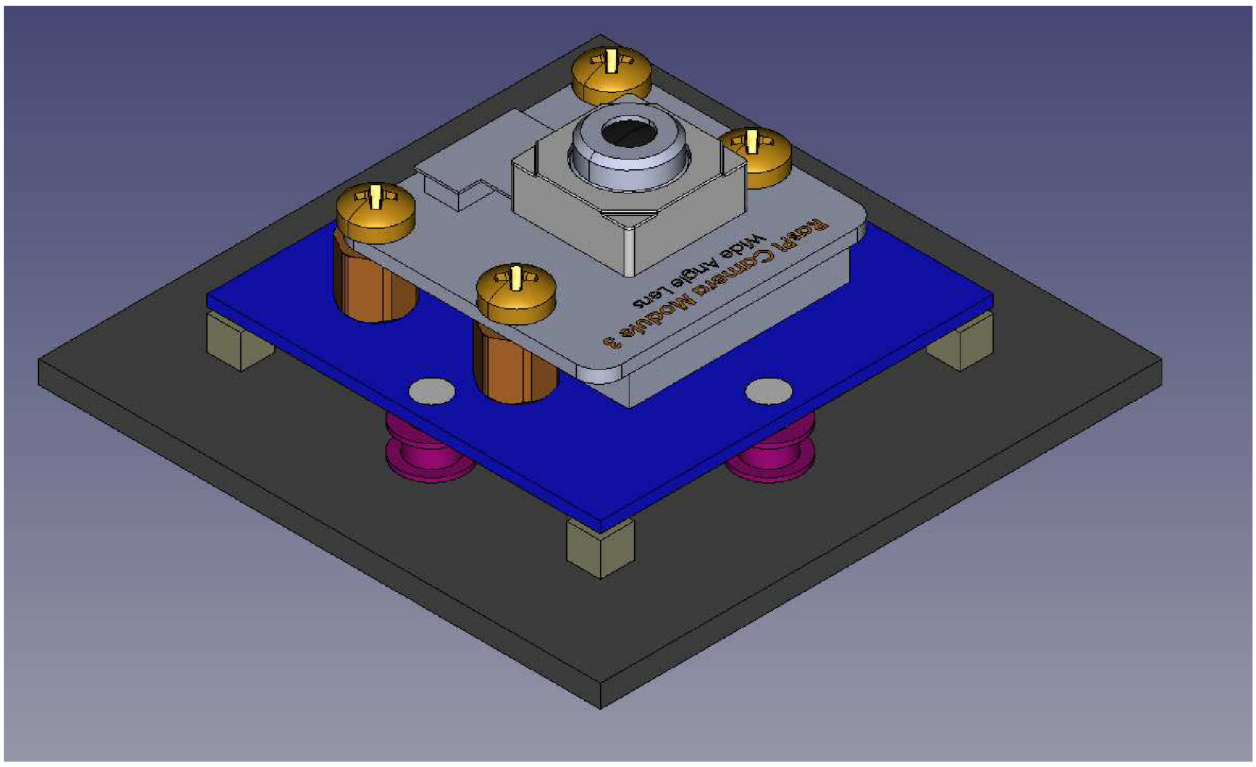

Figure 4.3: actors made of a magnet and a coil: assembled drawing. (drawing: Winfried Sembritzki)

We use none of the acceleration sensors of the first version anymore. The sensor on the camera platform has been removed. And the second sensor close to the motor is still there, but does not provide us with information of interest.

4.3 Partial Implementation in the Second Version

As stated, we could not implement all of the above ideas for the third version. We had to leave out the following aspects:

- The speakers are not yet in place. This is due to the fact that they do not fit into the 3D printed mount.

  Instead, we use the speakers of the first version, mounted into the testbed of the first version. This allows us to validate that the speakers generate movements (which should be audible). But there is no mechanical connection from the speakers of the first version to the sensors of the second version.

- The printed circuit board for the movable plate is not yet mounted on a flexible rod. Instead, it is mounted temporarily onto a stripboard. However, thin wires are used already.

- The softer, 3D-printed feet are still missing.

- The camera (dummy) is still missing.

In consequence, the mechanical realization allows to implement the electrical part from sensors to actors, but the mechanical loop from the actors back to the sensors is in concept stage only.

Figure 4.4 on the next page shows the printed circuit board for the movable plate. In the photo, the green printed circuit board (PCB) is mounted onto a yellowish stripboard temporarily. Please note the thin wires leading off the printed circuit board. Figure 4.5 on the following page shows the same PCB, and also the adapter at the other end of the cable from the stripboard.

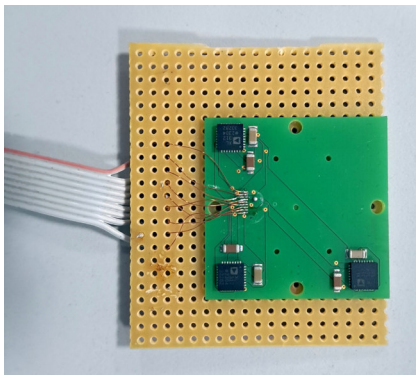

Figure 4.4: printed circuit board for the movable plate of the second version of the test bed. (photo: Winfried Sembritzki)

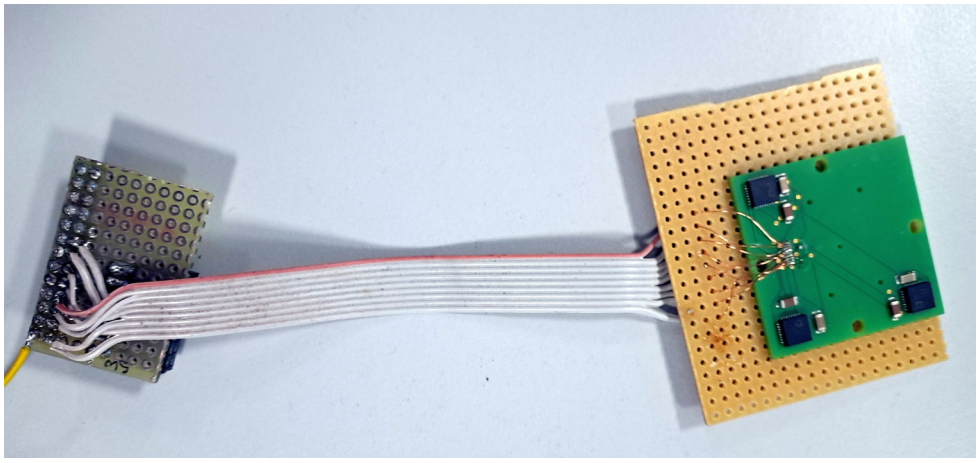

Figure 4.5: printed circuit with the sensors and also the adapter at the other end of the cable from the stripboard. (photo: Winfried Sembritzki)

4.4 The Electrical Interfaces of the Test Bed

The electrical interfaces of the third version of the test bed comprise:

- SPI bus for 3 sensors with 3 sub-sensors each,
  from stripboard to PYNQ board:
  1 bus (4 wires) plus 3 chip select wires
- USB of camera, from stripboard to PYNQ board
- 2 audio connections, from amplifier to PYNQ board:
  3 wires total, headphone jack
  (see Fig. 4.6 on the next page)
- motor power, from motor to (still missing) power driver:
  2 wires
- PWM motor signal, from (still missing) power driver to PYNQ board:
  2 wires
- USB for power and FPGA loading, from PYNQ board to external

- ethernet, from PYNQ board to external:
  (as soon as the microcontroller leads out measurement data
  and leads in PID parameters and motor speed parameter)

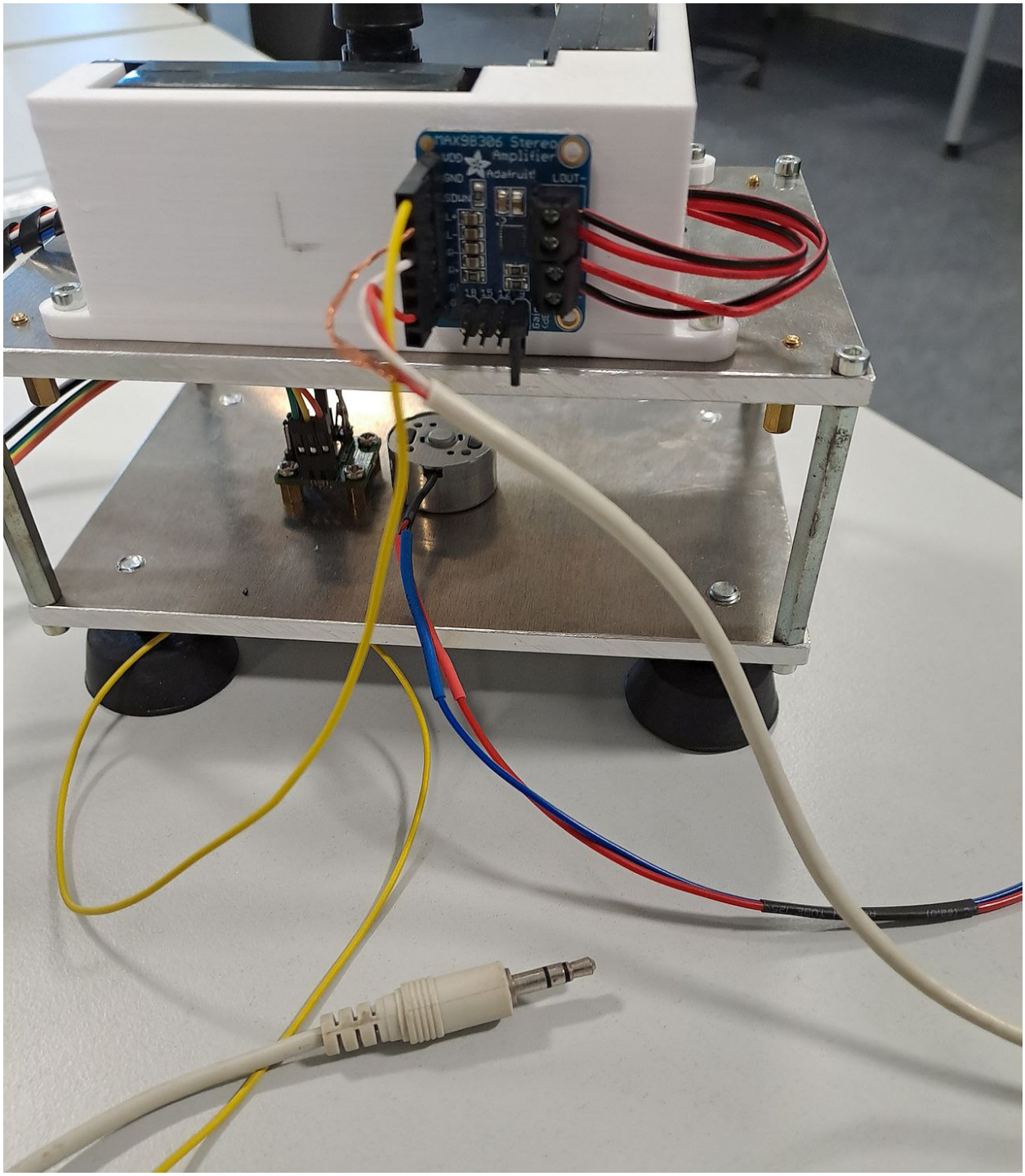

Figure 4.6: the electrical connections of the audio amplifier. (photo: Winfried Sembritzki)

4.5 Research Question for the Second and Third Version of the Test Bed

For our iteration with the second and third versions of the test bed, we reduce our research question in Sect. 1.2 on page 1 to:

How can an active control stabilize a camera-like dummy using an FPGA-based digital circuit, disturbed by vibrations at one frequency close to 1 kHz?

4.6 Measuring Vibrations Around the Third, Line-of-Sight Axis

Our arrangement of the sensors pays us a bonus: we can measure the rotational vibration around the third axis, which is the line-of-sight of the camera. We have no actor to counter this rotational vibration. But we can measure its angular amplitude, and we can calculate whether this angular amplitude causes the pixels in the camera to shift less than one pixel wide.

For measuring, we use the sensor's sub-sensors which are in the plane of the camera plate. (Each sensor consists of three sub-sensors in three differen directions.) Again, the difference of the accelerations of two adjacent sensors yields a value for the rotational acceleration. The difference of the accelerations of the two non-adjacent sensors, for two perpendicular directions, yields another value for the rotational acceleration. In total, we get three values from six applicable sub-sensors. By taking the mean value from these three values, we can reduce the noise of the sensors. It is advisable to calculate the total rotational acceleration directly from the values of all six sub-sensors in order to give each of them the same weight.

Chapter 5

Theoretical Background

5.1 AXI

written by Simon Pfennig

The Advanced eXtensible Interface (AXI) is a Protocol for communication between a Master and a Slave component. The communication is always initialized and controlled by the Master. There are two types of Transactions, Read and Write, which allow the Master to either read or write the contents of registers in the Slave Component. See [AMD23] for the specification. Data is transmitted in words, where each bit of a data word has its own line.

5.1.1 AXI-Lite

AXI-Lite is a bare-bones version of AXI. The mechanisms described below are true for AXI-Lite as well as the regular AXI. Features not used by AXI-Lite, like Transaction-IDs and Regions, are not used in this project and are therefore not described.

Read transactions are facilitated by two channels: Read-Address and Read-Data. The transaction is initialized by the Master sending the address of the register it wants to read on the Read-Address channel followed by the Slave responding with the corresponding data on the Read-Data channel.

Write transactions are facilitated by three channels: Write-Address, Write-Data, and Write-Response. The Master will first write the address he wants to set on the Write-Address channel, followed by the data on the Write-Data channel. Afterwards, the Slave will respond, on the Write-Response channel, with a status code indicating whether the transaction was successful.

Before something is written on any channel a handshake (see Figure 5.1 below) is performed to ensure both sender and receiver can process the data. Therefore every channel has, aside from the actual data signals, a Ready and Valid signal. The Valid signal is set by the sender when the data on the data signals is set to the actual data that is to be transmitted. The Ready signal in turn is set by the receiver to indicate it is ready to process the data. On the first rising edge of the clock when both are set the data is transmitted and both Valid and Ready are set to 0.

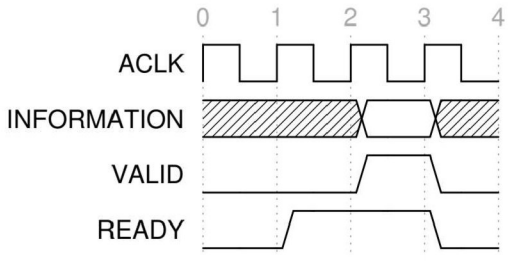

Figure 5.1: AXI Handshake [AMD23]

5.1.2 AXI-Stream

AXI-Stream (see [AMD21] for the specification) is a version of the AXI-Protocol that only uses one data channel. The only mandatory components are the clock and reset signals, as well as the data signals, the TValid, and TReady signal. Since there are no addresses involved and the Slave can not send but only receive data, AXI-Stream can be used for high-throughput one-directional use cases. This project also uses the TLast signal which is used to indicate that the transmitted data packet was the last one in a sequence.

5.1.3 AXI-Interconnect

The AXI-Standard only describes the communication between a single Master and a single Slave. But in practice, a single Master often needs to control multiple Slaves, or multiple Masters control the same Slave. An Interconnect is a component that enables these types of communications. It has multiple AXI-Slave and AXI-Master ports for various other components to connect to. When a master addresses a Slave, the address data is used by the interconnect to decide to which Slave it should forward the message to and vice versa. It also arbitrates when multiple communications are requested at the same time. In this project the "AXI Interconnect v2.1 LogiCORE IP" is used. See [AMD22a] for the documentation. This IP-Core also implements a translation between different AXI-Versions spoken by Master and Slave. This allows AXI-Lite components to communicate with regular AXI components.

5.2 SPI

written by Joscha Knobloch

Serial Peripheral Interface (SPI) is a synchronous bus meant for connecting multiple peripheral devices with a single host device. It uses a single data line per direction, a separate clock line, and chip select lines per sub.

Different naming schemes are common and a renaming is ongoing in the industry. The common naming before the current renaming is Master for the host device, Slave for the peripheral device, MISO(Master in Slave out), MOSI(Master out Slave in), SS(Slave select), and CLK(Clock). Current renaming efforts commonly use Main(formerly Master), Sub(formerly Slave), SDO(Serial Data Out, formerly MOSI), and SDI(Serial Data In, formerly MISO). The naming for the clock line is usually kept.

The main device controls the clock line, which is used to indicate a transmission. When the main device does not intend to transmit data, the clock is kept idle. This design means that the transmission is always bidirectional and the protocol over top is in charge of discarding unneeded transmissions in one direction.

To access any of the connected subs the main has individual chip select lines going to each sub. This allows for a trivial sub-selection but increases the need for output pins on the main.

There are 4 ways to interpret the clock signal which need to be equal on both ends of a communication. This sometimes requires a reconfiguration of the interface when accessing different sub-devices on a single bus. Clock polarity determines whether the clock is high or low when idle. Clock phase determines whether the data needs to be valid on the first or the second edge.

Internally SPI interfaces are often implemented with a shift register on each side of the communication forming a virtual ring buffer.

Oftentimes subs use a simple protocol to allow read and write access to a memory region where the first byte in a transfer indicates the direction and memory address(max 7 bits). This is not part of the SPI standard and needs to be checked for every used sensor. Some may have an extra bit for multi-byte reads/writes.

5.3 The Processing System and Python

written by Benjamin Bissendorf

Python is a programming language known for its accessibility to people new to programming and thus its large user base. There is a wide range of provided, community-supported modules enhancing the ecosystem.

The PYNQ-Z2 is a development board using features of the PYNQ framework by Xilinx. It is a Python-based development framework that allows the combination of normal Python programs and FPGA circuits to communicate. The board uses the ZYNQ XC7Z020-1CLG400C chip, which is a SoC that includes an ARM® Cortex®-A9 dual core processor and an FPGA. These subsystems are called the Processing System (PS) and the Programmable Logic (PL), respectively.

As Xilinx describes it, the power of this board is the combination of both the PS for standard tasks benefiting from an OS like communicating with USB-devices or communication via Ethernet, and the use of the PL for demanding or time-critical tasks [Adv22, cf.]. For interfacing between both systems, Xilinx provides the python package "pynq" that contains classes and functions to enable communication between these systems. For this, the "Overlay" class can be used, which dynamically loads a bitstream file into the FPGA and provides access to the design definitions such as AXI addresses. In the default operating system image, a Jupyter Notebook instance is included, enabling writing code via a web interface on the board and immediately starting the written code.

5.4 Sensors

written by Benjamin Bissendorf

The ADXL312 is a 3-axis digital accelerometer sensor package made by Analog Devices. Its subsensors each consist of a spring-suspended structure, acting as a differential capacitor and thus providing values proportional to their acceleration. To fit a wide range of applications, the chip is highly configurable and allows different data-rates, -resolutions, -ranges (ranging from $\pm 1,5g$ up to $\pm 12g$), and customizable data alignment. The sensor is able to measure the current acceleration up to 3200 times a second.

The onboard chip provides access to these settings and values via serial communication interfaces such as I²C and SPI at predefined register addresses. For the used SPI communication, 3-wire and 4-wire configurations are supported while using the clock-phase = 1 and clock-polarity = 1 configuration. Have a look at its datasheet for an elaborate listing and explanation of the available registers (see [ADX22, page 19]).

Although not in the scope of this project, the chip offers different ways of saving power. While using lower data rates already reduces the chips' current substantially, it also offers an additional "low-power"-mode which can be enabled in special registers. This additionally reduces the drawn current by about 30% but comes with an increase in noise. This might be useful in projects where the amount of available power is limited.

5.5 Actors

written by Niklas Seeliger

The camera platform is mounted on a central rod that is stiff enough to hold the platform in place when no other forces are enacted. It is however flexible enough that it can bend slightly which allows rotation of the entire platform. Mounted to the platform 90° from each other are rods connecting to actors that can push the platform up and down. Because the center is fixed in space by the rod in the middle, this causes rotation along one axis. The two actors can move the platform independently from each other in the two relevant axes.

These actors have to be able to move the platform very rapidly, that is with a high frequency, and with high precision but only a small distance. For those reasons, speakers were selected. Since speakers need to generate a very specific sound wave they have to move in a precise manner. Speakers are usually rated to generate all frequencies in the audible spectrum from $20Hz$ to $20kHz$. This is more than enough to correct vibrations up to $1kHz$. On the low end, it might even be enough to correct for some orbital movement, given the shutter speed is high enough. With the exception of the membrane that couples the actor to the air, the speakers are equivalent to a voice coil actuator. A mobile coil mounted around a static permanent magnet. When current passes through the coil the induced magnetic field interacts with that of the static magnet and the coil is moved linearly through a track it is mounted in.

Using speakers as the actors has the added benefit of being able to use dedicated audio hardware to generate analog signals. In the same vein, we can use standard equipment such as amplifiers and audio jacks. Since moving the platform simplifies to playing a sound we can make use of the on-board audio chip on our hardware. Using readily available hardware also lowers cost and development time.

Driving the audio signal is different from typical applications only in that we cannot buffer any audio data as that would introduce a significant delay to the feedback loop which would make the control loop significantly more difficult to impossible. Driving an analog signal from a digital System introduces some delay in the digital-to-analog conversion which we cannot avoid.

Chapter 6

Concept for the digital control system

This section describes the general concept and structure of our solution.

In figure 6.1 you can see an overview of the components. Highlighted with thicker arrows, the main controlling path can be seen: The "Sensors" provide information about the acceleration of the contraption, which is polled by the "Input Interface". It combines the data into a single stream and pushes it into the "Validation Data" component and the "Digital Controller" component. The latter one uses this information to approximate the current deflection of each sensor and outputs counter-steering signals which are transformed and sent via the "Output interface" to the "Actors". For testing purposes, the user has the ability to either provide different controlling parameters for the "Digital Controller" via the "Configuration" component or read the past sensor readings which are written by the "Validation Data" component, written into a text file in the filesystem.

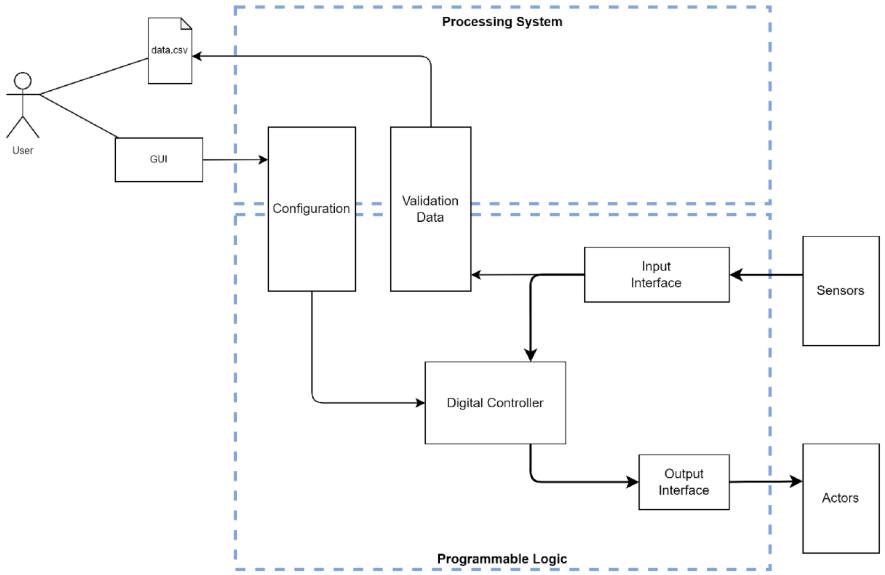

Figure 6.1: The structure of our concept for the digital control system

6.1 Configuration

written by Simon Pfennig

The Configuration Interface is used to send data from the PS to the PL. This can, for example, be used to adjust the parameters of the Controller during System Validation. For this purpose, the PS-General Purpose Input / Output (GPIO) Pins are used. These pins can be accessed from both PS and PL. In this project, a function, provided by the Jupyter-Notebook environment on the PS, is called to write data to an IP-Block which will hold the data in a register for further use by the PL.

6.2 Input Interface

written by Joscha Knobloch

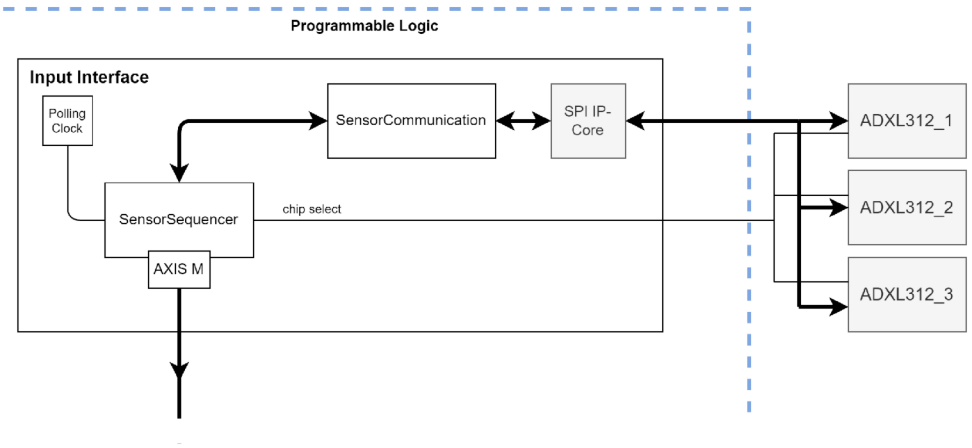

Figure 6.2: Internal structure of the Input Interface (BB)

The Input Interface is designated to retrieve data from the sensors and forward them to the "Validation Data" and "Digital Controller". The sensors are accessed by a common SPI Bus with individual chip select lines for each sensor. Externally a single AXI Stream interface can be used for the output as both the "Validation Data" and "Digital Controller" blocks need this interface.

Internally the Input Interface is broken up into a few blocks to reduce the complexity in any one block. The Polling Block is used to generate a clock signal for polling the sensor data. This is used by the SensorSequencer block, which controls the chip select lines and gives commands to the SensorComunication block. The commands are given by pulsing special specified lines and can be used to either configure a sensor or retrieve data from it.

The SensorCommunication block uses AXI4 Lite to communicate with the SPI IP-Core from Xilinx which then uses SPI to communicate with the sensors.

The SensorCommunication block does not control the chip select lines of the sensors. These will always be set accordingly by the SensorSequencer block, so the SensorCommunication block can communicate with the sensors with minimal extra logic. The SensorCommunication block is not aware that there are multiple sensors.

From the SensorCommunication towards the SensorSequencer, an AXI Stream interface is used as well.

6.3 Digital Controller

written by Janek Brumund

This section describes the concept for the Digital Controller unit. The following figure shows the detailed structure of the Controller (see Figure 6.3).

It consists of multiple components:

- Low Pass Filter
- Deflection Estimator
- High Pass Filter
- Controller

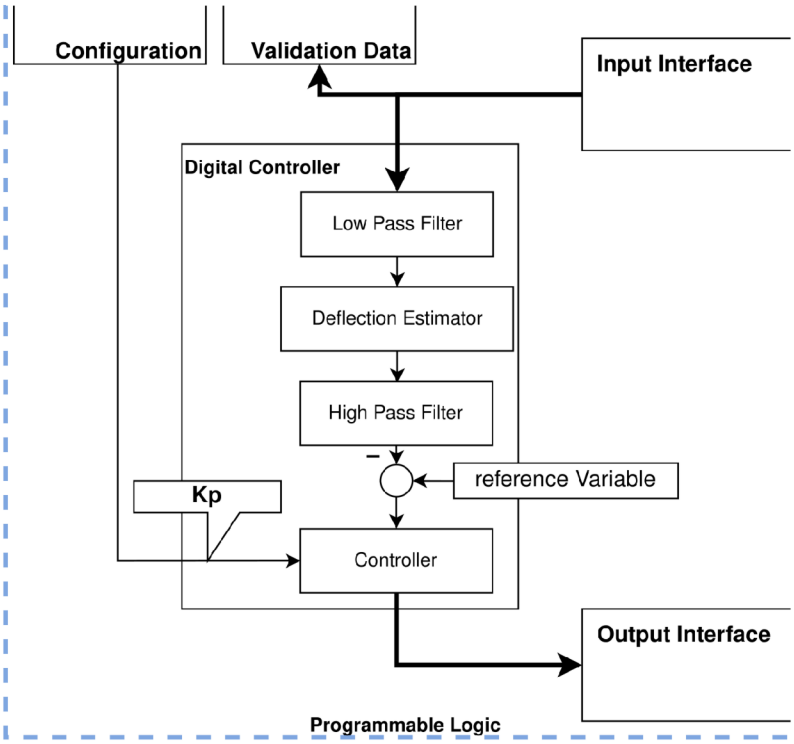

Figure 6.3: The structure of the concept for the digital controller

6.3.1 Filters

The "Low Pass-" and "High Pass Filter" are digital filters. They will pass signals and signal components with a certain frequency. The **Low Pass** filter filters all signal components with a higher frequency as configured. Respectively, the **High Pass** filter filters all components till a certain frequency. Use cases for each filter are:

- "Low Pass Filter": preventing control of fast signals that would lead to errors
- "High Pass Filter": preventing control of slow signals that could not be processed by the actors

Each filter should be implemented using an Intellectual Property (IP) core called "FIR Compiler" (see [AMD22b]). Finite Impulse Response (FIR) filters are non-recursive filters which means that they are inherently stable [cf. EVN24, p.264]. They depend only on the current and previous input values.

6.3.2 Deflection Estimator

The "Deflection Estimator" is used to calculate the current deflection of the platform. To do so, the measured acceleration is transformed. This is possible because the following equations (6.1 and 6.2) taken from [Men13, p.33] are valid :

$$s(t) = s_0 + v_0 t + \frac{1}{2} a t^2 = s_0 + \frac{1}{2}(v + v_0)t \tag{6.1}$$

$$v(t) = v_0 + at \tag{6.2}$$

The current velocity is calculated using the equation 6.2. With

$$v = at \text{ [cf. Men13, p.38]} \tag{6.3}$$

and equation 6.1 follows equation 6.4.

$$s(t) = s_0 + \frac{1}{2}v(t) \cdot t \tag{6.4}$$

The t in this equations is the sampling time t_a which is the inverse of the sampling frequency f_a.

$$t_a = \frac{1}{f_a}$$

As described in [ADX22] the measured acceleration will have a sensitivity of $2,9\frac{mg}{LSB}$. With

$$mg = g \cdot 10^{-3} = 9,81 \cdot 10^{-3}\frac{m}{s^2}$$

the unit for the acceleration a will be

$$[a] = 2,9\frac{mg}{LSB} = 2,9 \cdot 9,81 \cdot 10^{-3}\frac{\frac{m}{s^2}}{LSB}$$

This means, one Least Significant Bit (LSB) represents $2,9 \cdot 9,81 \cdot 10^{-3}\frac{m}{s^2}$. With equation 6.3 and $k = 2,9 \cdot 9,81 \cdot 10^{-3}$ this leads to

$$\begin{aligned}[v] &= [v_0] + [a] \cdot [t] \\ &= k\frac{\frac{m}{s^2}}{LSB} \cdot s \\ &= k\frac{\frac{m \cdot s}{s^2}}{LSB} \\ &= \frac{\frac{m}{s}}{LSB}\end{aligned}$$

The factor k is omitted because it has no influence on the unit. With $s = \frac{1}{2}vt$ [cf. Men13, p.38] this lead to

$$\begin{aligned}[s] &= \frac{1}{2} \cdot [v] \cdot [t] \\ &= \frac{\frac{m}{s}}{LSB} \cdot s \\ &= \frac{m}{LSB}\end{aligned}$$

Like k, $\frac{1}{2}$ has no effect on the unit. That's why this factor is eliminated.

In consultation with Professor Bredereke, the time for t is set to 1s. This is possible because the sampling time is constant and can be eliminated or set to 1s.

In figure 6.4 the location of all sensors is specified.

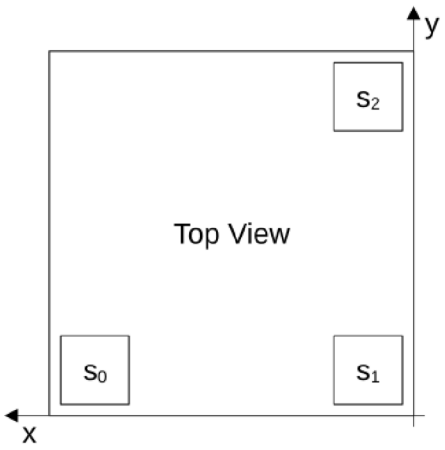

Figure 6.4: Top view of PCB – specification of sensor numbers

To determine the deflection on the x-axis the figure 6.5 serves as an illustration. The Printed Circuit Board (PCB) is shown in front view. Given a positive acceleration a_x the platform will be inclined to the left side. If a negative acceleration is given the platform tilts the other way around. This means there will be a positive deflection if the platform is inclined to the left.

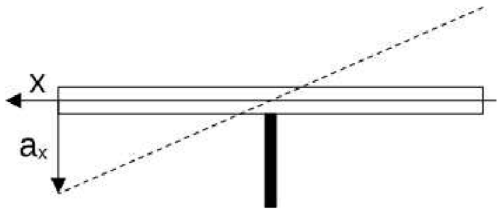

Figure 6.5: Positive acceleration on x axis

The deflection on the y-axis is illustrated in figure 6.6. The PCB is shown in side view. If a positive acceleration a_y is given the platform will be inclined to the back. If a negative acceleration is given the platform tilts the other way around. This means there will be a positive deflection if the platform is inclined to the back.

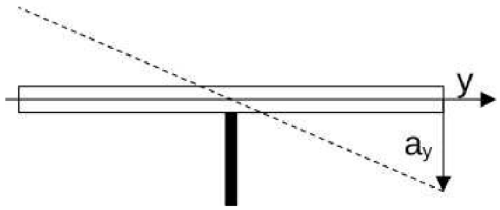

Figure 6.6: Positive acceleration on y-axis

The measured and used acceleration is the acceleration from the z-axis of the sensors. The acceleration due to gravity is subtracted from acceleration on the axis. The result is the acceleration in the respective axis without the disturbance caused by the acceleration of gravity. This must be done because of the alignment of the platform. With this acceleration, the new deflection is calculated and this is added to the previous deflection. The result is the current deflection.

Ideas to determine the direction of deflection:

1. Calculate the difference between two acceleration values from different sensors.
2. Compare two acceleration values with $g = 9,81\frac{m}{s^2}$ and calculate the deflection with the value which

is higher than g acceleration after subtracting g. Depending on which sensor is used the result is either added or subtracted from the total deflection.

Example for the first idea:

$$\begin{aligned} a_x &= a_{x,0} - a_{x,1}, \text{ with } a_{x,0} = 0,5\frac{m}{s^2} \text{ and } a_{x,1} = -0,5\frac{m}{s^2} \\ &= 0,5\frac{m}{s^2} - (-0,5\frac{m}{s^2}) \\ &= 1\frac{m}{s^2} \end{aligned}$$

In this case, the result is twice as high as expected because both values were identical in amount. It is possible to divide the value by 2 to get the correct value. However, if the values are not the same in terms of amounts, it is no longer so easy. For this reason, we will work with the second idea.

6.3.3 Controller

To generate a signal that could be used to control the actors, the "Controller" will be used. As mentioned earlier, we chose the PID controller to control the system. Since there is no standard and easy-to-use IP core for the PID controller, it was decided to implement only a P controller.

With respect to [Phi22] a controller is a module that gets the reference signal $w(t)$ and substracts the controlled signal $x(t)$ to generate the error signal $e(t)$ ($e(t) = w(t) - x(t)$). The used control element takes the error signal and calculates the actuating signal $y(t)$. The system could be affected by the disturbance signal $z(t)$. Figure 6.7 shows this relation in a control loop.

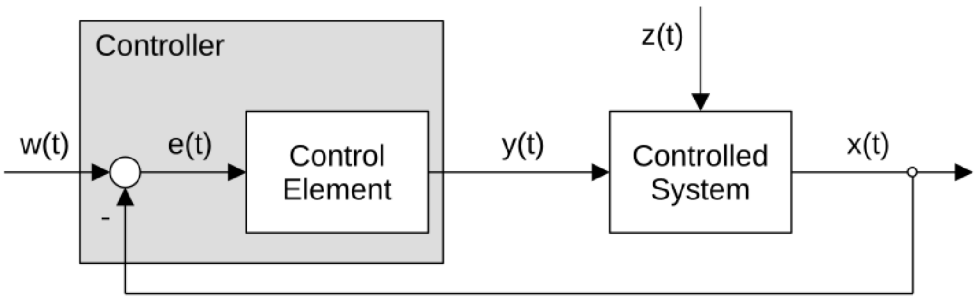

Figure 6.7: Control loop – relations of the different signals

The control element could consist of multiple elements. A PID controller consists of a proportional (P), an integral (I), and a differential (D) component. According to [Phi22, p.132], the equation of time for this control element is equation 6.5.

$$y(t) = K_P e(t) + K_I \int_0^t e(t)dt + K_D \frac{de(t)}{dt} \tag{6.5}$$

To implement a PID controller the above equation must be implemented in PL. Because of the complexity of implementing this equation, the above decision (using P instead of PID controller) was made. It is also assumed that a P controller is sufficient, as the micro-vibrations of a constant speed of the motor are currently to be controlled. This assumption is made because it is assumed that a constant rotational speed results in a constant deflection of the platform.

The components for each element are:

- P: $K_P e(t)$
- I: $K_I \int_0^t e(t)dt$

- D: $K_D \frac{de(t)}{dt}$

This means, that to implement a P controller the equation 6.6 must be implemented:

$$y(t) = K_P e(t), \text{ with } e(t) = w(t) - x(t) \tag{6.6}$$

Which is a multiplication of a (constant) factor K_P with the error signal.

Related to the previously described "Deflection Estimator", this means that the error signal is the current deflection of the platform.

6.4 Output Interface

written by Niklas Seeliger

The output interface is responsible for taking a new state for the actors and sending it to the audio chip (an ADAU1761, data sheet is available online: [Ana10]) as digital audio data. There it is converted to an analog signal and is modulated to the the headphone jack output. From the controller, the interface should receive two AXI-Streams (one for each axis). The received data is buffered until it can be sent to the audio chip. A more detailed view of the Output-Interface and how it interfaces with surrounding components is shown in Figure 6.8.

The audio chip requires two lines of communication to work properly. It receives the audio data as an Inter Integrated Chip Sound (I²S) Stream. This stream contains both audio channels (referred to as left and right channels). The audio chip also needs to be configured. The audio chip has registers internally which need to be written to the desired configuration. This can be done either through an SPI or an I²C interface.

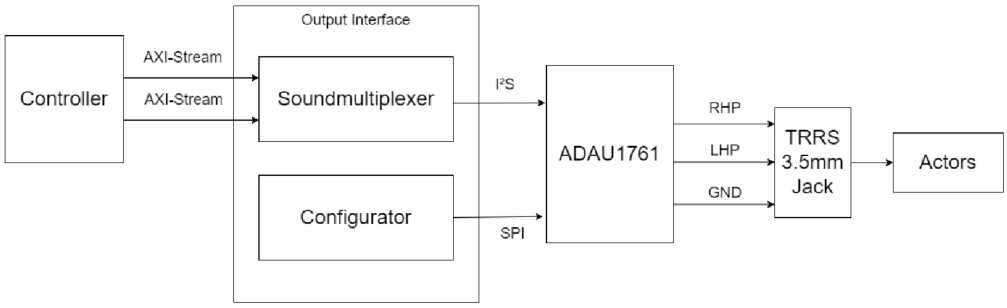

Figure 6.8: A detailed view of the output interface

6.4.1 I²S

I²S is a serial communication protocol designed for inter integrated chip sound data. The protocol is made up of two clock lines (serial clock, and word select line respectively) and one data line called serial data. The full specification is available online under [NXP22]. A brief summary is provided here.

The chip driving the serial clock and word select line is called the controller. The chip driving the serial data line is called the transmitter. The chip reading the serial data line is called the receiver. Either the transmitter or the receiver may the the controller (an external controller is also possible).

The **serial clock** line is the fastest clock in the interface. It is used to synchronize the bits of the audio data. With each rising clock edge a bit is read from the serial data line. The falling edges on this line are intended to trigger the next bit to be written to the serial data line by the transmitter. The data is transmitted in a twos complement format and can have any length of bits. Each word starts with the

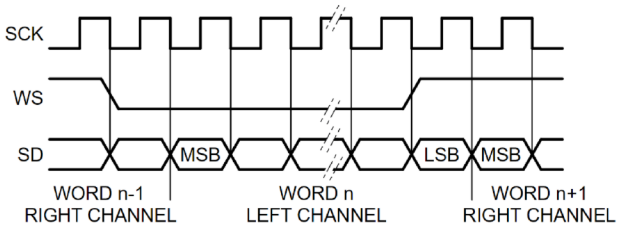

Figure 6.9: I²S basic timing (Image taken from [NXP22])

most significant bit (the sign bit). This line is often referred to as the bitclock or BCLK line because it triggers the next bit.

The **word select** line is used to multiplex between different audio channels. In the simplest form, these are the left and the right audio channels. When the word select line goes to a logical 0 the left audio channel will be transmitted and when it goes to a logical 1 the right audio channel will be transmitted. This channel will be transmitted for as long as the word select line the value that corresponds to it. The value can only change with falling serial clock edges. After the word select advances to the next channel, the next bit that will be transmitted is the least significant bit of the previous channel. The transmission of the next most significant bit is delayed by 1 bitclock cycle. Figure 6.9 shows this delay. This line is often referred to as the left-right clock or LRCLK because it always alternates and triggers the transmission of the next channel.

The **serial data** line is the line on which the audio data is transmitted bit by bit. It is written during the falling edges of the serial clock and read on rising edges. It is not specified how many bits the audio data may be. The number of serial clock cycles for each word select channel may be shorter or longer than the serial data being transmitted. The transmitter therefore needs to be ready to change to the next word before the current word is fully transmitted. Likewise, it needs to be able to transmit more bits than the audio data has (for example by buffering with zeroes).

The serial clock frequency can be calculated with the data width and the sampling rate. For a sampling rate of $48kHz$ (typical for sound) and a data precision of 24 bit provided the word select allows for exactly the transmission of as many bits as are needed the serial clock frequency can be calculated as shown in Equation 6.7.

$$\begin{aligned} f_{sclk} &= f_{sampling} \cdot precision \cdot 2 \\ &= 48\ kHz \cdot 24 \cdot 2 \\ &= 2.304\ MHz \end{aligned} \tag{6.7}$$

6.4.2 Configurator

The Configurator component is responsible for configuring the ADAU1761 chip. This includes general configuration such as enabling its clock as well as configuring the I²S interface and modulating the received data to the headphone output that is used to interface with the actors. A detailed list of registers that need to be written can be seen in Appendix C. The configuration should be written immediately after a system reset. To configure the audio chip the SPI protocol was selected (as opposed to I²C) because it is used to read the sensors and because it offers much higher throughput than I²C.

6.4.3 Soundmultiplexer

The sound multiplexer takes two AXI-Streams and outputs a time division multiplexed I²S stream. Since the throughput of the two AXI streams is far greater than the 48 kHz of the I²S Stream. Some values need to be discarded and cannot be sent to the audio chip. When a new I²S Word begins the last received data on that channel should be buffered to avoid data inconsistencies that might arise from changing

data while it is being serially sent over I²S to the audio chip. This introduces a time delay of at least 1 left-right clock cycle ($1/48\ kHz \approx 20.83\ \mu s$). This component should always be ready to receive data from the AXI Stream because the latest data should always be used to keep delay low. On the I²S side there is no reason to wait for the chip to be fully configured. The stream is valid even if the receiver has no knowledge of the last datum. Therefore no communication to the configuration component is required.

6.5 Validation Data

written by Benjamin Bissendorf

For testing and validation purposes, the data provided by the sensors are to be recorded and written into a file in the file system, following a defined data structure to read it afterwards. This PL module is dependent on the data provided by the "Input Interface": Each time new data is available on the incoming stream, it is written to memory for the slower PS to collect and write into the filesystem.

To achieve that, a combination of logic in the PL and the PS is used. An overview of the structure is given below in figure 6.10. The stream data, consisting of all the sensor data, is connected to a component splitting the data into 64 Bit chunks. Those are buffered and then written into memory. This data is extracted and the raw data is permanently written into a file, to be later postprocessed into a human and machine-readable CSV file. The details of implementation are documented in chapter 11.

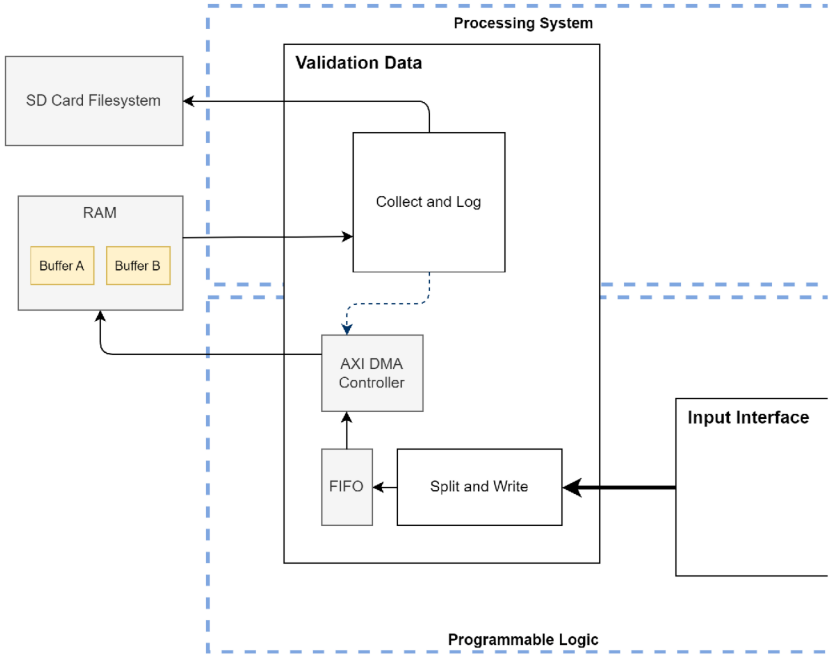

Figure 6.10: A schematic of the internal structure of Validation Data

Chapter 7

Implementation of Input Interface

written by Joscha Knobloch

This section describes the implementation of the input interface which itself is composed of a few blocks as described in section 6.2 and visualised by image 6.2 on page 18. The blocks seen in the image are represented by individual blocks in the Vivado block design. The SPI core is provided by Xilinx. The SernsorCommunication block was started with the "Create and package new IP" Wizard in Vivado and is written in VHDL. The PollingClock- and SonsorSequencer blocks are written in VHDL as well.

7.1 Sensor Communication

written by Joscha Knobloch

The SensorCommunication block connects to the SPI IP-Core via AXI Lite on the one side and the SensorSequencer block via AXI Stream on the other side.

Figure 7.1: Screenshot showing the custom Vivado IP core with its ports

The code is attached in the appendix. It consists of a top level file A.6 on page 78 and the main logic in a separate file A.7 on page 81.

Due to time constraints, this block is not finished and could therefore not be integrated into the overall solution for testing. However, simulations show that the AXI Lite interface is very likely working as expected. Due to tests done with Python as the AXI Master connected to the SPI IP-Core, the SPI functionality could also be shown working. As the SensorCommunication block is designed to use the same procedure on the AXI bus it is very likely that the SPI communication also works.

Therefore the only thing left would be to set up the specific sensor registers and acquire the sensor data. This should not be a very big task. Unfortunately, we could not finish it.

There is very little good information on how to use AXI Lite with VHDL on the internet. One could look up the details on the handshaking and implement the interface themselves. As the bus consists of five

independent channels with their own handshaking some sites recommend starting with the template that Vivado provides when using the "Create and package new IP" wizard for an AXI Lite Master core.

The provided template design is unfortunately not well documented and interwoven with an example that writes and then reads specific data to/from specified registers. It is very unclear how the template is supposed to be used. It specifies locations for “User logic”, however, it is necessary to alter code outside of those areas to get the logic working at all. To make use of the provided code one needs to go through, understand every line of code, and then adapt it. This might be more work than implementing the interface from scratch.

In the concept phase of the project the SPI IP-Core was selected as AXI is the standard protocol to be used between IP-Cores. Due to the complexity of using AXI Lite it could have been easier to implement the SPI interface directly in VHDL.

For comprehensibility, the AXI Lite interface, the handling of the SPI-Core and the handling of the acceleration sensors are described separately.

7.1.1 AXI Lite interface

The core contains one process per channel to facilitate the handshake. There are five channels in an AXI Lite bus as described in the theoretical background at section 5.1.1 on page 13. For channels going from the Master to the Slave, the valid flag needs to be set before the relevant clock cycle for the date to be read by the Slave. For the channels going from the Slave to the Master the ready flags need to be set to acknowledge the received data and signal the ability to receive data. These five processes were already present in the template, but some had to be modified heavily to be used for the general purpose.

There are also two processes for indicating a done read/write which were not present in the template. There is also a process that represents the AXI interface to the rest of the logic. It allows the next process to set a read/write bit, the address, and data if applicable, and then just trigger the transfer. A flag is pulsed high when the transfer is done and the results can be checked.

This part is very likely working as expected as it was tested in Vivado's simulator together with the SPI IP-Core. The handshake was visible and the response data made sense.

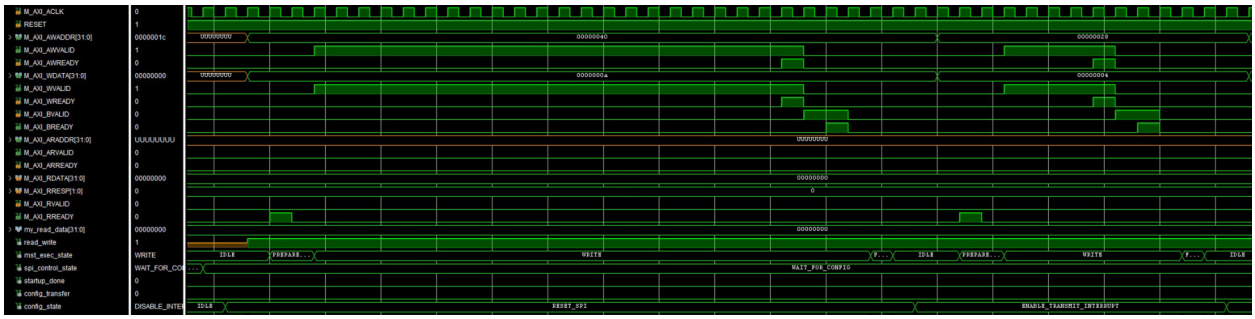

Figure 7.2: Screenshot showing the Vivado simulator with the AXI interface working

7.1.2 Handling of the SPI Core by Xilinx

To use the Xilinx SPI IP-Core it needs to be configured at startup. This is done by a procedure of AXI transmissions. Therefore the core contains a process that can configure the SPI core and command it to do SPI transmissions when the relevant flags are set.

To configure the SPI core a flag for configuring is set, a line is given a pulse to start the procedure, and a done flag is waited for. When configuring the core, the process resets the SPI interface, enables the transmit interrupt, disables the global IPIF interrupt, deselects the slave, reads the control register, modifies some flags, and writes the control register.

To make an SPI transmission one must only set a flag for SPI transmission, set the data to send, start

the procedure, wait for the done flag to be set, and retrieve the received data. While the transmission is ongoing the process sets the data register, sets the slave register, reads the control register, modifies some flags, writes the control register, waits for the transmission to finish by checking the status register, and then deselects the slave.

This logic could not be tested due to time constraints. However, the process's behavior stems from a Python script by Marakena Labs which we were able to test with an acceleration sensor. Therefore it is also very likely that the SPI interface is working. [mat21]

7.1.3 Handling of the acceleration sensors ADXL312

Finally, the acceleration sensors need to be accessed via SPI. In an earlier project stage, the ADXL355 sensor by analog devices was successfully sampled with the mentioned Python implementation for the SPI core. Due to time constraints, it was not possible to address the new sensor. To make it work it needs to be taken out of standby mode by setting some config registers. It can also be configured to use the smallest range of $\pm 1.5g$ and sample at a given rate.

7.2 Data Sequencer

written by Benjamin Bissendorf

The "Data Sequencer" is used to orchestrate the sensor configuration and sensor data retrieval process implemented by the "Sensor Communication" block.

Its outer interfaces are depicted in figure 7.3 below. To specify the rate at which the data of all sensors should be polled, an external enable bit must be periodically provided into the *trigger_poll* input. For the sensor configuration and data retrieval, the "Sensor Sequencer" is connected to the "Sensor Communication" block. For that, it uses the *comm_start_configure* flag to start configuring a sensor, while the *comm_start_data* flag is used to ask for the x-, y-, and z-axis data of one sensor. This data is returned into the *sensor_data* line, accepting $3 \cdot 16Bit = 48Bit$ of data. "Sensor Communication" can either signal that it and the sensors are configured with *comm_ready* or it can signal that the requested data is available with the *comm_data_ready* flag. When the data of all sensors are ready, they are published to *data_out* with the *data_out_valid* output set to '1'. The selection of which sensor is communicated with at the moment is the task of the Sequencer. For this the *select_sensor* vector is provided to connect to the SPI chip select pins of the sensors.

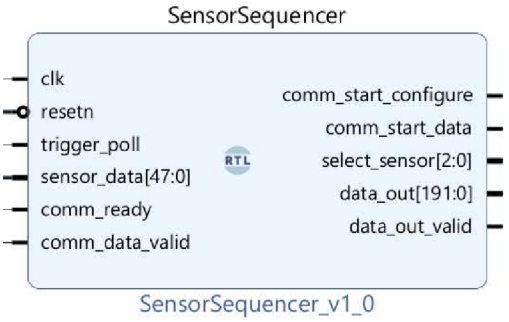

Figure 7.3: The Sensor Sequencer block

As one is familiar with the outer interfaces, the inner sequencing procedure is of interest. It was planned and implemented by following the deterministic finite state machine pattern, which can be seen in Figure 7.4. Generally, the block's sequence can be described by separating it into two parts: The configuration and the data retrieval.

After a system reset, the Sequencer waits for its counterpart, the "Sensor Communication", to be ready (comm_ready = '1'), as it has to configure the blocks it is dependent on, like the AXI SPI Master. When

this is done, the sensors have to be configured (polling rate, data resolution, etc.). The Sequencer starts the configuration for each of the sensors after another by alternating between the states *Configure* and *WaitForConfigured* until all are done. Then the state machine transitions into the state *WaitForTrigger*, belonging to the "Data Retrieval" section. This section represents the group of states in the standard operation of polling the sensor data. The state machine always waits for the external polling clock to trigger a new polling cycle. When this happens, the state machine transitions into the *RequestData* state. There the "Sensor Communication" is signaled to get the data of the current sensor and another transition into *WaitForData* happens. There the Sequencer waits for the data to arrive and to save it into the *data_out* vector. It transitions again to *RequestData* until all sensor data is collected. While transitioning from *WaitForData* to *WaitForTrigger*, the *data_out_valid* flag is set to start publishing the data into the following pipeline.

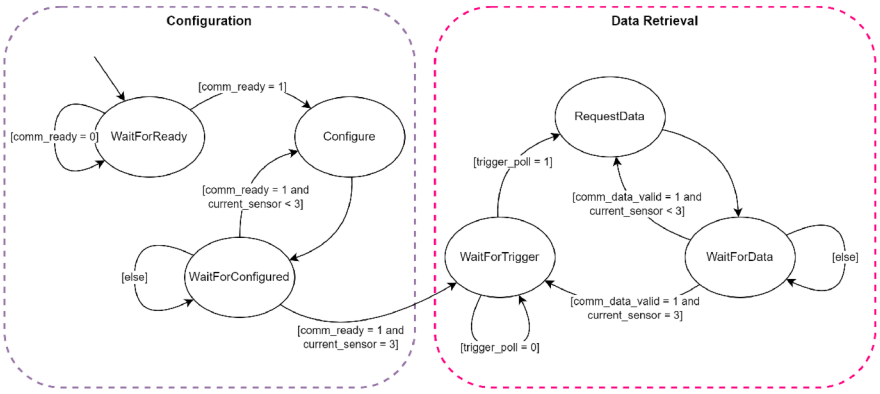

Figure 7.4: The finite state machine of the Data Sequencer

The complete VHDL implementation of this state machine can be seen in listing 9. A new VHDL type was defined consisting of the states in the previous diagram. Synchronous to the clock, states and their transition conditions are evaluated. One excerpt to point out can be found below in listing 1. It contains the definition of the transitions for the *WaitForData* state. When the sensor data is available, the incoming vector has to be written into a specific range of the output vector. The output vector is defined to have the sensor data in the order depicted in figure 7.5. As the sequencer polls *Sensor 1* first, its data has to be written into bits 191:144. This calculation is done in line 88 for $current\_sensor \in [1, 3]$.

```
84                  when WaitForData =>
85                      comm_start_data <= '0';
86
87                      if comm_data_valid = '1' then
88                          buffer_upper_bits := 64*3 - 1 - 64*(current_sensor-1);
89                          data_out(buffer_upper_bits downto
90                                  buffer_upper_bits - 16*3 + 1) <= sensor_data;
91
92                          if current_sensor = 3 then
93                              state <= WaitForTrigger;
94                              data_out_valid <= '1';
95                          else
96                              state <= RequestData;
97                          end if;
98                      end if;
```

Listing 1: SensorSequencer.vhd for the Input Interface component

Visible in the figure of the data format below are the empty spaces as each sensor has a range of 64 Bits of data, while only 48 Bits are used for the three axes. This was done to pre-format the data for easier logging in the Validation Data component and to enable easier extensions in future iterations.

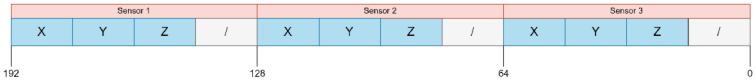

Figure 7.5: The data format of the outgoing stream

7.3 Polling Clock

written by Benjamin Bissendorf

As described in the previous section, the "Sensor Sequencer" needs an enable signal to indicate polling new data. The rate was determined to be 3200 times a second. The "Polling Clock" is the block implemented to reduce the incoming clock signal into a single enable output after a specified amount of ticks. As such, the outer interface is as simple as depicted in figure 7.6.

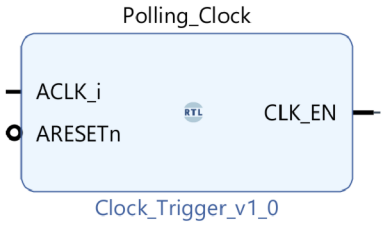

Figure 7.6: Our Clock_Trigger block used as the Polling_Clock

To calculate the number of ticks after which an enable signal has to be given, one has to know the system clock frequency and the polling frequency:

$$n_{ticks} = \frac{T_{poll}}{T_{clk}} = \frac{f_{clk}}{f_{poll}} = \frac{100.000.000Hz}{3200Hz} = 31.250$$

7.4 Sensor Dummy

written by Simon Pfennig

As the Input interface was not ready for integration with the other components, we decided to implement a Dummy Component (see Appendix 10) to emulate the sensor output for testing purposes. This component implements the interface with the Data Sequencer.

Internally it consists of three 16-bit counters which count from -500 to 500 and then reset. This implements a Sawtooth-Wave that can be sampled. The timespan, between each increase of the count can be configured and is independent of the polling from the Data Sequencer, as this aims to represent the actual variable that would be read by the sensor. Each of the counters can be initialized with a different offset, allowing a phaseshift between the signals.

The contents of the counters are exposed to the SensorData-Input of the Data Sequencer when it polls the data from the sensor Dummy. According to the SensorSelect-Signal one of the three counters will be used for that, simulating the request to different sensors. At this point, all three axes of a given simulated sensor show the same Sawtooth-Wave as output, since we only use one axis for testing.

Chapter 8

Implementation of Digital Controller

written by Janek Brumund

In this section, the implemented components are described. The complete block design is shown in Appendix D.1. There is the "Deflection_Estimator" on the left side of this figure. This block takes the measured acceleration on an Advanced eXtensible Interface (AXI) Stream interface and determines the deflection of the x and y axis (see section 6.3.2). The calculated data is passed on to two "Digital_P_Controller" blocks through an "AXI4-Stream Broadcaster" Intellectual Property (IP) core. The controller blocks calculate the actuating signal depending on the given reference variable ("Constant" core) and a given factor for K_P. The K_P is given through another IP core called "AXI GPIO" (see chapter 10). To determine which half of the data is passed on from the "Deflection_Estimator", two more "Constant" blocks are needed. One for a constant low level and one for a constant high level. The controller blocks pass the data to the "Output Interface" (see chapter 9).

8.1 Filters

To implement a digital filter ("High Pass-" or "Low Pass Filter") the "FIR Compiler" IP core is used. It is a highly configurable core that can take one or multiple sets of coefficients. The coefficients for each filter are calculated using the "pyFDA" ([CHI24]) tool. With this tool, it is possible to calculate coefficients for different types of filters. Multiple data paths can be configured. The width of the input is also configurable. Besides that the type of the coefficients is configurable. For more information on how to configure such an IP core see [AMD22].

To keep the calculations simple these components are not included in the complete design. This decision was made because using numbers with decimal places as coefficients leads to output values with decimal places. This makes the calculations needed in "Deflection_Estimator" and "Digital_P_Controller" more complex. To make it possible to solve the given problem in the given time the calculations must be easy enough to implement.

The designed filters could be reproduced using "pyFDA". All necessary files are available in the related repository on the GitLab server from the university.

8.1.1 Low Pass Filter

Figure 8.1 shows the response of the designed low pass filter. At $f \approx 1.0kHz$ the magnitude is around -5 dB. Before that, the magnitude is higher than this level. This means signals with a $f < 1.0kHz$ are almost unfiltered.

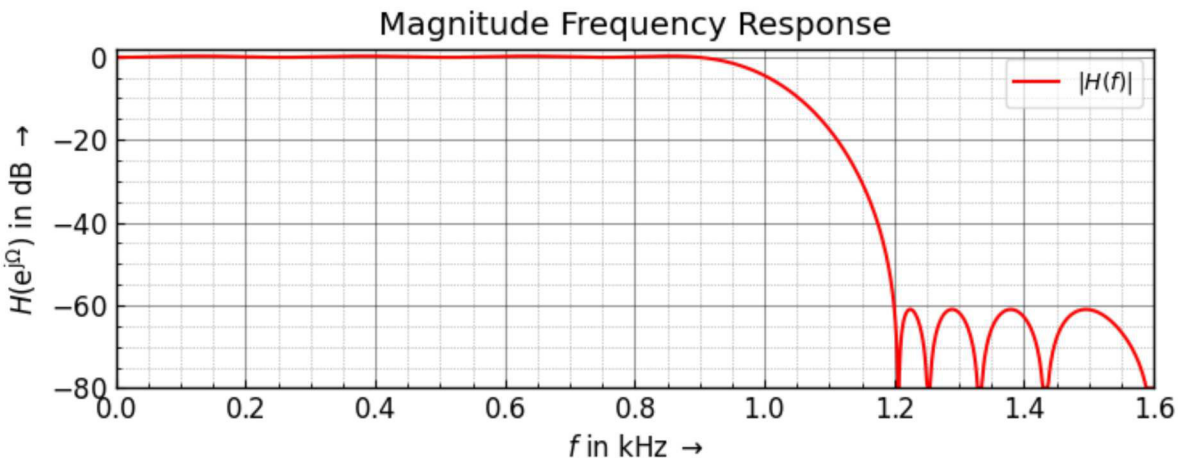

Figure 8.1: Design of Low Pass Filter with pyFDA

The coefficients for this filter are listed in table B.1. These coefficients are used in Vivado to generate the frequency response shown in the figure 8.2.

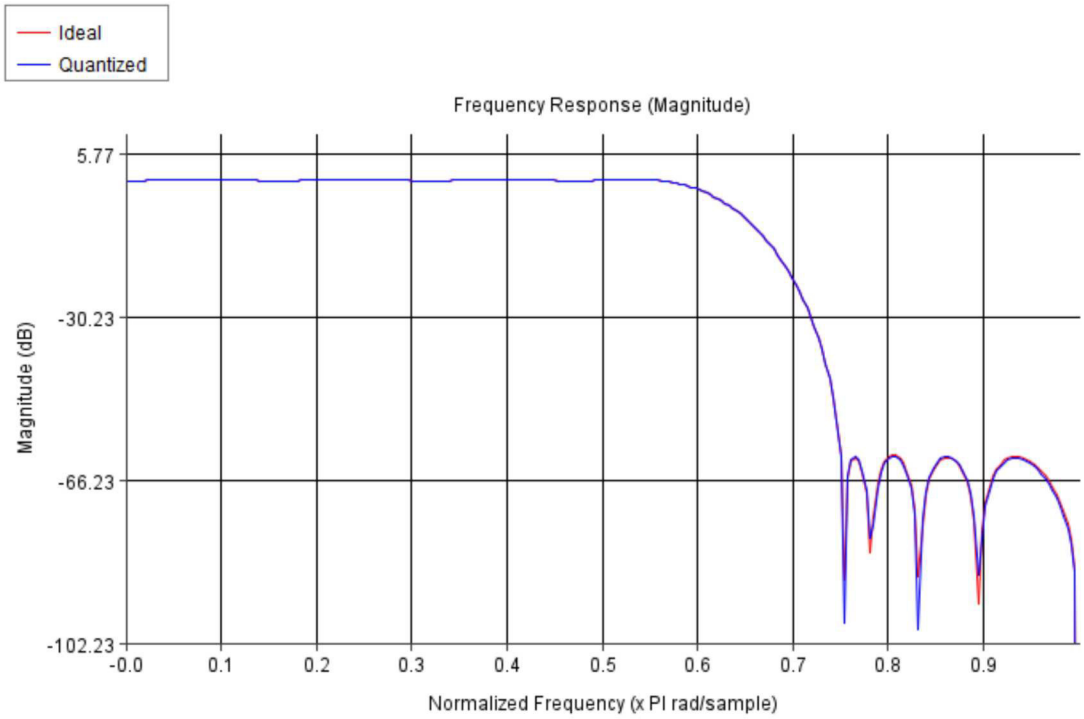

Figure 8.2: Magnitude in "Freq. Response" tab of the Low Pass "FIR Compiler"

The result of comparing the figures 8.1 and 8.2 is that the amplitude responses are almost the same. This means the developed filter should behave like a low pass filter.

8.1.2 High Pass Filter

Figure 8.3 shows the designed high pass filter. At $f \approx 50Hz$ the magnitude is -5 dB. With higher frequencies, the magnitude approaches 0 dB. This means signals with $f > 50Hz$ are almost unfiltered.

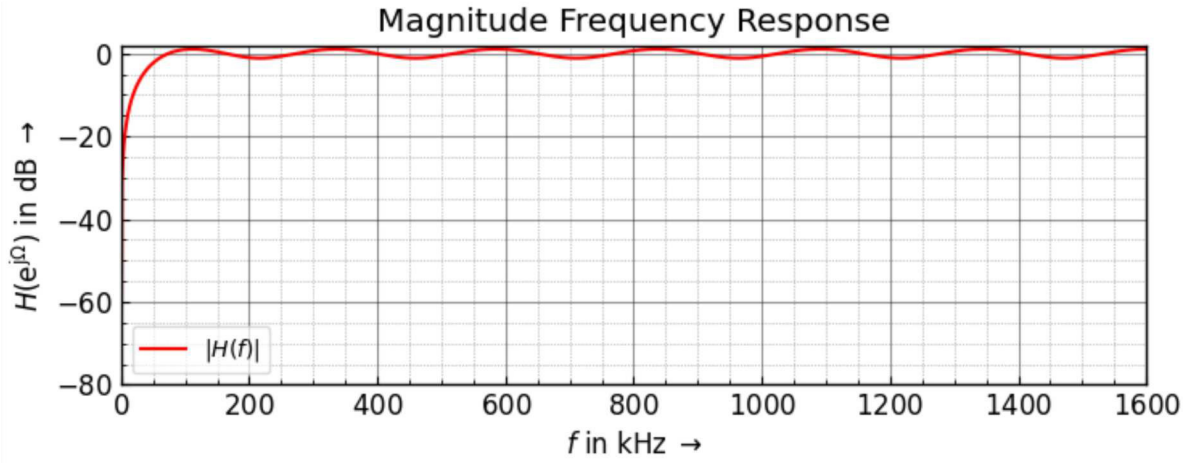

Figure 8.3: Design of High Pass Filter with pyFDA

The coefficients for this filter are listed in table B.1. These coefficients are used in Vivado to generate the frequency response shown in the figure 8.4.

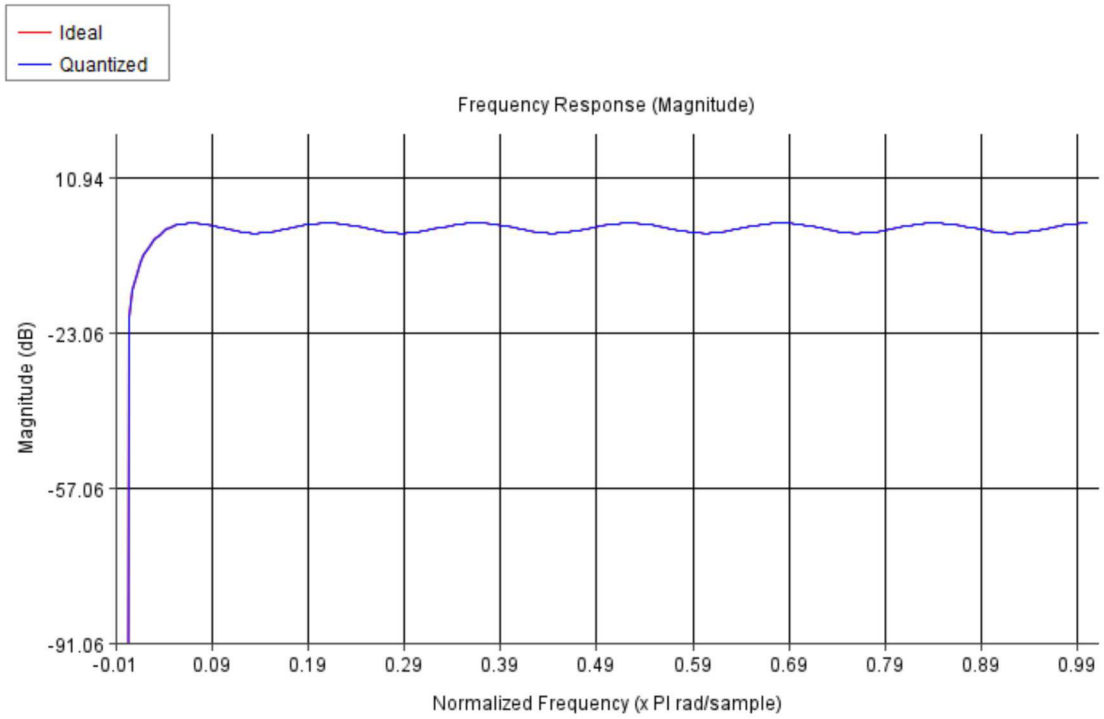

Figure 8.4: Magnitude in "Freq. Response" tab of the High Pass "FIR Compiler"

The result of comparing the figures 8.3 and 8.4 is that the amplitude responses are almost the same. This means the developed filter should behave like a high pass filter.

8.2 Deflection Estimator

The following section shows excerpts of the entire code in section A.4.

The "Deflection_Estimator" gets its data through an AXI4-Stream slave interface. The data is then split according to the defined arrangement. In the calculation phase, the old values are first saved (see listing 2).

```
pre_x_deflection := x_deflection;
pre_y_deflection := y_deflection;
pre_x_velocity   := x_velocity;
pre_y_velocity   := y_velocity;
```

Listing 2: Save previous values

After that, the new acceleration values are compared to the gravitational acceleration. Afterward, the current acceleration is calculated for the axis by subtracting the gravitational acceleration. This value is used to calculate the current velocity for the axis. The current velocity is then used to calculate the current deflection for the axis.

```
if (z0_input_data > G_ACCEL and z1_input_data < G_ACCEL) then
-- left inclination
    x_cur_accel := std_logic_vector(signed(z0_input_data) - signed(G_ACCEL));
    x_velocity  := std_logic_vector(signed(pre_x_velocity) + signed(x_cur_accel));
elsif (z0_input_data < G_ACCEL and z1_input_data > G_ACCEL) then
-- right inclination
    x_cur_accel := std_logic_vector(signed(z1_input_data) - signed(G_ACCEL));
    x_velocity  := std_logic_vector(signed(pre_x_velocity) - signed(x_cur_accel));
else
-- no inclination
    x_cur_accel := (others => '0');
    x_velocity  := std_logic_vector(signed(pre_x_velocity) + signed(x_cur_accel));
end if;

x_deflection <= std_logic_vector(signed(pre_x_deflection) + signed(x_velocity));
```

Listing 3: Calculate values for x axis

The calculation for the y-axis looks similar, which is why it was omitted.

Afterward, the values for the deflection on each axis are passed on to the output. The output is an AXI4-Stream master interface.

8.3 Controller

The "Digital_P_Controller" gets the data via an AXI4-Stream Slave interface and picks one half of this. This is determined by the "upper" port. The following is an excerpt of the complete "Digital_P_Controller" code in section A.5.

The data goes through the calculations shown in listing 4. First, the difference between "reference_variable" and "input_data" is calculated. Then this value is multiplied by the given factor "kp".

```
puffer := signed(reference_variable) - signed(input_data);
calculated_data <= std_logic_vector(signed(kp) * puffer);
```

Listing 4: Calculate the current error signal and the actuating signal

After the calculation is done, the calculated data is passed on to the output. The output is an AXI4-Stream Master interface.

Chapter 9

Implementation of Output Interface

written by Niklas Seeliger

The output interface is made up of components. The configurator which upon getting a reset signal configures the audio chip and the Soundmultiplexer, which receives AXI-Streams and multiplexes them into an I²S Stream that it sends to the audio chip.

9.1 ADAU1761 Operation

We want to use the audio chip to receive I²S data and drive the left and right headphone outputs to the levels of those channels. The headphone outputs of the chip are already hardwired to the "HP+Mic" labeled audio jack. However, the input side needs to be mapped to FPGA signals. We want to drive the headphone output with a sampling frequency of $48\ kHz$ and want to operate the chip with a master clock of 256 times that ($= 12.288\ MHz$).

The audio chip will be configured to be the Inter Integrated Chip Sound (I²S) controller and as such the serial clock and word select lines are inputs on the Field Programmable Gate Array which is programmed to transmit the audio data.

Checking the datasheet [Ana10] we need to interface with the following pins:

- **MCLK** needs to be driven to $12.288\ MHz$. Configured as output on pin U5
- **SDA/COUT** is the SPI MISO line. Configured as input on pin T9
- **ADDR1/CDATA** is the SPI MOSI line. Configured as output on pin M18
- **ADDR0/CLATCH** is the SPI chip select line. Configured as output on pin M17
- **SCL/CCLK** is the SPI clock line. Configured as output on pin U9
- **DAC_SDATA/GPIO0** is the I²S serial clock line. Configured as input on pin R18
- **LRCLK/GPIO3** is the I²S word select line. Configured as input on pin T17
- **BCLK/GPIO2** is the I²S serial data line. Configured as output on pin G18

The ADAU1761 is driven by a clock divider that divides the system clock of $125\ MHz$ by 10 resulting in a master clock of $12.5\ MHz$.

Byte 0	Byte 1	Byte 2	Byte 3
chip_adr[6:0], R/$\overline{W}$	subaddr[15:8]	subaddr[7:0]	data

Table 9.1: Send data to write a single register in ADAU1761

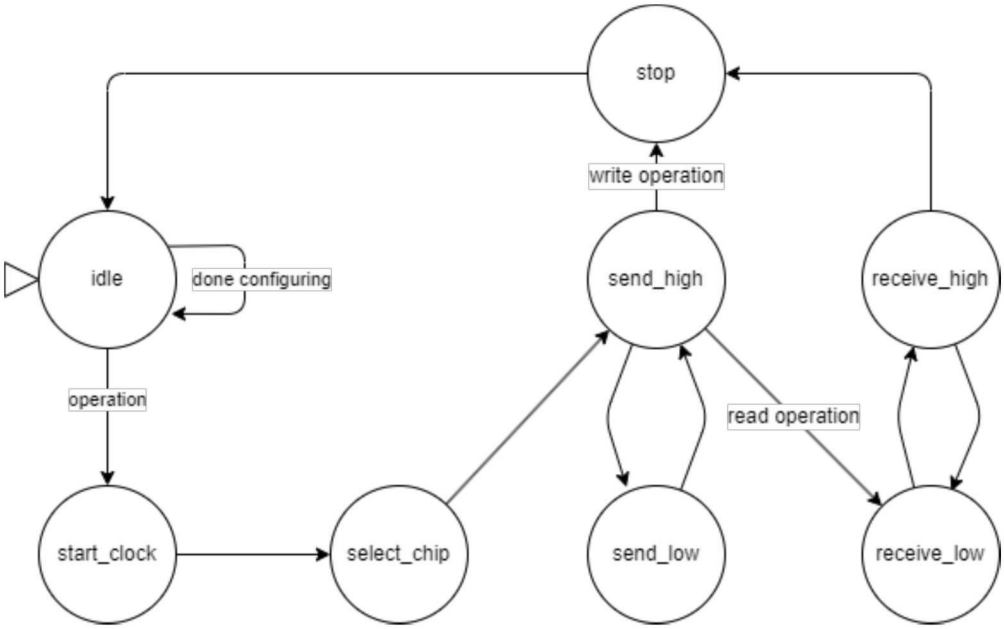

Figure 9.1: State Machine for transmitting and reading data over SPI

9.2 Configurator

Focusing on single byte write operations the audio chip requires the data in Table 9.1 to be sent over the SPI interface to write a register. Where "chip_adr[6:0]" is a series of zeroes, "subaddr[15:0]" is the register address and "data" is the byte that should be written into that register.

Given the desired register contents listed in Appendix C The data below needs to be sent in separate transactions to the audio chip to fully configure it. This list is hardcoded as an array and gets sequentially sent when a reset signal is received. To do this the state machine in Figure 9.1 was implemented. The implementation also allows the reading of data but this is not required to configure the chip.

0x00400001	0x00401f00	0x0040f201
0x00401501	0x004023e7	0x0040f97f
0x00401600	0x004024e7	0x0040fa03
0x00401700	0x00402903	0x0040f400
0x00401c2d	0x00402a03	0x0040f800
0x00401d00	0x00402b00	0x00402daa
0x00401e4d	0x00402c00	0x00402faa

9.3 Soundmultiplexer

The sound multiplexer consists of three processes. Two that implement the very simple AXI Stream slave interface to buffer the last received data on each channel. And one process to output sound data over I²S to the audio chip.

Figure 9.2 shows the process receiving data. It only waits for the rising AXI clock edge and checks if the "tvalid" flag is set. If so, it copies the data into an internal buffer for that channel. The left channel works in much the same way.

```
receiveRData : process(axi_R_clk, reset)
begin
    if reset = '0' then
        current_R_value <= (others => '0');
        axi_R_tready <= '1';
    else
        if rising_edge(axi_R_clk) then

            if axi_R_tvalid = '1' then
                current_R_value <= axi_R_tdata;
            end if;

        end if;
    end if;
end process;
```

Figure 9.2: VHDL process receiving Right Channel Audio data

A little more involved is the process that outputs the audio data to the I²S stream. This is shown in Figure 9.3. Since the audio chip is set up to control the i2s stream this process only needs to react to the bit clock and the left-right clock. It is impossible to specify the bitclock in the sensitivity list of the process because the output pin that goes to the sound chip is incapable of being routed to the clock input of a flipflop. This is a limitation of the selected Field Programmable Gate Array. Because of this the process instead relies on polling the bitclock and checks for rising and falling edges that way. This is the reason the process is sensitive to a "master_clk" that needs to be fast enough to recognize the edges with a low enough latency. In testing the system clock of $125\ MHz$ was sufficient.

By reacting to word select changes on rising edges the process reacts to the new channel on the next falling edge. This results in the 1-bit clock cycle required for I2S communication.

```
updateSerialData : process(master_clk, reset)
begin
    if reset = '0' then
        on_bit <= 0;
        i2s_sd <= '0';
        last_ws <= i2s_lrclk;
        current_output <= (others => '0');
        last_bclk <= i2s_bclk;
    elsif rising_edge(master_clk) then

        if last_bclk /= i2s_bclk then
            last_bclk <= i2s_bclk;

            if i2s_bclk = '0' then -- falling edge on bclk

                -- send bit if available
                if on_bit < DATA_WIDTH then
                    i2s_sd <= current_output(DATA_WIDTH - 1 - on_bit);
                    on_bit <= on_bit + 1;
                else
                    i2s_sd <= '0';
                end if;


            elsif i2s_bclk = '1' then -- rising edge on bclk
                -- buffer word select because bits have to be 1 edge delayed
                -- from word select line

                -- if changed then reset on bit and buffer output data
                if last_ws /= i2s_lrclk then
                    last_ws <= i2s_lrclk;

                    on_bit <= 0;

                    if i2s_lrclk = '0' then
                        current_output <= current_L_value;
                    else
                        current_output <= current_R_value;
                    end if;

                end if;

            end if;
        end if;

    end if;
end process;
```

Figure 9.3: VHDL process outputting the I²S data

By configuring the audio chip to control the I²S communication this process does not need to worry about any timing or bit width constraints as the audio chip fetches the data when it is ready for it and

with a speed that it can handle.

Everything put together gives the block diagram shown in Figure 9.4. Both of the AXI stream slave interfaces are unoccupied and there is a debug in and out on the Configurator that was used during development to read and write the registers via General Purpose Input / Output (GPIO) from Python. The system clock (of $125\ MHz$) is colored blue and the reset line is colored magenta.

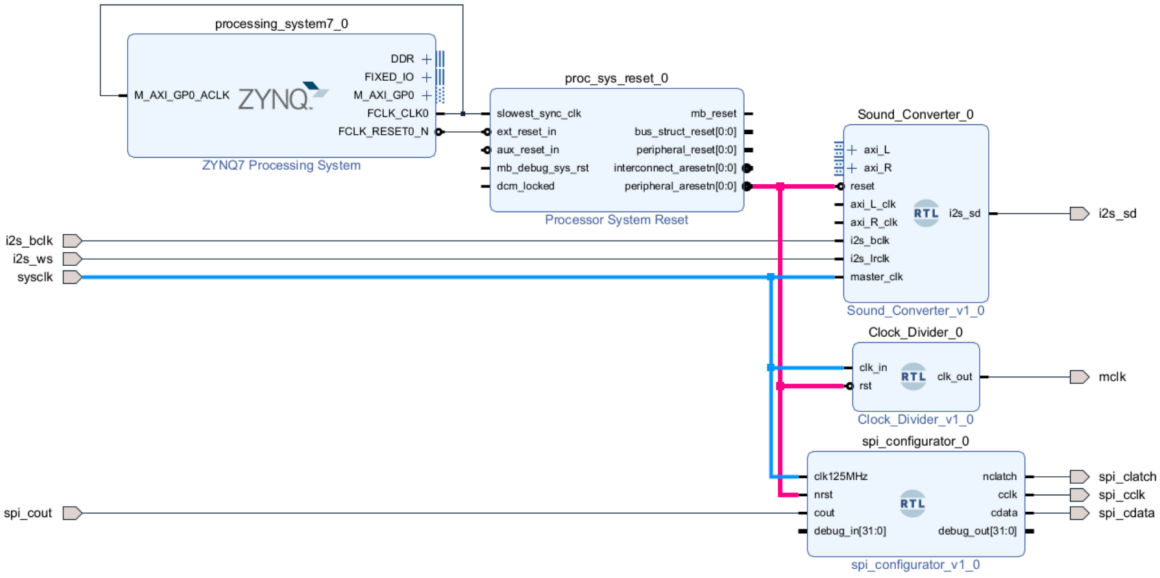

Figure 9.4: Exemplary block diagram showing how the Output interface connects

Chapter 10

Implementation of Configuration

written by Simon Pfennig

The Implementation of the Configuration System is divided into two sections. The Processing System (PS)-Side in which the user will make inputs via Jupyter-Notebook which will be sent to the Programmable Logic (PL)-Side where those inputs will be received by an IP-block to be stored for later use.

10.1 Configuration from the PL-Side

On the PL-side the "AxiGPIO" IP-block (documentation: [XIL12a]) is used. There are two blocks in use, which have the following parameter set:

- Component Name: Either "axi_controller1_kp" or "axi_controller2_kp"
- GPIO -> All outputs: Checked
- GPIO -> GPIO width: 32
- GPIO -> Enable dual channel: Unchecked

In this configuration, the blocks will implement an AXI-Slave which exposes a 32-bit register as output. The blocks can be indexed by their name and the contents of the register can be set as explained in section 10.2. The contents of the registers are read by the Controller.

As shown in Figure 10.1 the AXI-Interface of the blocks is then connected to an AXI-Interconnect which is connected to the AXI-GPIO interface of the PS. This allows the PL to access multiple AxiGPIO and other IP-blocks from the same GPIO connection.

The figure below is a block design that shows how to connect the GPIO-blocks to the Processing System block. When integrating into the final design it is only necessary to use a bigger AXI-Interconnect to accommodate other blocks.

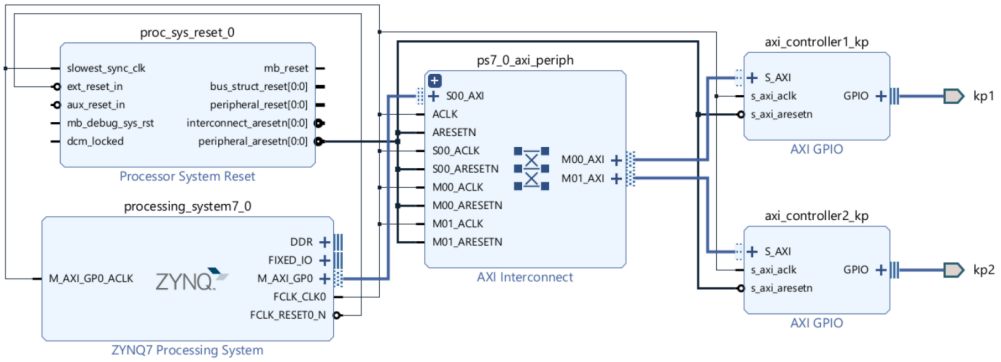

Figure 10.1: Example Integration of AXI-GPIO

10.2 Configuration from the PS-Side

The following code can be used in the Jupyter-Notebook environment to adjust the parameters given to the PL during runtime. See explanations of the code in Table 10.1 below.

```
1  %pip install -q ipywidgets=8.1.0
2  from ipywidgets import interact, fixed
3  import ipywidgets as widgets
4  from pynq import Overlay
5  from pynq.lib import AxiGPIO
6
7  overlay = Overlay("main_wrapper.bit")
8
9  channelP1 = AxiGPIO(overlay.ip_dict['axi_kp1']).channel1
10 channelP2 = AxiGPIO(overlay.ip_dict['axi_kp2']).channel1
11
12 def write_parameter(channel, value):
13     channel.write(value, 0xFFFFFFFF)
14
15 interact(lambda v: write_parameter(channelP1, v), v = widgets.IntSlider(value=0,
16          min=-2147483648, max=2147483647,  step=1 , description='P1'))
17 interact(lambda v: write_parameter(channelP2, v), v = widgets.IntSlider(value=0,
18          min=-2147483648, max=2147483647, step=1 , description='P2'))
```

Listing 5: Python Code to Configure the parameters of the Controller

Lines	Explanation
1-5	Installs and imports the Jupyter Widgets package version 8.1.0 (latest tested version). This allows the creation of interactive elements inside Jupyter Notebook.
7	Loads a bitstream file of a board design and runs it. See chapter 12 for further information.
9-10	Gets the references of the channels associated with the GPIO-IP-block (see section 10.1 for more details). It is indexed by the name given to the IP block in Vivado.
12-13	Defines the callback that will write the configured values using the AXI-protocol via the GPIO-pins between the PS and PL. The bitmask (0xFFFFFFFF) signals, that all 32 Bits of the receiving register will be overwritten by the new value.
15-18	Defines two sliders with a range representing a 32-bit signed integer (see figure 10.2). The callback (lines 10-11) will be called whenever the value of the sliders is changed. The value will thereafter be readable from the corresponding register in the PL.

Table 10.1: Explanations to listing 5

The following figure shows the sliders implemented by the code above. Their values can be changed by dragging the handle, or by clicking on the label displaying the current value and entering a number directly.

Figure 10.2: Sliders used to configure Parameters

Chapter 11

Implementation of Validation Data

written by Benjamin Bissendorf

This chapter describes the implementation of the previously shown concept of the "Validation Data" component. The names used for its internal block are derived from the overview figure 6.10. In the following, a short overview of the implementation is given and in their subsections, the details of the blocks of this component are lined out and discussed.

The "Split and Write" block acts as an AXI4 Stream Slave connected to the "Input Interface" component. From it, the data of the three sensors each with three values for the x-, y-, and z-axis, resulting in $3 \cdot 64Bit = 192Bit$ arrives at the polling rate of 3200 times a second. The Direct Memory Access Controller is configured to write chunks of 64 Bits of data to the RAM, so the incoming data has to be split beforehand by the block. To send this split data, the block again acts as an AXI4 Stream Master for the pipeline into memory.

The "Collect and Log" block runs on the PS and has two primary tasks: To configure the Direct Memory Access Controller and to collect and write the data into a filestream. As the PL does not know where to write the data, an AXI4 interface from the PS to the Direct Memory Access Controller is used and configured with Python to set the destination address and amount of data. The corresponding Python script waits for the buffer in memory to be filled and then orders the Direct Memory Access Controller to write into a second buffer. Between being done with switching and waiting for the second buffer to be filled, the data of the first buffer is written persistently into a file. To avoid data from being lost in the period after the first buffer is full and the Direct Memory Access is ordered to use the second one, an intermediate FIFO block is used, which also buffers the data and provides them in the right order to the Direct Memory Access when it is available again. After the second buffer is filled too, this process starts again, resulting in an endless loop of collecting data, until the user asks to stop.

The Vivado block design of this component can be seen in figure D.2 in the appendix.

11.1 Split and Write

written by Benjamin Bissendorf

The "Split and Write" block needs to act as an AXI 4 Stream Slave to receive the incoming sensor data and as an AXI 4 Stream Master to send the split outgoing data. To follow "Seperation of Concerns", additional blocks are used for this. In figure 11.1 below, the outer interfaces from the block can be seen. The Stream Slave accepts incoming data and provides them internally while signaling their arrival with a dedicated flag. When data is ready to be sent, the Master has to be signaled with a *go* flag and then confirms its transfer with a *done* flag. This allows the main block to concentrate on its primary task of splitting the incoming data:

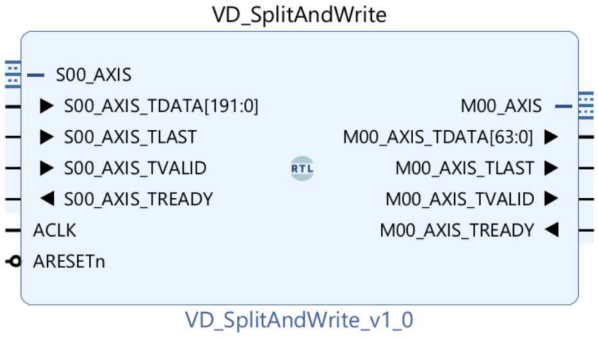

Figure 11.1: The isolated SplitAndWrite block with its interfaces

The internal processing can be described with a finite state machine depicted below in figure 11.2: After a reset, the state machine starts in state "WaitForIncoming". It then waits for new data to arrive, indicated by the signal *data_available_flag*, provided by the "AXIS_To_Vector" subblock. When available, the input vector is saved in internal registers. The machine then transitions to the next state "SendNextChunk". This state determines the range to send from the saved registers and writes these bits into the output buffer. Finally, it updates the internal counter for the sent bits and sets the *go_flag* to 1 before always going into the next state "WaitForDone". With the go flag set, the AXI Master sends the data to the Direct Memory Access Controller. When the transfer between the Master and the Direct Memory Access Controller is complete, the Master signals this with its *done_flag*. As the state machine waits for this, this leads to a transition into the next state. This is either the "SendNextChunk" state if not all bits are sent yet, or the "WaitForIncoming" state.

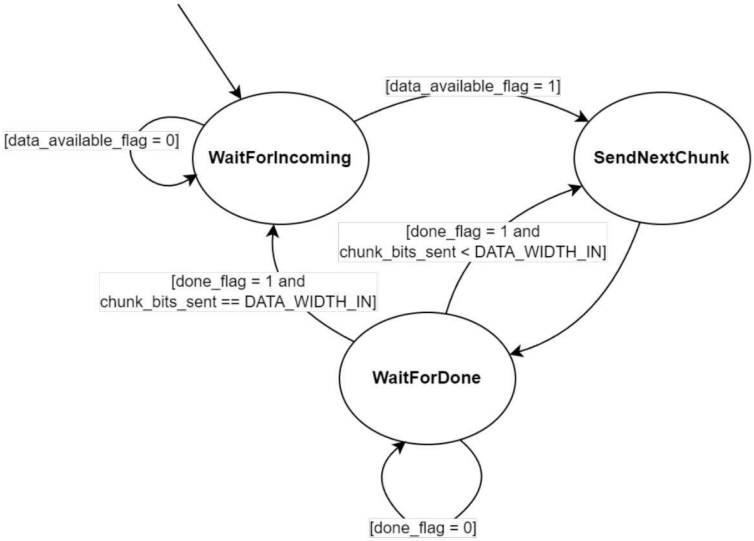

Figure 11.2: The finite state machine of the "WriteAndSplit" block

The VHDL implementation follows the described state machine. For its full code, please see listing 2 in the appendix. There, for the different states, a new type was defined to consist of these states. The current state is saved to the signal "state". When resetting, the initial state "WaitForIncoming" is set. After that, the state machine above was implemented in the VHDL state-case pattern, and state transitions are used when the previously mentioned conditions are met.

When splitting and sending the data, a remark has to be made about the order of the bits: Generally, higher-order bytes will be transferred first. The block is generic and thus configurable but in this implementation with 192 incoming bits and 64 outgoing bits each, the 64 most significant bits are sent first. With the data format described previously in section 7.2, this means the values of *Sensor 1* are sent first, *Sensor 2* second and *Sensor 3* last into memory. When reading from memory, they appear in this

order.

When the Direct Memory Access is configured to write a defined amount of bytes into memory, from the AXI Stream Master it expects the AXI Stream *TLAST* bit to be set for the last word of the buffer. To account for this, the used generic Master is configured to send it after the defined amount of 9600 words, as later discussed in section 11.4.

11.2 FIFO Buffer

written by Simon Pfennig

Between the Sensor Data and Direct Memory Access Controller a FIFO-buffer is used. The "AXI4-Stream Data FIFO" IP-block (documentation: [XIL12b]) as seen in the figure below, is used. On one side, it acts as an AXI-Stream Slave, which receives data and stores them in a FIFO storage. On the other side, it acts as an AXI-Stream-Master that relays the stored information whenever the connected Slave is ready.

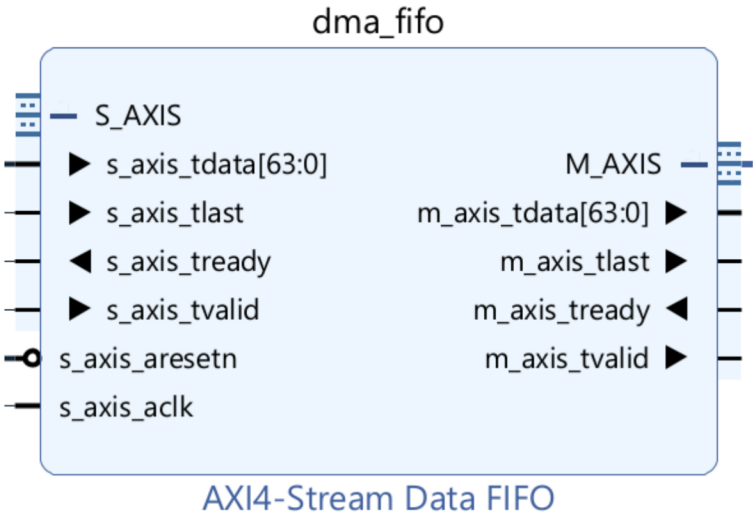

Figure 11.3: The FIFO block

The IP-block was configured to store up to 300 64-bit integers. It also saves the *TLast* signal from the AXI-Stream as it is needed to signal the Direct Memory Access when the buffer is filled.

The FIFO is needed because there is a delay introduced by the PS when switching between the buffers (see section 11.4). While the switch happens it is possible, that the sensors send a new burst of data, while the Direct Memory Access is not able to receive it. If this happens without the FIFO buffer, data would be lost. While testing, this would drop about 3 to 4 sensor bursts per switch. To account for longer delays, for example, when other software is running on the PS, we decided to increase the FIFO size to 300 (a multiple of 3 because each burst will send 3 times 64 bits).

It is still not guaranteed that there won't be any data lost, but since this is only intended for calibration purposes, this risk seems acceptable.

11.3 AXI DMA Controller

written by Benjamin Bissendorf

In contrast to the provisioning of the debugging parameters in section 10.1 where the internal AXI GPIO interface was used, the logging of the sensor data is a more demanding task regarding the requested throughput and independence of the PS. With Direct Memory Access, the RAM can be accessed without the CPU and thus is a faster alternative. It is crucial for the data transfer to happen fast enough before

new data arrives. In its datasheet (see [DMA22, p. 8]) Xilinx writes about their results testing the throughput: They mention sending $10.000B$ at the clock speed of $100MHz$ with a transfer rate of about $298.59\frac{MB}{s}$ which resulted in a total time of

$$t = \frac{10.000B}{298.590.000\frac{B}{s}} = 33,49\mu s$$

While new data, with $3 \cdot 8B = 24B$ arrives every $\frac{1}{3200}s = 0,3125 = 312,5\mu s$, which leaves enough space for deviations. In our experimentation, we were successfully transferring data at this rate without loss over Direct Memory Access.

The used Direct Memory Access Controller is an IP-core made and provided by Xilinx in the used version of Vivado. It can be configured for writing and reading, and for several channel widths, while also offering more features like Scatter/Gather and MicroDMA we don't use. In this project's implementation, the block as seen below in figure 11.4 is configured only to use the writing channel, because the logging mechanism has no use for reading from memory. As it was decided to write the data of each sensor as a chunk of 64 bits, this is also configured as the AXI channel's incoming width. See the full configuration in table 11.1 for more information.

Name	Value
Width of Buffer Length Register	26
Address Width	32
Enable Write Channel	True
Memory Map Data Width	64
Stream Data Width	64
Max Burst Size	16
Allow Unaligned Transfers	False

Table 11.1: Direct Memory Access Controller configuration

The block uses the incoming *S_AXI_LITE* signals for configuration, which is done in Python over the internal AXI GPIO line from the PS. There the destination memory address and the amount of bytes the controller waits for to write are configured. This configuration is described in the following section 11.4. The actual data to write into memory has to be provided at the *S_AXIS_S2MM* signals. To these, the FIFO block mentioned before is connected. To communicate with the RAM, the controller has to be connected to it over the High-Performance interface of the Processing System. This implementation uses the HP Slave "HP0" with a data width of 64 bits.

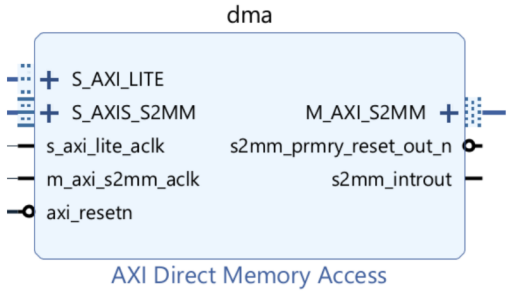

Figure 11.4: The Direct Memory Access Controller block

11.4 Collect and Log

written by Simon Pfennig

The Code below is used in the Jupyter-Notebook environment on the PS. When executed, it will read data from the sensor stream and save it to a file as bytes. See explanations of the code in table 11.2 below.

```
BUFFER_SIZE = 9600 # The number of Elements in each Buffer
ITERATIONS = 5 # How often every Buffer is filled
OUTPUT_FILE_PATH = "test.data" # Which file to append the data to

# Import Numpy and the method to allocate RAM
from pynq import allocate
import numpy as np

# Allocate and initialize 2 buffers
buffer_a = allocate(shape=(BUFFER_SIZE,), dtype=np.uint64)
buffer_a.fill(0)
buffer_b = allocate(shape=(BUFFER_SIZE,), dtype=np.uint64)
buffer_b.fill(0)

recvchannel = overlay.dma.recvchannel
recvchannel.start()

with open(OUTPUT_FILE_PATH, "wb") as data_file:
    # Read one Buffer to empty the FIFO
    recvchannel.transfer(buffer_a, 0, BUFFER_SIZE*8)
    recvchannel.wait()

    # Read into Buffer A
    recvchannel.transfer(buffer_a, 0, BUFFER_SIZE*8)
    recvchannel.wait()

    for _ in range(ITERATIONS):
        # Start Reading into Buffer B while writing the Contents of Buffer A into a File
        recvchannel.transfer(buffer_b, 0, BUFFER_SIZE*8) # Send Command to start filling Buffer
        data_file.write(buffer_a) # Write other Buffer to File
        recvchannel.wait() # Wait for the first buffer to be filled

        # Start Reading into Buffer A while writing the contents of Buffer B into a File
        recvchannel.transfer(buffer_a, 0, BUFFER_SIZE*8)
        data_file.write(buffer_b)
        recvchannel.wait()
```

Listing 6: Python Code to Read the Validation Buffers

Lines	Explanation
1-3	Initializes Constants. BUFFER_SIZE is the number of 64-bit integers in each buffer, it needs to be a multiple of three because there are three sensors with 64 bits of data each. We choose 9600 since this corresponds to one second of data when three sensors send data with a frequency of $3,2kHz$, which is the case for this setup. This time should be sufficient to write the collected data onto a file before the next buffer runs out of space as pointed out in section 11.3. ITERATIONS is the number of times each buffer is read and saved to a file. Amounting to a total of $\frac{BUFFERSIZE}{3} * ITERATIONS * 2$ sensor bursts being written to the file at OUTPUT_FILE_PATH.
10-13	Allocates 2 buffers in RAM. They are returned as extensions of Numpy arrays and can be used as such. They are extended with the physical address of the array in RAM, which is used in the transfer()-Method to give to the Direct Memory Access.
15-16	Gets the object used to communicate with the Direct Memory Access via AXI and starts the Direct Memory Access's output channel.
20-21	Reads one buffer worth of data. Because the FIFO buffer on the board will fill up even when the Direct Memory Access is not reading the data, the data in the FIFO will not match up with the data currently supplied by the sensors. Since the FIFO is smaller than the buffer these lines will remove this mismatch by rejecting the first buffer.
24-25	Writes the command to the Direct Memory Access to start filling the first buffer and waits for it to be full. It uses the physical RAM address of the buffer, provided by the buffer_a object with an offset of zero (start writing at the start of the buffer), write BUFFER_SIZE times 8 bytes (since we save 8-byte integers)
29-31	Since Buffer A is full at this point, the Direct Memory Access is commanded to write data to Buffer B. The slack between the moment when Buffer A is full and the moment when Buffer B can be filled is handled by the FIFO. While Buffer B is filled, the contents of Buffer A will be saved to a file. After that, we wait for Buffer B to be full.
34-36	The same as lines 29-31 but the roles of the buffers are reversed. This creates a loop in which one buffer is always filled with data while the other one is saved to a file.

Table 11.2: Explanations to Listing 6

The data is saved directly as a stream of bytes. This is to ensure the PS has enough time to save the array before the next buffer is filled. To make evaluation easier it is preferable to have the data available in CSV format. Therefore the code below can be used to convert the byte file into a CSV file. See explanations of the code in Table 11.3 below.

```
1  CSV_OUTPUT_PATH = "test.csv"
2
3  with open(OUTPUT_FILE_PATH, "rb") as data_file:
4      with open(CSV_OUTPUT_PATH, "w") as csv_file:
5          csv_file.write("X1\tY1\tZ1\tX2\tY2\tZ2\tX3\tY3\tZ3\n")
6          while True:
7              b = data_file.read(24) # Read 3 Longs of Data
8              if not b: # EOF
9                  break
10             line = "{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\n"
11             line = line.format(
12                 int.from_bytes(b[6:8], byteorder="little", signed=True),    #X1
13                 int.from_bytes(b[4:6], byteorder="little", signed=True),    #Y1
14                 int.from_bytes(b[2:4], byteorder="little", signed=True),    #Z1
15
16
17
```

```
18                  int.from_bytes(b[14:16], byteorder="little", signed=True), #X2
19                  int.from_bytes(b[12:14], byteorder="little", signed=True), #Y2
20                  int.from_bytes(b[10:12], byteorder="little", signed=True), #Z2
21
22                  int.from_bytes(b[22:24], byteorder="little", signed=True), #X3
23                  int.from_bytes(b[20:22], byteorder="little", signed=True), #Y3
24                  int.from_bytes(b[18:20], byteorder="little", signed=True), #Z3
25
26              )
27              csv_file.write(line)
```

Listing 7: Python Code to Convert the Buffer Data to CSV

Lines	Explanation
3-4	Opens the file in which the sensor data is stored, as well as a new file in which the data converted to CSV will be stored.
5	Writes the header of the CSV file.
7-9	Reads 24 bytes of data (one sensor reading) from the data file.
10-22	Converts the data into 9 16-bit integers, one for each axis of each sensor, formats them into CSV, and writes it to the CSV file.

Table 11.3: Explanations to figure 7

11.5 Test

written by Simon Pfennig

The figures below show a visualization of the output from the CSV file created by the code in listing 7. The input data was the Sawtooth-wave created by the Sensor Dummy (see. 7.4).

The first figure shows that the Sawtooth-wave is transmitted as expected. With three different waves for the three different sensors each spanning from -500 to 500 and with a phase shift of 100 between each of the waves, showing that the Validation Component outputs the expected values.

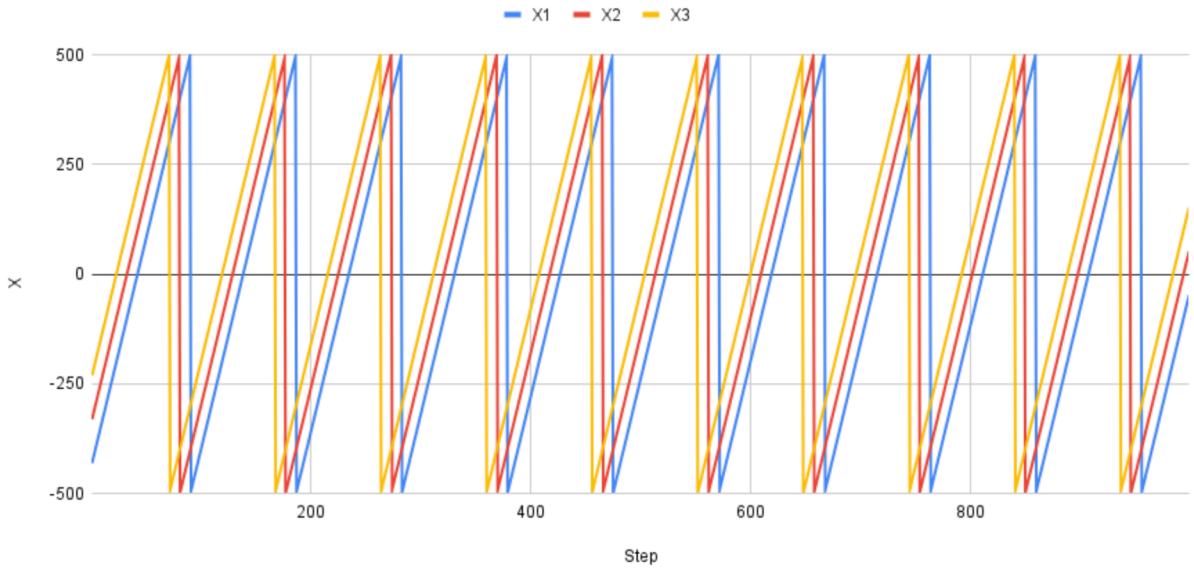

Figure 11.5: Validation Data Phase Shift

The second figure shows the moment the Direct Memory Access switched from Buffer A to Buffer B (at Step 3200). If a sensor burst were to be lost there would be a larger gap between the values of 3200 and 3201 as between the other values. Since that is not the case we can conclude, that the FIFO buffer was large enough to account for the delay from switching.

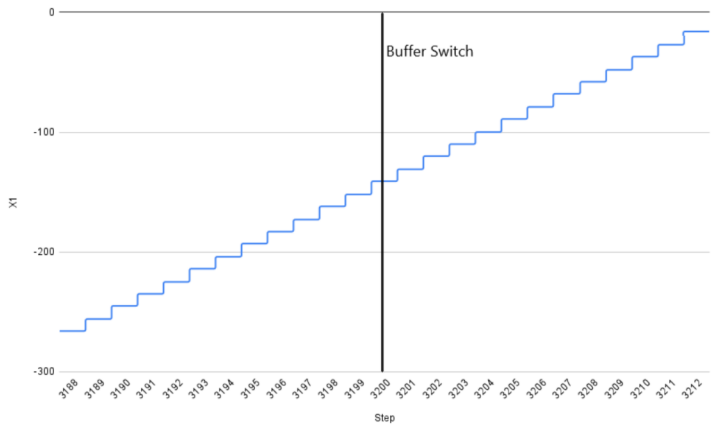

Figure 11.6: Validation Data Buffer Switch

Chapter 12

Integration

written by Benjamin Bissendorf and Simon Pfennig

12.1 System level integration

written by Benjamin Bissendorf

To integrate all the implemented digital components into the system structure described in chapter 6 a Vivado Block Diagram is used. As illustrated below in figure 12.1 the blocks of the individual components were inserted and connected as described in their respective chapters. Now following is the connection of the blocks on the boundary of their component resulting in the full system. For better visibility, the sensor data path is highlighted.

The internal origin of the sensor data is the "Sensor Communication" block of the "Input Interface". Unfortunately, this block was not finished in time. To compensate, we implemented a "Sensor Dummy" (orange) which provides test data. This is polled, collected, and then provided for the "Digital Controller" and "Validation Data" interface via an intermediate AXI 4 Stream Broadcaster.

In the top right of the figure is the "Digital Controller" component. After its deflection estimation, the sensor positions are given to the P-Controller (blue path). Their output is then given to the "Sound Converter" of the "Output interface". Its I^2S signals are broken out to the board for the board-internal audio processor to process and output.

At the bottom of the figure, the "SplitAndWrite" block of the "Validation Data" receives the bright green sensor data. Internally this is split and then transferred into memory (indicated by the dark green path).

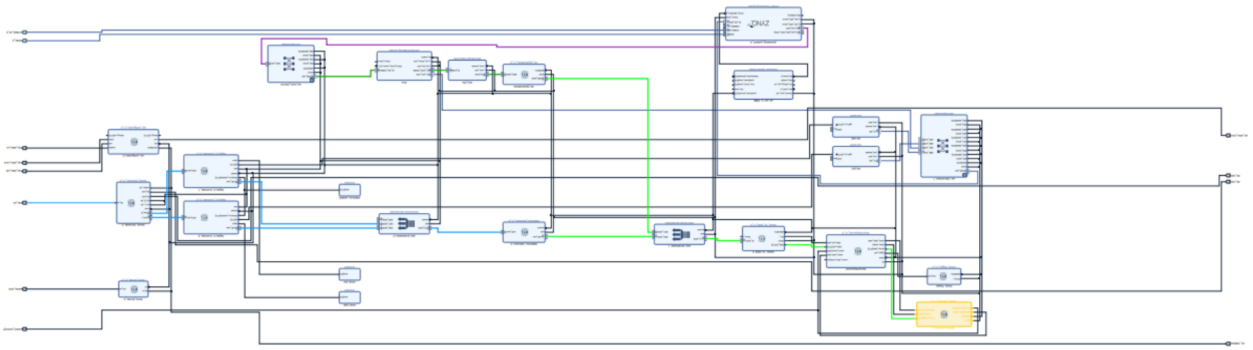

Figure 12.1: Full integration Vivado block diagram

12.2 Start of operation

written by Simon Pfennig

To prepare the pynq-z2 board for operation, an SD card with the correct board image is needed. The latest image file is available on the manufacturer's website (see [Tul20]). The .img file then must be written onto a bootable SD card. After inserting the card into the board and powering it up with the included power adapter, the board signals its availability with blue flashing LEDs.

When connected to a Network via Ethernet, the board's Jupyter-Notebook environment is exposed `http://pynq:9090/` by default. To log in you need the credentials, which by default are:

Username: xilinx

Password: xilinx

The next step is to move the Bitstream (.bit) and Hardware Handoff (.hwh) files onto the board. For this, we used the following script to copy the files from the Vivado project onto the board. The ssh credentials are the same as above. The .bit file and .hwh files need to have the same name and be in the same directory on the board, otherwise, the paths can be changed.

```
set pynq_host=[IP of Pynq-Board]
scp VCS_FPGA.gen\sources_1\bd\main\hw_handoff\main.hwh xilinx@%pynq_host%
    :~/jupyter_notebooks/embeds/py_dma_test/bitstream/main_wrapper.hwh
scp VCS_FPGA.runs\impl_1\main_wrapper.bit xilinx@%pynq_host%
    :~/jupyter_notebooks/embeds/py_dma_test/bitstream/
```

Figure 12.2: Upload Script for Bitstream and Hardware Handoff

After loading the files onto the board you can use the Jupyter-Notebook environment to load the bitstream and the hardware handoff-file. The following code snippet can be used for this purpose.

```
from pynq import Overlay
overlay = Overlay("main_wrapper.bit")
```

Figure 12.3: Load Bitstream onto board

Chapter 13

Evaluation

written by Benjamin Bissendorf and Niklas Seeliger

The integration result is synthesized and implemented in Vivado. Its bitstream was generated and then loaded onto the board as described in the previous section. Unfortunately, we were unable to sense an immediate response to the audio output of the mechanical model. Due to time constraints, we are not able to fully locate the source of the error. We know there is an issue with the connection from the board to the speakers from earlier tests. But debug output of the controller indicated another problem earlier in the chain.

On the other hand, the collected data, provided on the central data stream, was successfully written into a file by the "Validation Data" component. We were able to process the raw values, as explained in section 11.4, into a readable CSV file, which proved the success of this component.

Reflecting on the project as a whole we have definitely succeeded in demonstrating the capability of the board to control vibrations. The specified design is capable of reading sensor data as fast as is required to control the vibrations. The data can be written to RAM to be analyzed. Control Parameters can be set for the controller. The controller is capable of calculating corrections and the output interface is capable of driving the actors to the specified state. From a digital perspective actively controlling vibrations is feasible. Due to a lack of time we were however unable to read acceleration data from the sensors. Moreover, we could not drive the actors of the first testbed. Since we could drive normal wired headphones, we conclude this is due to a wiring issue in the testbed.

Chapter 14

Summary

written by Jan Bredereke

We investigated how an active control can stabilize a satellite's on-board camera in an inexpensive way, disturbed by micro-vibrations coming from the reaction wheels of the Attitude Determination and Control System (ADCS), and maybe also disturbed by its orbital movement. This research question turned out to have two major aspects, the digital part of the control and the mechanical part.

14.1 Digital Part of the Control

From several ways of approaching the digital part, we decided for an active control running in the FPGA of a PYNQ-Z2 board. Furthermore, we chose the control to be a PID controller, adjusted experimentally. See Chap. 2.

It turned out that obviously it will be feasible to implement the digital part in this way. We did not have the time and human resources for a complete implementation of the digital part. But our design is capable of reading sensor data as fast as is required to control the vibrations. (See Sect. 6.2 and Chap. 7.) Our implementation took data produced at this rate in the FPGA and wrote it to a file. We then visualized it offline for further analysis. (See Sect. 6.5 and Chap. 11.) We implemented setting the parameters used by the control algorithm in the FPGA; we set them from the Jupyter notebook graphical user interface. (See Sect. 6.1 and Chap. 10.) We implemented a simple control algorithm in the FPGA. (See Sect. 6.3 and Chap. 8.) We implemented outputting the control commands from the FPGA to the analog output of the PYNQ-Z2 board. (See Sect. 6.4 and Chap. 9.) Figure 6.1 on page 17 presents an overview of the general structure of the digital control system. All of the blocks have either been implemented or at least shown to be implementable.

Due to lack of time and human resources, we could not complete implementing several submodules of the digital part. We report on this future work in Sect. 15.1 below.

14.2 Mechanical Part

The mechanical part of a test bed was provided to us by the VIBES project; we didn't intend to work on it. Nevertheless, we conceived several improvements, and we implemented a few of them. Starting point was the insight that the simplification to translational vibrations would prevent any digital control to work in the test bed provided. Measuring and controlling rotational vibrations is necessary. Since rotational vibrations, not translational vibrations, are the only vibrations relevant in orbit, we decided to improve the design of the test bed with this respect. This allowed us, additionally, to make the task for the digital part much more similar to the real task in orbit. Other aspects of the first version of the test bed required improvement, too, in order to not prevent a successful demonstration of the digital part. (For details see Chapter 3.)

Improving the mechanical aspects of the test bed wasn't handed to the students in the project, since their teaching course should concentrate on the digital part. Furthermore, human resources and time were scarce anyway. Instead, the conceptual and practical work on the test bed was done by Prof. Bredereke in close collaboration with the lab engineer Mr. Sembritzki. We built a new printed circuit board with an arrangement of sensors capable to handle rotational vibrations. We also improved the wiring to this board. Designing these improvements early in the project allowed us to keep the electrical interface stable for the rest of the project; this helped to design and implement the digital part of the control. (For details see Chapter 4.)

We achieved the insight that integrating our new sensor board into a robust actor mount will require substantial more work on the actors and on their mount; and it will definitely need several more iterations of the test bed. In Chapter 15.2 below, we discuss what will be required, and we will present ideas on what can be tried.

For preliminary testing of the digital part, finally we used the new sensor mount together with the old actor mount. This reduced the test setup to a linear chain from sensors to actors, without the mechanical loop back from the actors to the sensors.

Chapter 15

Future Work

written by Jan Bredereke

Some work on the digital part of the control and substantial work on the mechanical part remains for future work. Prof. Bredereke considers continuing the work on the digital part in another course in the next winter term, depending on progress with the mechanical part. The open mechanical challenges probably should be addressed by mechanical engineers of the VIBES project; nevertheless Prof. Bredereke and the lab engineer Mr. Semritzki are interested in discussing solutions and maybe in working on selected aspects. When all parts have been implemented, an experimental proof has to be conducted that the control indeed reduces vibrations as intended.

Access to the Git repository with the source code may be requested from Prof. Bredereke.

15.1 Digital Part of the Control

The digital part of the control still requires the following work:

1. **Most notably, the input interface needs to be completed.** The major challenge here is to understand and handle the complexities of an AXI-Lite master.

   AXI is a standard interface between IP cores, in particular to pre-fabricated IP cores[1]. (On AXI, see Sect. 5.1.) Nearly all of our current interfaces are AXI-Stream interfaces. AXI-Stream is the simplest form of an AXI interface. The students managed to understand and handle this rather rapidly.

   However, the pre-fabricated IP core for accessing the SPI bus to the sensors must be accessed itself through an AXI-Lite interface. The accessing VHDL code therefore must implement the master side of the AXI-Lite connection. The master side is more involved that the slave side.

   In many applications, developers do not need to write an AXI-Lite master. Usually, microcontroller IP cores and DMA IP cores take this role, only. As long as the software on the microcontroller of the System-on-Chip (SoC) orchestrates the behaviour of the digital components on the FPGA, there is no need to write an AXI-Lite master.

   Our design is special with respect to which component orchestrates the others. We don't want to hand this role to the microcontroller, since this would require to use a real-time operating system. This would conflict with other requirements. For details see Chap. 2. Therefore, we need to write at least one component capable of the AXI-Lite master role.

   A preliminary test already showed that we can access the sensors via the SPI IP core, if we use the microcontroller and a Python script as AXI-Lite master. See Chap. 7.

[1] An IP core is a ready-for-use digital circuit design usually providing often-used functionality, serving as building block for rapid digital system design.

As soon as our VHDL code can poll the sensors on the SPI bus via the AXI-Lite interface, we can replace the current dummy data source with the real sensor data source.

2. **The digital controller requires some reworking.**

   (a) **The low pass filter and the high pass filter in front of the actual control component are still missing.**

   It is clear that the pre-fabricated "FIR Compiler" can generate a suitable filter from a set of filter parameters. Furthermore, the generated IP core uses AXI-Stream interfaces, which just need to be chained together with little effort.

   A certain challenge comes from the fact that using integer values should be preferred over floating point values. This is supported by the FIR Compiler. But the details still need to be worked out.

   The work in Chap. 8 employs the tool "pyFDA" to calculate parameters for the filters. This work should be revisited in order to document the design path from the high-level requirements on the filters down to the actual filter parameters.

   (b) **The "deflection estimator" requires revisiting.**

   Its task is to take the acceleration values from the sensors and to compute the current location of the printed circuit board (PCB) with the sensors (and, ultimately, with the camera). The PCB can be tilted sideways on two axes. Compare Sect. 4.2.

   The concept for this, documented in Chap. 6.3, requires a revisit of the physical laws used. Currently, the formulae used implicitely assume a uniform acceleration. However, the accelerations by the vibrations are obviously ever-changing. Therefore, a double integration is required to calculate the current location from a history of accelerations. The integrations need to be done using time-discrete samples. This discretization renders approximations of the current location. Such an integration can be done by summing up over time. The current VHDL code already does some summing up. However, it is not yet clear how this connects to the physics of the test bed.

   (c) Only if actually needed: **the current controller could be extended to a full PID controller.**

   Currently, our controller is a simpler P controller, without I and D. It is likely that this approach is sufficient to solve our control task.

   In case that experimentation shows that the more advanced I and D parameters are actually required, the controller needs to be improved. Currently, the P controller boils down to a simple multiplication with the k_P parameter.

   Two solutions paths suggest themselves. The first is to implement a more complex controller in VHDL manually. This should be feasible since the algorithm is of rather limited complexity. The other path is to look for an IP core providing a PID controller. A preliminary search of us rendered an IP core with non-matching interfaces only (Wishbone instead of AXI).

3. **It should be validated that the bandwith of all data path segments of the control loop indeed is appropriate.**

   The sensors shall be sampled at 3200 Hz. The following transmission via an AXI-Stream interface is capable of higher throughput. The I^2S stream on the output side is slower than the AXI-Stream but faster than the sensors are read. Should, in the future, data be read faster than than 48 kHz, some values would be discarded by the sound multiplexer driving the I^2S connection. See Sect. 6.4.

   This architecture should be validated in a systematic way with respect to potential for losing data and minimizing latency.

4. **The initialization of the audio chip might be made more self-contained.**

   The audio chip on the PYNQ-Z2 board (outside of the FPGA and the SoC) needs a substantial set of initialization data written into its configuration registers. (See Chap. 9.) Currently, this initialization is done by a Python script running on the microcontroller. If no dependency on

software is desired, even for a one-time initialization, then this initialization should be done by VHDL code executed on the FPGA, too.

5. **An integration test** including *all* digital components, in particular the extended an the reworked ones, must be done.

6. **Experiments with the complete test bed need to be performed in order to find suitable parameters for the two PID controllers.** This experimentation will be intertwined with the experimental proof of vibration reduction, as discussed in Sect. 15.3 below. Experience from other projects has shown that finding suitable parameters may need a bit of time.

When all of the above have been achieved, the following bonus value might be worked on:

7. **Use the existing mechanical hardware, sensors, and digital circuitry for measuring the vibrations around the line-of-sight axis of the camera.**

   We have no actor to counter this rotational vibration. But we can measure its angular amplitude, and we can calculate whether this angular amplitude causes the pixels in the camera to shift less than one pixel wide. See Sect. 4.6.

   Our digital part already writes all relevant data to a file. Therefore, this task is just an offline evaluation of the data in the file.

15.2 Mechanical Part

The mechanical part, including the analog electronics, requires the following work:

1. **The actors need substantial work.**

   The current freely hovering mount for the camera and the sensor is fragile, definitely not suitable for a rocket launch, and likely has substantial coupling between the two axes. See Sect. 3.

   Our idea of placing the mount on a flexible, central rod and pushing the mount from below instead of sideways should improve this. See Sect. 4.2. The details still need to be worked out and must be implemented. We expect this work to require several iterations of the test bed; in particular, if resonances shall not interfere.

   Using speakers as actors, in the way proposed in Sect. 4.2, can be an intermediate step, only. They are still too fragile for a rocket launch. A design using dedicated magnet-coil pairs will be much more robust. Also, it will demand much less precious space onboard. See Sect. 4.2, again. However, such a design likely will require several iterations of the test bed until being mature.

   Using piezo actors potentially can improve the situation further, due to their even more increased robustness and even smaller size. But before embarking on this solution, one should have assessed whether a piezo actor or a stack of piezo actors can achieve an amplitude sufficient for our purpose. Furthermore, the high voltage required might be an issue. Compare Sect. 4.2.

2. Building on the above actor design, **the PCB with the sensors must be integrated with a camera, mounted on the flexible rod, and mechanically be connected to the actors.**

3. As soon as the camera-sensor mount is in place, **the wiring of the camera-sensor mount must be completed.**

   Our idea of using thin, short, single, unshielded wires can be used here. We have already implemented this idea, but only for connecting the sensor mount to a temporary, fixed stripboard below it. Compare Sect. 4.3.

   An even distribution of the wires to all sides of the PCB might improve the mechanical properties of the connections further. We will need about eleven signal wires (4 SPI, 3 SPI chip select, 4 USB). Therefore, we should attach about three wires to each of the four sides of the PCB.

4. **The issues in the analog electronics of the test bed need to be analyzed and resolved.** See Chap. 13.

5. **A camera must be added.**

   Starting from an iteration of the test bed where the mechanical loop is closed, we need to add a dummy with the approximate size and weight of a camera.

   As soon as the evaluation of image blurring is expected, a camera with a narrow angle of view must be bought and added.

6. As soon as the mechanical design becomes mature and a basic control of mid-strength vibrations has been achieved, **the low-end-price sensors should be replaced by somewhat less noisy and thus more sensitive sensors.**

   Using low-end-price sensors saves substantial cost in the phase of experimentation with numerous iterations of the test bed. See Sect. 4.1. The project VIBES already did prove that micro-vibrations can be measured with affordable sensors. But it is preferable to perform the final experimentations with less noisy sensors.

7. **Softer feet for the test bed are required.**

   The current off-the-shelf feet are way too stiff. We already experimented with 3D-printed feet from a soft material. They had the form of a ball, attached at the top and at the bottom. However, this was still too stiff.

   Another idea is to reduce the ball to a thin column. This will allow for easier movements to the sides. We expect the sideways movements of the bottom metal plate to be dominant when the shaker motor induces rotational vibrations into the two-storey construction with the lower and the upper metal plate. Compare Fig. 3.1 and Fig. 3.2 on page 5.

   Yet another idea is to use rubber balls, but without attachments at the top or at the bottom. They could rest in shallow bowls, instead. This renders maximal freedom for sideways movements. The price is a more difficult handling when setting up and stowing away the test bed.

8. **Resonances and other mechanical deficiencies of the test bed need to be explored and resolved.**

   As proposed in Chap. 4, we should start out reducing vibrations at a single frequency close to 1 kHz, avoiding resonance frequencies as far as possible.

   After this succeded, we need to explore any resonances and other mechanical deficiencies, in order to reduce and mitigate them. This will require further iterations of the test bed. This latter step will be intertwined with the experimental proof of vibration reduction, as discussed in Sect. 15.3 below.

15.3 Experimental Proof of Vibration Reduction

When all parts have been implemented, an experimental proof has to be conducted that the control indeed reduces vibrations as intended. As discussed in Sect. 15.1 above, this must be intertwined with finding suitable parameters for the two PID controllers. As discussed in Sect. 15.2 above, this must be iterated for exploring and resolving resonances and other mechanical deficiencies of the test bed. The goal is to show the reduction over the entire frequency band applicable.

When the validation data taken from the sensors in the test bed indicate that the vibrations have been reduced to a satisfactory level, a cross check should be done. We should measure how much the actual camera images are blurred without and with vibration reduction. The internship report [Ger23] provides substantial advice on how such measurements can be performed.

Additionally, external measuring equipment for measuring the vibrations directly might be used. For example, a laser interferometer would provide extremely sensitive measurements. The price tag of such equipment might be prohibitive, however.

Bibliography

[Adv22] Inc. Advanced Micro Devices. *Pynq Libraries*. https://pynq.readthedocs.io/en/v3.0.0/pynq_libraries.html. Accessed: 2024-02-13. 2022.

[ADX22] *Datasheet ADXL312*. Rev. C. Analog Devices, Inc. 2022. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/adxl312.pdf.

[AMD21] *AMBA® AXI-Stream Protocol Specification*. Advanced Micro Devices. 2021. URL: https://developer.arm.com/documentation/ihi0051/latest/.

[AMD22a] *AXI Interconnect v2.1 LogiCORE IP Product Guide*. Advanced Micro Devices. 2022. URL: https://docs.xilinx.com/r/en-US/pg059-axi-interconnect/AXI-Interconnect-v2.1-LogiCORE-IP-Product-Guide.

[AMD22b] *FIR Compiler v7.2 - LogiCORE IP Product Guide*. AMD Xilinx. 2022. URL: https://www.xilinx.com/support/documents/ip_documentation/fir_compiler/v7_2/pg149-fir-compiler.pdf (visited on 02/12/2024).

[AMD23] *AMBA® AXI Protocol Specification*. Advanced Micro Devices. 2023. URL: https://developer.arm.com/documentation/ihi0022/latest.

[Ana10] Analog Devices. *Data Sheet ADAU1761*. datasheet. Accessed: 2024-2-15. Sept. 2010.

[CHI24] chipmuenk. *Python Filter Design Analysis Tool*. 2024. URL: https://github.com/chipmuenk/pyfda (visited on 02/14/2024).

[DMA22] *AXI DMA v7.1*. Advanced Micro Devices, Inc. Apr. 2022. URL: https://docs.xilinx.com/viewer/book-attachment/ePquvyIHSl7mKfi0ecEn4Q/CPPzqKuxCU1Q4a3rXOA1jw.

[ECSS13] *Spacecraft Mechanical Loads Analysis Handbook*. Standard ECSS-E-HB-32-26A. European Cooperation for Space Standardization. Noordwijk, The Netherlands, Feb. 19, 2013.

[ESSB11] *ESA pointing error engineering handbook*. Standard ESSB-HB-E-003. European Space Agency. July 19, 2011.

[EVN24] S. Esakkirajan, T. Veerakumar, and Badri N. Subudhi. "FIR Filter Design". In: *Digital Signal Processing: Illustration Using Python*. Singapore: Springer Nature Singapore, 2024, pp. 263–302. ISBN: 978-981-99-6752-0. DOI: 10.1007/978-981-99-6752-0_7. URL: https://doi.org/10.1007/978-981-99-6752-0_7.

[Ger23] Tim Gersting. *Development of an optical payload for VIBES MVTB*. Internship report. City Univ. of Applied Sciences Bremen, Germany, Mar. 30, 2023.

[mat21] matteocastagnaro. *spi_test_pmodB*. MakarenaLabs. July 2021. URL: https://github.com/MakarenaLabs/Common-PL-Devices-on-PYNQ/blob/main/SPI/spi_test_pmodB.ipynb (visited on 02/15/2024).

[Men13] Dietmar Mende. *Physik - Gleichungen und Tabellen*. 16., aktualisierte Aufl. Hanser eLibrary. 2013. ISBN: 9783446438613. URL: https://dx.doi.org/10.3139/9783446438613.

[NXP22] NXP. *I2s Bus Specification*. Accessed: 2024-2-15. Feb. 2022.

[Phi22] H.-W. Philippsen. *Einstieg in die Regelungstechnik mit Python*. 4., aktualisierte Aufl. Hanser, 2022. ISBN: 9783446474338. URL: https://doi.org/10.3139/9783446474338.

[SVP19] Geert Smet, Jeroen Vandersteen, and Massimo Palomba. "The Consequences of Your Microvibration Requirement on Mechanisms Design and Verification – Some Dos and Don'ts". In: *Guidance, Navigation, and Control 2019. Proc. of the 42nd AAS Rocky Mountain Section Guidance and Control Conference* (Breckenridge, Colorado, USA, Jan. 31–Feb. 6, 2019). Ed. by Heidi E. Hallowell. Vol. 169. Advances in the Astronautical Sciences. Univelt, Inc., San Diego, California, USA, 2019, pp. 135–136. URL: http://www.univelt.com/book=7544 (visited on 12/22/2023).

[Tul20] TUL Corporation. *TUL PYNQ-Z2 board*. https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html. Accessed: 2024-02-16. 2020.

[Wit23] Philipp Wittje. *Pinlayout Testbench*. Dokumentation zur Verbindung der Komponenten mit dem PYNQ Z2-Board. German. VIBES – City Univ. of Applied Sciences Bremen, Germany. Dec. 6, 2023.

[XIL12a] *LogiCORE IP AXI GPIO (v1.01.b)*. Xilinx. 2012. URL: https://docs.xilinx.com/v/u/1.01b-English/ds744_axi_gpio.

[XIL12b] *LogiCORE IP AXI4-Stream FIFO(v2.01a)*. Xilinx. 2012. URL: https://docs.xilinx.com/v/u/en-US/ds806_axi_fifo_mm_s.

Appendix A

Code

A.1 SplitAndWrite

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  -- Uncomment the following library declaration if using
5  -- arithmetic functions with Signed or Unsigned values
6  --use IEEE.NUMERIC_STD.ALL;
7
8  -- Uncomment the following library declaration if instantiating
9  -- any Xilinx leaf cells in this code.
10 --library UNISIM;
11 --use UNISIM.VComponents.all;
12
13 entity VD_SplitAndWrite is
14     Generic (
15         DATA_WIDTH_OUT       : INTEGER := 64;
16         DATA_WIDTH_IN        : INTEGER := 64*3;
17         MEM_BUFFER_SIZE      : INTEGER := 100
18     );
19     Port (
20             ACLK             : in STD_LOGIC;
21             ARESETn          : in STD_LOGIC;
22
23             S00_AXIS_TDATA   : in STD_LOGIC_VECTOR(DATA_WIDTH_IN-1 downto 0);
24             S00_AXIS_TREADY  : out STD_LOGIC;
25             S00_AXIS_TVALID  : in STD_LOGIC;
26             S00_AXIS_TLAST   : in STD_LOGIC;
27
28             M00_AXIS_TDATA   : out STD_LOGIC_VECTOR(DATA_WIDTH_OUT-1 downto 0);
29             M00_AXIS_TREADY  : in STD_LOGIC;
30             M00_AXIS_TVALID  : out STD_LOGIC;
31             M00_AXIS_TLAST   : out STD_LOGIC
32     );
33 end VD_SplitAndWrite;
34
35 architecture Behavioral of VD_SplitAndWrite is
36
37     -- ----------- Signals ------------
38     signal data_available_flag : STD_LOGIC;
```

```
39      signal data_in : STD_LOGIC_VECTOR(DATA_WIDTH_IN-1 downto 0);
40
41      signal data_out : STD_LOGIC_VECTOR(DATA_WIDTH_OUT-1 downto 0);
42      signal go_flag : STD_LOGIC := '0';
43      signal done_flag : STD_LOGIC := '0';
44
45      type State_t is ( WaitForIncoming, SendNextChunk, WaitForDone );
46      subtype data_out_range is natural range 0 to DATA_WIDTH_IN + DATA_WIDTH_OUT;
47      signal state : State_t := WaitForIncoming;
48      signal chunk_bits_sent : data_out_range := 0;
49
50
51
52
53      -- ----------- Components ---------
54      component AXIS_To_Vector is
55          Generic (
56              DATA_WIDTH    : INTEGER := 64
57                   );
58          Port (
59                  ACLK         : in STD_LOGIC;
60                  ARESETn      : in STD_LOGIC;
61                  S_AXIS_TDATA    : in STD_LOGIC_VECTOR(DATA_WIDTH-1 downto 0);
62                  S_AXIS_TREADY   : out STD_LOGIC;
63                  S_AXIS_TVALID   : in STD_LOGIC;
64                  S_AXIS_TLAST    : in STD_LOGIC;
65
66                  stream_data     : out STD_LOGIC_VECTOR(DATA_WIDTH-1 downto 0);
67                  stream_data_av : out STD_LOGIC --When data is available
68                  );
69      end component;
70
71      component Vector_To_AXIS is
72          Generic (   DATA_WIDTH  : INTEGER := 64;
73                      SEND_LAST_AFTER : INTEGER := 100
74                   );
75          Port (  ACLK         : in STD_LOGIC;
76                  ARESETn      : in STD_LOGIC;
77
78                  M_AXIS_TDATA    : out STD_LOGIC_VECTOR(DATA_WIDTH - 1 downto 0);
79                  M_AXIS_TREADY   : in STD_LOGIC;
80                  M_AXIS_TVALID   : out STD_LOGIC;
81                  M_AXIS_TLAST    : out STD_LOGIC;
82
83                  Data_i : in STD_LOGIC_VECTOR(DATA_WIDTH - 1 downto 0);
84                  Done_o : out STD_LOGIC;
85                  Go_i    : in STD_LOGIC
86                  );
87      end component;
88
89  begin
90
91      assert DATA_WIDTH_OUT mod DATA_WIDTH_IN = 0 report "WIDTH_OUT has to be a multiple of WIDTH_IN" severity fail
92
93      -- Component handling incoming AXI stream
```

```
94      axis_input : AXIS_To_Vector
95      generic map (DATA_WIDTH_IN)
96      port map (
97          ACLK => ACLK,
98          ARESETn => ARESETn,
99          S_AXIS_TDATA => S00_AXIS_TDATA,
100         S_AXIS_TREADY => S00_AXIS_TREADY,
101         S_AXIS_TVALID => S00_AXIS_TVALID,
102         S_AXIS_TLAST => S00_AXIS_TLAST,
103         stream_data => data_in,
104         stream_data_av => data_available_flag
105     );
106
107     -- Component handling outgoing AXI Stream
108     axis_output : Vector_To_AXIS
109     generic map (DATA_WIDTH_OUT, MEM_BUFFER_SIZE)
110     port map (
111         ACLK => ACLK,
112         ARESETn => ARESETn,
113         M_AXIS_TDATA => M00_AXIS_TDATA,
114         M_AXIS_TREADY => M00_AXIS_TREADY,
115         M_AXIS_TVALID => M00_AXIS_TVALID,
116         M_AXIS_TLAST => M00_AXIS_TLAST,
117         Data_i => data_out,
118         Done_o => done_flag,
119         Go_i => go_flag
120     );
121
122     process (ACLK)
123     begin
124
125         if rising_edge(ACLK) then
126
127             if ARESETn = '0' then
128                 state <= WaitForIncoming;
129             else
130                 -- state machine
131                 case state is
132                     when WaitForIncoming =>
133                         if data_available_flag = '1' then
134                             state <= SendNextChunk;
135                             chunk_bits_sent <= 0;
136                         end if;
137
138                     when SendNextChunk =>
139                         -- determine range for sending
140                         data_out <= data_in((DATA_WIDTH_IN - chunk_bits_sent - 1)
141                             downto (DATA_WIDTH_IN - chunk_bits_sent - DATA_WIDTH_OUT));
142                         chunk_bits_sent <= chunk_bits_sent + DATA_WIDTH_OUT;
143                         go_flag <= '1'; -- signal data available for sending
144
145                         state <= WaitForDone;
146
147                     when WaitForDone =>
148                         go_flag <= '0';
```

```
                        if done_flag = '1' then
                            if chunk_bits_sent >= DATA_WIDTH_IN then
                                state <= WaitForIncoming;
                            else
                                state <= SendNextChunk;
                            end if;
                        end if;
                end case;
            end if;
        end if;

    end process;

end Behavioral;


```

Listing 8: VD_SplitAndWrite.vhd file for the ValidationData component

A.2 SensorSequencer

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

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

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

entity SensorSequencer is
    port (
        clk : in STD_LOGIC;
        resetn : in STD_LOGIC;

        trigger_poll : in STD_LOGIC;
        sensor_data : in STD_LOGIC_VECTOR(3*16-1 downto 0);
        comm_ready : in STD_LOGIC;
        comm_data_valid : in STD_LOGIC;

        comm_start_configure : out STD_LOGIC;
        comm_start_data : out STD_LOGIC;
        select_sensor : out STD_LOGIC_VECTOR(3-1 downto 0);
        data_out : out STD_LOGIC_VECTOR(3*64-1 downto 0); -- Data format (each 16B): (s1x, s1y, s1z, /, s2x, s2y,
        data_out_valid : out STD_LOGIC
    );
end SensorSequencer;

architecture Behavioral of SensorSequencer is
```

```
33
34      type SensorSequencerState_t is ( WaitForReady, Configure, WaitForConfigured, WaitForTrigger, RequestData, 
35      signal state : SensorSequencerState_t := WaitForReady;
36
37      signal current_sensor : natural range 0 to 3 := 0;
38
39  begin
40
41      process (clk)
42          variable buffer_upper_bits : integer range 0 to 64*3 := 0;
43      begin
44          if rising_edge(clk) then
45              if resetn = '0' then
46                  data_out <= (others => '0');
47                  data_out_valid <= '0';
48                  comm_start_configure <= '0';
49                  comm_start_data <= '0';
50                  current_sensor <= 0;
51              else
52                  case state is
53                      when WaitForReady =>
54                          current_sensor <= 0;
55                          if comm_ready = '1' then
56                              state <= Configure;
57                          end if;
58                      when Configure =>
59                          current_sensor <= current_sensor + 1;
60                          comm_start_configure <= '1';
61
62                          state <= WaitForConfigured;
63                      when WaitForConfigured =>
64                          comm_start_configure <= '0';
65
66                          if comm_ready = '1' then
67                              if current_sensor = 3 then
68                                  state <= WaitForTrigger;
69                              else
70                                  state <= Configure;
71                              end if;
72                          end if;
73                      when WaitForTrigger =>
74                          current_sensor <= 0;
75                          data_out_valid <= '0';
76                          if trigger_poll = '1' then
77                              state <= RequestData;
78                          end if;
79                      when RequestData =>
80                          current_sensor <= current_sensor + 1;
81                          comm_start_data <= '1';
82
83                          state <= WaitForData;
84                      when WaitForData =>
85                          comm_start_data <= '0';
86
87                          if comm_data_valid = '1' then
```

```
88                              buffer_upper_bits := 64*3 - 1 - 64*(current_sensor-1);
89                              data_out(buffer_upper_bits
90                                  downto buffer_upper_bits - 16*3 + 1) <= sensor_data;
91
92                              if current_sensor = 3 then
93                                  state <= WaitForTrigger;
94                                  data_out_valid <= '1';
95                              else
96                                  state <= RequestData;
97                              end if;
98                          end if;
99                  end case;
100             end if;
101         end if;
102     end process;
103
104     -- Asynchronous One-Hot encoding of select_sensor by current_sensor
105     -- select_sensor = (s3, s2, s1)
106     process (current_sensor)
107     begin
108         select_sensor <= (others => '0');
109         if current_sensor > 0 then
110             select_sensor(current_sensor-1) <= '1';
111         end if;
112     end process;
113
114
115 end Behavioral;
```

Listing 9: SensorSequencer.vhd for the Input Interface component

A.3 Sensor Dummy

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use ieee.numeric_std.all;
4
5  entity Sensor_Dummy is
6       Generic (
7                  HALF_AMPLITUDE  : INTEGER := 500;
8                  PERIOD : INTEGER := 1;
9                  DATA_OFFSET1 : INTEGER := 0;
10                 DATA_OFFSET2 : INTEGER := 100;
11                 DATA_OFFSET3 : INTEGER := 200
12              );
13     Port (  ACLK         : in STD_LOGIC;
14             ARESETn      : in STD_LOGIC;
15             Comm_Data    : out STD_LOGIC_VECTOR(3*16-1 downto 0);
16             Comm_Ready   : out STD_LOGIC;
17             Comm_Data_Valid  : out STD_LOGIC;
18             Comm_Start_Data  : in STD_LOGIC;
19             Sensor_Select  : in STD_LOGIC_VECTOR(2 downto 0)
20             );
```

```
21  end Sensor_Dummy;
22
23  architecture Behaviour of Sensor_Dummy is
24      signal Data_Counter1                : INTEGER   := DATA_OFFSET1;
25      signal Data_Counter2                : INTEGER   := DATA_OFFSET2;
26      signal Data_Counter3                : INTEGER   := DATA_OFFSET3;
27      signal CLK_Counter                  : INTEGER   := 0;
28      signal Duration_Counter             : INTEGER   := 0;
29      signal Ready                        : STD_LOGIC   := '1';
30
31  begin
32      Comm_Ready <= '1';
33
34      process (ACLK)
35      begin
36          if rising_edge(ACLK) then
37                if(ARESETn = '0') then
38                      Duration_Counter <= 1;
39                      Data_Counter1 <= DATA_OFFSET1;
40                      Data_Counter2 <= DATA_OFFSET2;
41                      Data_Counter3 <= DATA_OFFSET3;
42                else
43                  Duration_Counter <= Duration_Counter + 1;
44                  if (Duration_Counter = PERIOD) then
45                      Duration_Counter <= 0;
46
47                      if (Data_Counter1 = HALF_AMPLITUDE) then
48                          Data_Counter1 <= -HALF_AMPLITUDE;
49                      else
50                          Data_Counter1 <= Data_Counter1 + 1;
51                      end if;
52
53                      if (Data_Counter2 = HALF_AMPLITUDE) then
54                          Data_Counter2 <= -HALF_AMPLITUDE;
55                      else
56                          Data_Counter2 <= Data_Counter2 + 1;
57                      end if;
58
59                      if (Data_Counter3 = HALF_AMPLITUDE) then
60                          Data_Counter3 <= -HALF_AMPLITUDE;
61                      else
62                          Data_Counter3 <= Data_Counter3 + 1;
63                      end if;
64                  end if;
65              end if;
66          end if;
67      end process;
68
69      process (ACLK)
70      begin
71          if rising_edge(ACLK) then
72                if(ARESETn = '0') then
73                      Ready <= '0';
74                      Comm_Data_Valid <= '0';
75                      Comm_Data <= (others => '0');
```

```
76              else
77                    if(Comm_Start_Data = '1') then
78                        Comm_Data_Valid <= '1';
79                        if (Sensor_Select(0) = '1') then
80                            Comm_Data(47 downto 32) <= STD_LOGIC_VECTOR(to_signed(Data_Counter1, 16));
81                            Comm_Data(31 downto 16) <= STD_LOGIC_VECTOR(to_signed(Data_Counter1, 16));
82                            Comm_Data(15 downto 0) <= STD_LOGIC_VECTOR(to_signed(Data_Counter1, 16));
83                        elsif (Sensor_Select(1) = '1') then
84                            Comm_Data(47 downto 32) <= STD_LOGIC_VECTOR(to_signed(Data_Counter2, 16));
85                            Comm_Data(31 downto 16) <= STD_LOGIC_VECTOR(to_signed(Data_Counter2, 16));
86                            Comm_Data(15 downto 0) <= STD_LOGIC_VECTOR(to_signed(Data_Counter2, 16));
87                        else
88                            Comm_Data(47 downto 32) <= STD_LOGIC_VECTOR(to_signed(Data_Counter3, 16));
89                            Comm_Data(31 downto 16) <= STD_LOGIC_VECTOR(to_signed(Data_Counter3, 16));
90                            Comm_Data(15 downto 0) <= STD_LOGIC_VECTOR(to_signed(Data_Counter3, 16));
91                        end if;
92                    else
93                        Comm_Data_Valid <= '0';
94                    end if;
95              end if;
96          end if;
97      end process;
98  end Behaviour;
```

Listing 10: SensorDummy.vhd file

A.4 Deflection Estimator

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity Deflection_Estimator is
6           generic (
7                   -- Users to add parameters here
8           INPUT_DATA_WIDTH    : integer := 32; -- C_S00_AXIS_TDATA_WIDTH/9
9           CALC_DATA_WIDTH     : integer := 64;
10
11          G_ACCEL_INT         : integer := 345;   -- 2.9 mg/LSB
12                                                  -- gravitational acceleration 9.81 m/s^2 =>
13                                                  -- 1 LSB = X m/s^2
14                  -- User parameters ends
15                  -- Do not modify the parameters beyond this line
16
17
18                  -- Parameters of Axi Slave Bus Interface S00_AXIS
19                  C_S00_AXIS_TDATA_WIDTH        : integer        := 192;
20
21                  -- Parameters of Axi Master Bus Interface M00_AXIS
22                  C_M00_AXIS_TDATA_WIDTH        : integer        := 64
23          );
24          port (
25                  -- Users to add ports here
```

```
26                  aclk : in std_logic;
27                  aresetn : in std_logic;
28                  -- User ports ends
29                  -- Do not modify the ports beyond this line
30
31
32                  -- Ports of Axi Slave Bus Interface S00_AXIS
33                  s00_axis_tready        : out std_logic;
34                  s00_axis_tdata        : in std_logic_vector(C_S00_AXIS_TDATA_WIDTH-1 downto 0);
35                  s00_axis_tlast        : in std_logic;
36                  s00_axis_tvalid        : in std_logic;
37
38                  -- Ports of Axi Master Bus Interface M00_AXIS
39                  m00_axis_tvalid        : out std_logic;
40                  m00_axis_tdata        : out std_logic_vector(C_M00_AXIS_TDATA_WIDTH-1 downto 0);
41                  m00_axis_tlast        : out std_logic;
42                  m00_axis_tready        : in std_logic
43
44          );
45  end Deflection_Estimator;
46
47  architecture arch_imp of Deflection_Estimator is
48
49  type State_t is (IDLE, CALCULATE, PUSH_DATA);
50  signal state            : State_t;
51
52  signal calc_done        : std_logic;
53  signal data_pushed      : std_logic;
54  signal data_pushed_delay: std_logic;
55  signal data_received    : std_logic;
56
57  signal z0_input_data    : std_logic_vector(INPUT_DATA_WIDTH-1 downto 0);
58  signal z1_input_data    : std_logic_vector(INPUT_DATA_WIDTH-1 downto 0);
59  signal z2_input_data    : std_logic_vector(INPUT_DATA_WIDTH-1 downto 0);
60
61  -- axis over sensor 1 and sensor 2
62  signal x_deflection     : std_logic_vector(CALC_DATA_WIDTH-1 downto 0) := (others => '0');
63  -- axis over sensor 2 and sensor 3
64  signal y_deflection     : std_logic_vector(CALC_DATA_WIDTH-1 downto 0) := (others => '0');
65
66  signal output_data      : std_logic_vector(C_M00_AXIS_TDATA_WIDTH-1 downto 0);
67
68  constant G_ACCEL        : std_logic_vector(INPUT_DATA_WIDTH-1 downto 0) :=
69                                              std_logic_vector(to_unsigned(G_ACCEL_INT, 32));
70
71  signal m_axis_tlast     : std_logic;
72  signal m_axis_tvalid    : std_logic;
73  signal m_axis_tlast_delay   : std_logic;
74  signal m_axis_tvalid_delay  : std_logic;
75
76  signal s_axis_tready    : std_logic;
77  begin
78
79  -- I/O assignments
80  m00_axis_tlast  <= m_axis_tlast_delay;
```

```
81  m00_axis_tvalid <= m_axis_tvalid_delay;
82  m00_axis_tdata  <= output_data;
83
84  s00_axis_tready <= s_axis_tready;
85
86  -- control state machine:
87  CONTROL_STATE_MACHINE : process (aclk)
88  begin
89      if (rising_edge(aclk)) then
90          if (aresetn = '0') then
91              state <= IDLE;
92          else
93              case (state) is
94                  when IDLE       =>
95                  -- default state: waiting for data and fallback state in case of
96                  --                reset
97                  -- Slave actions are performed here
98                  if (data_received = '1') then
99                      state <= CALCULATE;
100                 end if;
101
102                 when CALCULATE  =>
103                 -- calculation state: state in which the calculation is done
104                 if (calc_done = '1') then
105                     state <= PUSH_DATA;
106                 end if;
107
108                 when PUSH_DATA  =>
109                 -- output state: puts calculated data to output stream
110                 -- Master actions are performed here
111                 if (data_pushed_delay = '1') then
112                     state <= IDLE;
113                 end if;
114
115                 when others     =>
116                     state <= IDLE;
117             end case;
118         end if;
119     end if;
120 end process;
121
122 -- idle state handling:
123 IDLE_PROCESS : process (aclk)
124 variable puffer : std_logic_vector(15 downto 0);
125 begin
126     if (rising_edge(aclk)) then
127         if (state = IDLE) then
128             if (aresetn = '0') then
129                 data_received   <= '0';
130                 s_axis_tready   <= '0';
131                 z0_input_data   <= (others => '0');
132                 z1_input_data   <= (others => '0');
133                 z2_input_data   <= (others => '0');
134             else
135                 s_axis_tready   <= '1';
```

```
136
137                 if (s00_axis_tvalid = '1') then
138                 -- Input data structure is:
139                 -- | x0 [16] | y0 [16] | z0 [16] | [16] | x1 [16] | y1 [16] | z1 [16] | [16] |
140                 -- | x2 [16] | y2 [16] | z2 [16] | [16] |
141                     puffer := s00_axis_tdata(C_S00_AXIS_TDATA_WIDTH-1-32 downto
142                                              C_S00_AXIS_TDATA_WIDTH-48);
143                     z0_input_data <= (others => puffer(15));
144                     z0_input_data(15 downto 0) <= puffer;
145
146                     puffer := s00_axis_tdata(C_S00_AXIS_TDATA_WIDTH-1-64-32 downto
147                                              C_S00_AXIS_TDATA_WIDTH-64-48);
148                     z1_input_data <= (others => puffer(15));
149                     z1_input_data(15 downto 0) <= puffer;
150
151                     puffer := s00_axis_tdata(C_S00_AXIS_TDATA_WIDTH-1-64-64-32 downto
152                                              C_S00_AXIS_TDATA_WIDTH-64-64-48);
153                     z2_input_data <= (others => puffer(15));
154                     z2_input_data(15 downto 0) <= puffer;
155                 end if;
156
157                 if (s00_axis_tlast = '1') then
158                     data_received   <= '1';
159                     s_axis_tready   <= '0';
160                 end if;
161             end if;
162         else
163             data_received   <= '0';
164             s_axis_tready   <= '0';
165         end if;
166     end if;
167 end process;
168
169 -- calcuation of the data:
170 CALCULATE_PROCESS : process (aclk)
171 -- axis over sensor 1 and sensor 2
172 variable pre_x_deflection   : std_logic_vector(CALC_DATA_WIDTH-1 downto 0) := (others => '0');
173 -- axis over sensor 2 and sensor 3
174 variable pre_y_deflection   : std_logic_vector(CALC_DATA_WIDTH-1 downto 0) := (others => '0');
175
176 -- axis over sensor 1 and sensor 2
177 variable x_velocity         : std_logic_vector(CALC_DATA_WIDTH-1 downto 0) := (others => '0');
178 -- axis over sensor 2 and sensor 3
179 variable y_velocity         : std_logic_vector(CALC_DATA_WIDTH-1 downto 0) := (others => '0');
180 -- axis over sensor 1 and sensor 2
181 variable pre_x_velocity     : std_logic_vector(CALC_DATA_WIDTH-1 downto 0) := (others => '0');
182 -- axis over sensor 2 and sensor 3
183 variable pre_y_velocity     : std_logic_vector(CALC_DATA_WIDTH-1 downto 0) := (others => '0');
184
185 variable x_cur_accel        : std_logic_vector(INPUT_DATA_WIDTH-1 downto 0) := (others => '0');
186 variable y_cur_accel        : std_logic_vector(INPUT_DATA_WIDTH-1 downto 0) := (others => '0');
187 begin
188     if (rising_edge(aclk)) then
189         if (state = CALCULATE) then
190             if (aresetn = '0') then
```

```
191                 x_deflection       <= (others => '0');
192                 y_deflection       <= (others => '0');
193                 pre_x_deflection   := (others => '0');
194                 pre_y_deflection   := (others => '0');
195                 x_velocity         := (others => '0');
196                 y_velocity         := (others => '0');
197                 pre_x_velocity     := (others => '0');
198                 pre_y_velocity     := (others => '0');
199                 calc_done              <= '0';
200             else
201                 -- Do calcuation here
202                 pre_x_deflection := x_deflection;
203                 pre_y_deflection := y_deflection;
204                 pre_x_velocity   := x_velocity;
205                 pre_y_velocity   := y_velocity;
206 
207                 ----------------------------------------------------------------
208                 -- x : compare sensor 1 to sensor 2 ----------------------------
209                 ----------------------------------------------------------------
210                 if (z0_input_data > G_ACCEL and z1_input_data < G_ACCEL) then
211                 -- left inclination
212                     x_cur_accel := std_logic_vector(signed(z0_input_data) - signed(G_ACCEL));
213                     x_velocity  := std_logic_vector(signed(pre_x_velocity) + signed(x_cur_accel));
214                 elsif (z0_input_data < G_ACCEL and z1_input_data > G_ACCEL) then
215                 -- right inclination
216                     x_cur_accel := std_logic_vector(signed(z1_input_data) - signed(G_ACCEL));
217                     x_velocity  := std_logic_vector(signed(pre_x_velocity) - signed(x_cur_accel));
218                 else
219                 -- no inclination
220                     x_cur_accel := (others => '0');
221                     x_velocity  := std_logic_vector(signed(pre_x_velocity) + signed(x_cur_accel));
222                 end if;
223 
224                 x_deflection <= std_logic_vector(signed(pre_x_deflection) + signed(x_velocity));
225 
226                 ----------------------------------------------------------------
227                 -- y : compare sensor 1 to sensor 2 ----------------------------
228                 ----------------------------------------------------------------
229                 if (z1_input_data < G_ACCEL and z2_input_data > G_ACCEL) then
230                 -- back inclination
231                     y_cur_accel := std_logic_vector(signed(z2_input_data) - signed(G_ACCEL));
232                     y_velocity  := std_logic_vector(signed(pre_y_velocity) + signed(y_cur_accel));
233                 elsif (z1_input_data > G_ACCEL and z2_input_data < G_ACCEL) then
234                 -- front inclination
235                     y_cur_accel := std_logic_vector(signed(z1_input_data) - signed(G_ACCEL));
236                     y_velocity  := std_logic_vector(signed(pre_y_velocity) - signed(y_cur_accel));
237                 else
238                 -- no inclination
239                     y_cur_accel := (others => '0');
240                     y_velocity  := std_logic_vector(signed(pre_y_velocity) + signed(y_cur_accel));
241                 end if;
242 
243                 y_deflection <= std_logic_vector(signed(pre_y_deflection) + signed(y_velocity));
244 
245                 -- end calcuation
```

```
246                 calc_done <= '1';
247             end if;
248         else
249             calc_done <= '0';
250         end if;
251     end if;
252 end process;
253 
254 -- push data handling:
255 PUSH_DATA_PROCESS : process (aclk)
256 begin
257     if (rising_edge(aclk)) then
258         if (state = PUSH_DATA) then
259             if (aresetn = '0') then
260                 data_pushed     <= '0';
261                 m_axis_tvalid   <= '0';
262                 m_axis_tlast    <= '0';
263                 output_data     <= (others => '0');
264             else
265                 m_axis_tvalid   <= '1';
266                 m_axis_tlast    <= '1';
267 
268                 if (m00_axis_tready = '1') then
269                     output_data(C_M00_AXIS_TDATA_WIDTH-1 downto INPUT_DATA_WIDTH)
270                                             <= x_deflection(CALC_DATA_WIDTH-1 downto 32);
271                     output_data(INPUT_DATA_WIDTH-1 downto 0)
272                                             <= y_deflection(CALC_DATA_WIDTH-1 downto 32);
273                 end if;
274 
275                 if (m_axis_tvalid_delay = '1') then
276                     data_pushed <= '1';
277                 end if;
278             end if;
279         else
280             data_pushed     <= '0';
281             m_axis_tvalid   <= '0';
282             m_axis_tlast    <= '0';
283         end if;
284     end if;
285 end process;
286 
287 -- Delay for one clock cycle for certain signals
288 MAKE_DELAY : process (aclk)
289 begin
290     if (rising_edge(aclk)) then
291         if (aresetn = '0') then
292             m_axis_tvalid_delay <= '0';
293             m_axis_tlast_delay  <= '0';
294 
295             data_pushed_delay   <= '0';
296         else
297             m_axis_tvalid_delay <= m_axis_tvalid;
298             m_axis_tlast_delay  <= m_axis_tlast;
299 
300             data_pushed_delay   <= data_pushed;
```

```
        end if;
    end if;
end process;

end arch_imp;
```

Listing 11: Code for "Deflection_Estimator"

A.5 Digital P Controller

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

entity Digital_P_Controller is
        generic (
                -- Users to add parameters here
        KP_WIDTH            : integer := 32;
        IN_CORE_DATA_WIDTH  : integer := 32; -- half of C_S00_AXIS_TDATA_WIDTH
                -- User parameters ends
                -- Do not modify the parameters beyond this line


                -- Parameters of Axi Slave Bus Interface S00_AXIS
                C_S00_AXIS_TDATA_WIDTH        : integer        := 64;

                -- Parameters of Axi Master Bus Interface M00_AXIS
                C_M00_AXIS_TDATA_WIDTH        : integer        := 32
        );
        port (
                -- Users to add ports here

                kp: in std_logic_vector(KP_WIDTH-1 downto 0);
                aclk : in std_logic;
                aresetn : in std_logic;

                upper : in std_logic;

                reference_variable : in std_logic_vector(KP_WIDTH-1 downto 0);
                -- User ports ends
                -- Do not modify the ports beyond this line


                -- Ports of Axi Slave Bus Interface S00_AXIS
                s00_axis_tready        : out std_logic;
                s00_axis_tdata        : in std_logic_vector(C_S00_AXIS_TDATA_WIDTH-1 downto 0);
                s00_axis_tlast        : in std_logic;
                s00_axis_tvalid        : in std_logic;

                -- Ports of Axi Master Bus Interface M00_AXIS
                m00_axis_tvalid        : out std_logic;
                m00_axis_tdata        : out std_logic_vector(C_M00_AXIS_TDATA_WIDTH-1 downto 0);
                m00_axis_tlast        : out std_logic;
```

```
44                m00_axis_tready        : in std_logic
45
46        );
47    end Digital_P_Controller;
48
49    architecture arch_imp of Digital_P_Controller is
50
51    type State_t is (IDLE, CALCULATE, PUSH_DATA);
52    signal state            : State_t;
53
54    signal calc_done        : std_logic;
55    signal data_pushed      : std_logic;
56    signal data_pushed_delay: std_logic;
57    signal data_received    : std_logic;
58
59    signal input_data       : std_logic_vector(IN_CORE_DATA_WIDTH-1 downto 0);
60    signal calculated_data  : std_logic_vector((IN_CORE_DATA_WIDTH+KP_WIDTH)-1 downto 0);
61    signal output_data      : std_logic_vector(C_M00_AXIS_TDATA_WIDTH-1 downto 0);
62
63    signal m_axis_tlast     : std_logic;
64    signal m_axis_tvalid    : std_logic;
65    signal m_axis_tlast_delay   : std_logic;
66    signal m_axis_tvalid_delay  : std_logic;
67
68    signal s_axis_tready    : std_logic;
69    begin
70
71    -- I/O assignments
72    m00_axis_tlast  <= m_axis_tlast_delay;
73    m00_axis_tvalid <= m_axis_tvalid_delay;
74    m00_axis_tdata  <= output_data;
75
76    s00_axis_tready <= s_axis_tready;
77
78    -- control state machine:
79    CONTROL_STATE_MACHINE : process (aclk)
80    begin
81        if (rising_edge(aclk)) then
82            if (aresetn = '0') then
83                state <= IDLE;
84            else
85                case (state) is
86                    when IDLE       =>
87                    -- default state: waiting for data and fallback state in case of
88                    --                reset
89                    -- Slave actions are performed here
90                    if (data_received = '1') then
91                        state <= CALCULATE;
92                    end if;
93
94                    when CALCULATE  =>
95                    -- calculation state: state in which the calculation is done
96                    if (calc_done = '1') then
97                        state <= PUSH_DATA;
98                    end if;
```

```
99
100                 when PUSH_DATA  =>
101                 -- output state: puts calculated data to output stream
102                 -- Master actions are performed here
103                 if (data_pushed_delay = '1') then
104                     state <= IDLE;
105                 end if;
106
107                 when others     =>
108                     state <= IDLE;
109             end case;
110         end if;
111     end if;
112 end process;
113
114 -- idle state handling:
115 IDLE_PROCESS : process (aclk)
116 begin
117     if (rising_edge(aclk)) then
118         if (state = IDLE) then
119             if (aresetn = '0') then
120                 data_received   <= '0';
121                 s_axis_tready   <= '0';
122                 input_data      <= (others => '0');
123             else
124                 s_axis_tready   <= '1';
125
126                 if (s00_axis_tvalid = '1') then
127                     if (upper = '0') then
128                         input_data  <= s00_axis_tdata(IN_CORE_DATA_WIDTH-1 downto 0);
129                     else
130                         input_data  <= s00_axis_tdata(C_S00_AXIS_TDATA_WIDTH-1 downto
131                                                       IN_CORE_DATA_WIDTH);
132                     end if;
133                 end if;
134
135                 if (s00_axis_tlast = '1') then
136                     data_received   <= '1';
137                     s_axis_tready   <= '0';
138                 end if;
139             end if;
140         else
141             data_received   <= '0';
142             s_axis_tready   <= '0';
143         end if;
144     end if;
145 end process;
146
147 -- calcuation of the data:
148 CALCULATE_PROCESS : process (aclk)
149 variable puffer : signed (IN_CORE_DATA_WIDTH-1 downto 0) := (others => '0');
150 begin
151     if (rising_edge(aclk)) then
152         if (state = CALCULATE) then
153             if (aresetn = '0') then
```

```
154                 calculated_data <= (others => '0');
155                 calc_done <= '0';
156             else
157                 -- Do calcuation here
158                 puffer := signed(reference_variable) - signed(input_data);
159                 calculated_data <= std_logic_vector(signed(kp) * puffer);
160                 -- end calcuation
161                 calc_done <= '1';
162             end if;
163         else
164             calc_done <= '0';
165         end if;
166     end if;
167 end process;
168 
169 -- push data handling:
170 PUSH_DATA_PROCESS : process (aclk)
171 begin
172     if (rising_edge(aclk)) then
173         if (state = PUSH_DATA) then
174             if (aresetn = '0') then
175                 data_pushed     <= '0';
176                 m_axis_tvalid   <= '0';
177                 m_axis_tlast    <= '0';
178                 output_data     <= (others => '0');
179             else
180                 m_axis_tvalid   <= '1';
181                 m_axis_tlast    <= '1';
182 
183                 if (m00_axis_tready = '1') then
184                     output_data <= calculated_data((IN_CORE_DATA_WIDTH+KP_WIDTH)-1 downto
185                                                    IN_CORE_DATA_WIDTH);
186                 end if;
187 
188                 if (m_axis_tvalid_delay = '1') then
189                     data_pushed <= '1';
190                 end if;
191             end if;
192         else
193             data_pushed     <= '0';
194             m_axis_tvalid   <= '0';
195             m_axis_tlast    <= '0';
196         end if;
197     end if;
198 end process;
199 
200 -- Delay for one clock cycle for certain signals
201 MAKE_DELAY : process (aclk)
202 begin
203     if (rising_edge(aclk)) then
204         if (aresetn = '0') then
205             m_axis_tvalid_delay <= '0';
206             m_axis_tlast_delay  <= '0';
207 
208             data_pushed_delay   <= '0';
```

```
209         else
210             m_axis_tvalid_delay <= m_axis_tvalid;
211             m_axis_tlast_delay  <= m_axis_tlast;
212 
213             data_pushed_delay   <= data_pushed;
214         end if;
215     end if;
216 end process;
217 
218 end arch_imp;
```

Listing 12: Code for "Digital_P_Controller"

A.6 Sensor Communication top level

```
1  
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
5  
6  entity spiTestIP_v1_0 is
7          generic (
8                  -- Users to add parameters here
9  
10                 -- User parameters ends
11                 -- Do not modify the parameters beyond this line
12 
13 
14                 -- Parameters of Axi Master Bus Interface M00_AXI
15                 C_M00_AXI_START_DATA_VALUE        : std_logic_vector        := x"AA000000";
16                 C_M00_AXI_TARGET_SLAVE_BASE_ADDR        : std_logic_vector        := x"40000000";
17                 C_M00_AXI_ADDR_WIDTH        : integer        := 32;
18                 C_M00_AXI_DATA_WIDTH        : integer        := 32;
19                 C_M00_AXI_TRANSACTIONS_NUM        : integer        := 4
20         );
21         port (
22                 -- Users to add ports here
23 
24                 -- User ports ends
25                 -- Do not modify the ports beyond this line
26 
27 
28                 -- Ports of Axi Master Bus Interface M00_AXI
29                 m00_reset       : in std_logic;
30                 m00_axi_error        : out std_logic;
31                 m00_axi_aclk        : in std_logic;
32                 m00_axi_awaddr        : out std_logic_vector(C_M00_AXI_ADDR_WIDTH-1 downto 0);
33                 m00_axi_awprot        : out std_logic_vector(2 downto 0);
34                 m00_axi_awvalid        : out std_logic;
35                 m00_axi_awready        : in std_logic;
36                 m00_axi_wdata        : out std_logic_vector(C_M00_AXI_DATA_WIDTH-1 downto 0);
37                 m00_axi_wstrb        : out std_logic_vector(C_M00_AXI_DATA_WIDTH/8-1 downto 0);
38                 m00_axi_wvalid        : out std_logic;
```

```
39                  m00_axi_wready        : in std_logic;
40                  m00_axi_bresp        : in std_logic_vector(1 downto 0);
41                  m00_axi_bvalid        : in std_logic;
42                  m00_axi_bready        : out std_logic;
43                  m00_axi_araddr        : out std_logic_vector(C_M00_AXI_ADDR_WIDTH-1 downto 0);
44                  m00_axi_arprot        : out std_logic_vector(2 downto 0);
45                  m00_axi_arvalid        : out std_logic;
46                  m00_axi_arready        : in std_logic;
47                  m00_axi_rdata        : in std_logic_vector(C_M00_AXI_DATA_WIDTH-1 downto 0);
48                  m00_axi_rresp        : in std_logic_vector(1 downto 0);
49                  m00_axi_rvalid        : in std_logic;
50                  m00_axi_rready        : out std_logic;
51
52                  m00_sensor_data : out std_logic_vector(16*3-1 downto 0);
53                  m00_comm_ready : out std_logic;
54                  m00_comm_start_configure : std_logic;
55                  m00_comm_start_data : std_logic
56          );
57  end spiTestIP_v1_0;
58
59  architecture arch_imp of spiTestIP_v1_0 is
60
61          -- component declaration
62          component spiTestIP_v1_0_M00_AXI is
63                  generic (
64                  C_M_START_DATA_VALUE        : std_logic_vector        := x"AA000000";
65                  C_M_TARGET_SLAVE_BASE_ADDR        : std_logic_vector        := x"40000000";
66                  C_M_AXI_ADDR_WIDTH        : integer        := 32;
67                  C_M_AXI_DATA_WIDTH        : integer        := 32;
68                  C_M_TRANSACTIONS_NUM        : integer        := 4
69                  );
70                  port (
71
72                  RESET   : in std_logic;
73                  ERROR        : out std_logic;
74
75
76
77                  M_AXI_ACLK        : in std_logic;
78                  M_AXI_AWADDR        : out std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0);
79                  M_AXI_AWPROT        : out std_logic_vector(2 downto 0);
80                  M_AXI_AWVALID        : out std_logic;
81                  M_AXI_AWREADY        : in std_logic;
82                  M_AXI_WDATA        : out std_logic_vector(C_M_AXI_DATA_WIDTH-1 downto 0);
83                  M_AXI_WSTRB        : out std_logic_vector(C_M_AXI_DATA_WIDTH/8-1 downto 0);
84                  M_AXI_WVALID        : out std_logic;
85                  M_AXI_WREADY        : in std_logic;
86                  M_AXI_BRESP        : in std_logic_vector(1 downto 0);
87                  M_AXI_BVALID        : in std_logic;
88                  M_AXI_BREADY        : out std_logic;
89                  M_AXI_ARADDR        : out std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0);
90                  M_AXI_ARPROT        : out std_logic_vector(2 downto 0);
91                  M_AXI_ARVALID        : out std_logic;
92                  M_AXI_ARREADY        : in std_logic;
93                  M_AXI_RDATA        : in std_logic_vector(C_M_AXI_DATA_WIDTH-1 downto 0);
```

```
94                  M_AXI_RRESP         : in std_logic_vector(1 downto 0);
95                  M_AXI_RVALID         : in std_logic;
96                  M_AXI_RREADY         : out std_logic
97                  );
98          end component spiTestIP_v1_0_M00_AXI;
99
100
101 begin
102
103 -- Instantiation of Axi Bus Interface M00_AXI
104 spiTestIP_v1_0_M00_AXI_inst : spiTestIP_v1_0_M00_AXI
105         generic map (
106                 C_M_START_DATA_VALUE        => C_M00_AXI_START_DATA_VALUE,
107                 C_M_TARGET_SLAVE_BASE_ADDR        => C_M00_AXI_TARGET_SLAVE_BASE_ADDR,
108                 C_M_AXI_ADDR_WIDTH        => C_M00_AXI_ADDR_WIDTH,
109                 C_M_AXI_DATA_WIDTH        => C_M00_AXI_DATA_WIDTH,
110                 C_M_TRANSACTIONS_NUM        => C_M00_AXI_TRANSACTIONS_NUM
111         )
112         port map (
113             sensor_data => m00_sensor_data,
114                 comm_ready => comm_ready,
115                 comm_start_configure => comm_start_configure,
116                 comm_start_data => comm_start_data,
117                 ERROR        => m00_axi_error,
118                 M_AXI_ACLK        => m00_axi_aclk,
119                 RESET   => m00_reset,
120                 M_AXI_AWADDR        => m00_axi_awaddr,
121                 M_AXI_AWPROT        => m00_axi_awprot,
122                 M_AXI_AWVALID        => m00_axi_awvalid,
123                 M_AXI_AWREADY        => m00_axi_awready,
124                 M_AXI_WDATA        => m00_axi_wdata,
125                 M_AXI_WSTRB        => m00_axi_wstrb,
126                 M_AXI_WVALID        => m00_axi_wvalid,
127                 M_AXI_WREADY        => m00_axi_wready,
128                 M_AXI_BRESP        => m00_axi_bresp,
129                 M_AXI_BVALID        => m00_axi_bvalid,
130                 M_AXI_BREADY        => m00_axi_bready,
131                 M_AXI_ARADDR        => m00_axi_araddr,
132                 M_AXI_ARPROT        => m00_axi_arprot,
133                 M_AXI_ARVALID        => m00_axi_arvalid,
134                 M_AXI_ARREADY        => m00_axi_arready,
135                 M_AXI_RDATA        => m00_axi_rdata,
136                 M_AXI_RRESP        => m00_axi_rresp,
137                 M_AXI_RVALID        => m00_axi_rvalid,
138                 M_AXI_RREADY        => m00_axi_rready
139         );
140
141         -- Add user logic here
142
143
144         -- User logic ends
145
146 end arch_imp;
147
```

```
148
```

Listing 13: Code for "Deflection_Estimator"

A.7 Sensor Communication

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity spiTestIP_v1_0_M00_AXI is
6           generic (
7                   -- Users to add parameters here
8
9                   -- User parameters ends
10                  -- Do not modify the parameters beyond this line
11
12                  -- The master will start generating data from the C_M_START_DATA_VALUE value
13                  C_M_START_DATA_VALUE        : std_logic_vector        := x"AA000000";
14                  -- The master requires a target slave base address.
15      -- The master will initiate read and write transactions on the slave with base address specified here as a
16                  C_M_TARGET_SLAVE_BASE_ADDR        : std_logic_vector        := x"40000000";
17                  -- Width of M_AXI address bus.
18      -- The master generates the read and write addresses of width specified as C_M_AXI_ADDR_WIDTH.
19                  C_M_AXI_ADDR_WIDTH        : integer        := 32;
20                  -- Width of M_AXI data bus.
21      -- The master issues write data and accept read data where the width of the data bus is C_M_AXI_DATA_WIDTH
22                  C_M_AXI_DATA_WIDTH        : integer        := 32;
23                  -- Transaction number is the number of write
24      -- and read transactions the master will perform as a part of this example memory test.
25                  C_M_TRANSACTIONS_NUM        : integer        := 4
26          );
27          port (
28                  -- Users to add ports here
29
30                  -- User ports ends
31                  -- Do not modify the ports beyond this line
32
33
34                  -- Asserts when ERROR is detected
35                  ERROR        : out std_logic;
36                  -- AXI clock signal
37                  M_AXI_ACLK        : in std_logic;
38                  -- resets the whole core with all state machines
39                  RESET : in std_logic;
40
41
42                  sensor_data : out std_logic_vector(16*3-1 downto 0);
43                  comm_ready : out std_logic;
44                  comm_start_configure : std_logic;
45                  comm_start_data : std_logic;
46
47                  -- Master Interface Write Address Channel ports. Write address (issued by master)
```

```
48                  M_AXI_AWADDR        : out std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0);
49                  -- Write channel Protection type.
50      -- This signal indicates the privilege and security level of the transaction,
51      -- and whether the transaction is a data access or an instruction access.
52                  M_AXI_AWPROT        : out std_logic_vector(2 downto 0);
53                  -- Write address valid.
54      -- This signal indicates that the master signaling valid write address and control information.
55                  M_AXI_AWVALID       : out std_logic;
56                  -- Write address ready.
57      -- This signal indicates that the slave is ready to accept an address and associated control signals.
58                  M_AXI_AWREADY       : in std_logic;
59                  -- Master Interface Write Data Channel ports. Write data (issued by master)
60                  M_AXI_WDATA         : out std_logic_vector(C_M_AXI_DATA_WIDTH-1 downto 0);
61                  -- Write strobes.
62      -- This signal indicates which byte lanes hold valid data.
63      -- There is one write strobe bit for each eight bits of the write data bus.
64                  M_AXI_WSTRB         : out std_logic_vector(C_M_AXI_DATA_WIDTH/8-1 downto 0);
65                  -- Write valid. This signal indicates that valid write data and strobes are available.
66                  M_AXI_WVALID        : out std_logic;
67                  -- Write ready. This signal indicates that the slave can accept the write data.
68                  M_AXI_WREADY        : in std_logic;
69                  -- Master Interface Write Response Channel ports.
70      -- This signal indicates the status of the write transaction.
71                  M_AXI_BRESP         : in std_logic_vector(1 downto 0);
72                  -- Write response valid.
73      -- This signal indicates that the channel is signaling a valid write response
74                  M_AXI_BVALID        : in std_logic;
75                  -- Response ready. This signal indicates that the master can accept a write response.
76                  M_AXI_BREADY        : out std_logic;
77                  -- Master Interface Read Address Channel ports. Read address (issued by master)
78                  M_AXI_ARADDR        : out std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0);
79                  -- Protection type.
80      -- This signal indicates the privilege and security level of the transaction,
81      -- and whether the transaction is a data access or an instruction access.
82                  M_AXI_ARPROT        : out std_logic_vector(2 downto 0);
83                  -- Read address valid.
84      -- This signal indicates that the channel is signaling valid read address and control information.
85                  M_AXI_ARVALID       : out std_logic;
86                  -- Read address ready.
87      -- This signal indicates that the slave is ready to accept an address and associated control signals.
88                  M_AXI_ARREADY       : in std_logic;
89                  -- Master Interface Read Data Channel ports. Read data (issued by slave)
90                  M_AXI_RDATA         : in std_logic_vector(C_M_AXI_DATA_WIDTH-1 downto 0);
91                  -- Read response. This signal indicates the status of the read transfer.
92                  M_AXI_RRESP         : in std_logic_vector(1 downto 0);
93                  -- Read valid. This signal indicates that the channel is signaling the required read data.
94                  M_AXI_RVALID        : in std_logic;
95                  -- Read ready. This signal indicates that the master can accept the read data and response inform
96                  M_AXI_RREADY        : out std_logic
97          );
98  end spiTestIP_v1_0_M00_AXI;
99
100 architecture implementation of spiTestIP_v1_0_M00_AXI is
101
102         -- function called clogb2 that returns an integer which has the
```

```
103         -- value of the ceiling of the log base 2
104         function clogb2 (bit_depth : integer) return integer is
105                  variable depth  : integer := bit_depth;
106                  variable count  : integer := 1;
107          begin
108                   for clogb2 in 1 to bit_depth loop  -- Works for up to 32 bit integers
109               if (bit_depth <= 2) then
110                 count := 1;
111               else
112                 if(depth <= 1) then
113                         count := count;
114                       else
115                         depth := depth / 2;
116                   count := count + 1;
117                       end if;
118                     end if;
119           end loop;
120           return(count);
121         end;
122
123         -- Example user application signals
124
125         signal my_read_data        : std_logic_vector(C_M_AXI_DATA_WIDTH-1 downto 0);
126
127
128         -- TRANS_NUM_BITS is the width of the index counter for
129         -- number of write or read transaction..
130          constant  TRANS_NUM_BITS  : integer := clogb2(C_M_TRANSACTIONS_NUM-1);
131
132         -- Example State machine to initialize counter, initialize write transactions,
133          -- initialize read transactions and comparison of read data with the
134          -- written data words.
135          type state is ( IDLE,
136                        PREPARE_READ,
137                        PREPARE_WRITE,
138                        READ,
139                        WRITE,
140                        FINISH);
141
142          signal read_write : std_logic; -- read 0, write 1
143
144          signal mst_exec_state  : state := IDLE ;
145
146          type spi_control_state_t is (IDLE,
147          START_CONFIG,
148          WAIT_FOR_CONFIG,
149          START_TRANSFER,
150          WAIT_FOR_TRANSFER,
151          FINISH);
152          signal spi_control_state : spi_control_state_t := IDLE ;
153
154          signal startup_done : std_logic := '0';
155
156          signal config_transfer : std_logic := '0';
157
```

```
158         type config_state_type is (IDLE,
159         RESET_SPI,
160         ENABLE_TRANSMIT_INTERRUPT,
161         DISABLE_INTERRUPT,
162         DESELECT_SLAVE,
163         READ_STATUS_REG,
164         WRITE_STATUS_REG,
165         FINISH );
166         signal config_state : config_state_type:= IDLE;
167         signal config_state_wait : integer range 0 to 3 := 0;
168         signal config_state_status_register : std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0);
169         signal config_done_pulse : std_logic:= '0';
170         signal config_start_pulse: std_logic:= '0';
171
172
173         type transfer_state_type is (IDLE,
174         SET_DATA_REGISTER,
175         SET_SLAVE_REGISTER,
176         READ_CONTROL_REG,
177         WRITE_CONTROL_REG,
178         READ_STATUS_REG_WHILE,
179         DESELECT_SLAVE,
180         READ_DATA, FINISH);
181         signal transfer_state : transfer_state_type:=IDLE;
182         signal transfer_state_wait : integer range 0 to 3:= 0;
183         signal transfer_temp_register : std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0);
184         signal spi_receive_data : std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0);
185         signal spi_send_data : std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0);
186         signal transfer_done_pulse : std_logic:= '0';
187         signal transfer_start_pulse : std_logic:= '0';
188
189
190         -- Initiate AXI transactions
191     signal init_axi_txn        :  std_logic;
192
193     -- AXI active low reset signal
194     signal m_axi_areset        : std_logic;
195
196         -- AXI4LITE signals
197         --write address valid
198         signal axi_awvalid        : std_logic;
199         --write data valid
200         signal axi_wvalid        : std_logic;
201         --read address valid
202         signal axi_arvalid        : std_logic;
203         --read data acceptance
204         signal axi_rready        : std_logic;
205         --write response acceptance
206         signal axi_bready        : std_logic;
207         --write address
208         signal axi_awaddr        : std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0);
209         --write data
210         signal axi_wdata        : std_logic_vector(C_M_AXI_DATA_WIDTH-1 downto 0);
211         --read addresss
212         signal axi_araddr        : std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0);
```

```
213         --Asserts when there is a write response error
214         signal write_resp_error        : std_logic;
215         --Asserts when there is a read response error
216         signal read_resp_error         : std_logic;
217         --A pulse to initiate a write transaction
218         signal start_single_write      : std_logic;
219         --A pulse to initiate a read transaction
220         signal start_single_read       : std_logic;
221         --flag that marks the completion of write trasactions.
222         -- The number of write transaction is user selected by the parameter C_M_TRANSACTIONS_NUM.
223         signal write_done              : std_logic;
224         --flag that marks the completion of read trasactions.
225         -- The number of read transaction is user selected by the parameter C_M_TRANSACTIONS_NUM
226         signal read_done               : std_logic;
227         --The error register is asserted when any of the write response error,
228         -- read response error or the data mismatch flags are asserted.
229         signal error_reg               : std_logic;
230         --Expected read data used to compare with the read data.
231         signal expected_rdata          : std_logic_vector(C_M_AXI_DATA_WIDTH-1 downto 0);
232         --Flag marks the completion of comparison of the read data with the expected read data
233         signal last_read               : std_logic;
234         signal init_txn_ff             : std_logic;
235         signal init_txn_ff2            : std_logic;
236         signal init_txn_edge           : std_logic;
237         signal init_txn_pulse          : std_logic;
238
239
240 begin
241         -- I/O Connections assignments
242
243         --Adding the offset address to the base addr of the slave
244         M_AXI_AWADDR        <= axi_awaddr;
245         --AXI 4 write data
246         M_AXI_WDATA         <= axi_wdata;
247         M_AXI_AWPROT        <= "000";
248         M_AXI_AWVALID       <= axi_awvalid;
249         --Write Data(W)
250         M_AXI_WVALID        <= axi_wvalid;
251         --Set all byte strobes in this example
252         M_AXI_WSTRB         <= "1111";
253         --Write Response (B)
254         M_AXI_BREADY        <= axi_bready;
255         --Read Address (AR)
256         M_AXI_ARADDR        <= axi_araddr;
257         M_AXI_ARVALID       <= axi_arvalid;
258         M_AXI_ARPROT        <= "001";
259         --Read and Read Response (R)
260         M_AXI_RREADY        <= axi_rready;
261
262         m_axi_areset <= RESET;
263         --Example design I/O
264         --init_txn_pulse        <= ( not init_txn_ff2)  and  init_txn_ff;
265
266         ----------------------
267         --Write Address Channel
```

```
268         ----------------------
269
270         -- The purpose of the write address channel is to request the address and
271         -- command information for the entire transaction.  It is a single beat
272         -- of information.
273
274         -- Note for this example the axi_awvalid/axi_wvalid are asserted at the same
275         -- time, and then each is deasserted independent from each other.
276         -- This is a lower-performance, but simplier control scheme.
277
278         -- AXI VALID signals must be held active until accepted by the partner.
279
280         -- A data transfer is accepted by the slave when a master has
281         -- VALID data and the slave acknoledges it is also READY. While the master
282         -- is allowed to generated multiple, back-to-back requests by not
283         -- deasserting VALID, this design will add rest cycle for
284         -- simplicity.
285
286         -- Since only one outstanding transaction is issued by the user design,
287         -- there will not be a collision between a new request and an accepted
288         -- request on the same clock cycle.
289
290           process(M_AXI_ACLK)
291           begin
292             if (rising_edge (M_AXI_ACLK)) then
293               --Only VALID signals must be deasserted during reset per AXI spec
294               --Consider inverting then registering active-low reset for higher fmax
295               if (m_axi_areset = '0' or init_txn_pulse = '1') then
296                 axi_awvalid <= '0';
297               else
298                 --Signal a new address/data command is available by user logic
299                 if (start_single_write = '1') then
300                   axi_awvalid <= '1';
301                 elsif (M_AXI_AWREADY = '1' and axi_awvalid = '1') then
302                   --Address accepted by interconnect/slave (issue of M_AXI_AWREADY by slave)
303                   axi_awvalid <= '0';
304                 end if;
305               end if;
306             end if;
307           end process;
308
309
310
311         ----------------------
312         --Write Data Channel
313         ----------------------
314
315         --The write data channel is for transfering the actual data.
316         --The data generation is speific to the example design, and
317         --so only the WVALID/WREADY handshake is shown here
318
319            process(M_AXI_ACLK)
320            begin
321              if (rising_edge (M_AXI_ACLK)) then
322                if (m_axi_areset = '0' or init_txn_pulse = '1' ) then
```

```
323                 axi_wvalid <= '0';
324               else
325                 if (start_single_write = '1') then
326                   --Signal a new address/data command is available by user logic
327                   axi_wvalid <= '1';
328                 elsif (M_AXI_WREADY = '1' and axi_wvalid = '1') then
329                   --Data accepted by interconnect/slave (issue of M_AXI_WREADY by slave)
330                   axi_wvalid <= '0';
331                 end if;
332               end if;
333             end if;
334           end process;
335
336
337         ------------------------------
338         --Write Response (B) Channel
339         ------------------------------
340
341         --The write response channel provides feedback that the write has committed
342         --to memory. BREADY will occur after both the data and the write address
343         --has arrived and been accepted by the slave, and can guarantee that no
344         --other accesses launched afterwards will be able to be reordered before it.
345
346         --The BRESP bit [1] is used indicate any errors from the interconnect or
347         --slave for the entire write burst. This example will capture the error.
348
349         --While not necessary per spec, it is advisable to reset READY signals in
350         --case of differing reset latencies between master/slave.
351
352           process(M_AXI_ACLK)
353           begin
354             if (rising_edge (M_AXI_ACLK)) then
355               if (m_axi_areset = '0' or init_txn_pulse = '1') then
356                 axi_bready <= '0';
357               else
358                 if (M_AXI_BVALID = '1' and axi_bready = '0') then
359                   -- accept/acknowledge bresp with axi_bready by the master
360                   -- when M_AXI_BVALID is asserted by slave
361                    axi_bready <= '1';
362                 elsif (axi_bready = '1') then
363                   -- deassert after one clock cycle
364                   axi_bready <= '0';
365                 end if;
366               end if;
367             end if;
368           end process;
369         --Flag write errors
370           write_resp_error <= (axi_bready and M_AXI_BVALID and M_AXI_BRESP(1));
371
372
373         ------------------------------
374         --Read Address Channel
375         ------------------------------
376
377
```

```
    -- A new axi_arvalid is asserted when there is a valid read address
    -- available by the master. start_single_read triggers a new read
    -- transaction
    process(M_AXI_ACLK)
    begin
      if (rising_edge (M_AXI_ACLK)) then
        if (m_axi_areset = '0' or init_txn_pulse = '1') then
          axi_arvalid <= '0';
        else
          if (start_single_read = '1') then
            --Signal a new read address command is available by user logic
            axi_arvalid <= '1';
          elsif (M_AXI_ARREADY = '1' and axi_arvalid = '1') then
          --RAddress accepted by interconnect/slave (issue of M_AXI_ARREADY by slave)
            axi_arvalid <= '0';
          end if;
        end if;
      end if;
    end process;


  ---------------------------------
  --Read Data (and Response) Channel
  ---------------------------------

  --The Read Data channel returns the results of the read request
  --The master will accept the read data by asserting axi_rready
  --when there is a valid read data available.
  --While not necessary per spec, it is advisable to reset READY signals in
  --case of differing reset latencies between master/slave.

    process(M_AXI_ACLK)
    begin
      if (rising_edge (M_AXI_ACLK)) then
        if (m_axi_areset = '0' or init_txn_pulse = '1') then
          axi_rready <= '1';
        else
          if (M_AXI_RVALID = '1' and axi_rready = '0') then
           -- accept/acknowledge rdata/rresp with axi_rready by the master
           -- when M_AXI_RVALID is asserted by slave
            axi_rready <= '1';
          elsif (axi_rready = '1') then
            -- deassert after one clock cycle
            axi_rready <= '0';
          end if;
        end if;
      end if;
    end process;

  --Flag write errors
    read_resp_error <= (axi_rready and M_AXI_RVALID and M_AXI_RRESP(1));


  -- Expected read data
```

```
433             process(M_AXI_ACLK)
434             begin
435               if (rising_edge (M_AXI_ACLK)) then
436                 if (m_axi_areset = '0' or init_txn_pulse = '1' ) then
437                   my_read_data         <= (others => '0');
438                 elsif (M_AXI_RVALID = '1' and axi_rready = '1') then
439                   -- Signals a new write address/ write data is
440                   -- available by user logic
441                   my_read_data         <= std_logic_vector(M_AXI_RDATA);
442                 end if;
443               end if;
444             end process;
445
446
447           --implement master command interface state machine
448           process(M_AXI_ACLK)
449           begin
450             if (rising_edge (M_AXI_ACLK)) then
451               if (m_axi_areset = '0' ) then
452                 -- reset condition
453                 -- All the signals are ed default values under reset condition
454                 mst_exec_state  <= IDLE;
455                 init_axi_txn <= '0';
456                 start_single_write <= '0';
457                 start_single_read  <= '0';
458                 ERROR <= '0';
459               else
460                 -- state transition
461                 case (mst_exec_state) is
462
463                   when IDLE =>
464                        if (init_txn_pulse = '1') then
465                            if (read_write = '0') then
466                                mst_exec_state  <= PREPARE_READ;
467                            else
468                                mst_exec_state  <= PREPARE_WRITE;
469                            end if;
470                        end if;
471
472                   when PREPARE_READ =>
473                        if (start_single_read = '0') then
474                            start_single_read <= '1';
475                        else
476                            start_single_read <= '0';
477                            mst_exec_state  <= READ;
478                        end if;
479
480                   when PREPARE_WRITE =>
481                        if (start_single_write = '0') then
482                            start_single_write <= '1';
483                        else
484                            start_single_write <= '0';
485                            mst_exec_state  <= WRITE;
486                        end if;
487
```

```
488                 when READ =>
489                      if (read_done = '1') then
490                          mst_exec_state  <= FINISH;
491                      end if;
492
493                 when WRITE =>
494                      if (write_done = '1') then
495                          mst_exec_state  <= FINISH;
496                      end if;
497
498                 when FINISH =>
499                      init_axi_txn <= '0';
500                      mst_exec_state  <= IDLE;
501                 when others  =>
502                     mst_exec_state  <= IDLE;
503               end case  ;
504             end if;
505           end if;
506         end process;
507
508       --/*
509       -- Check for write completion.
510       --
511       -- */
512         process(M_AXI_ACLK)
513         begin
514           if (rising_edge (M_AXI_ACLK)) then
515             if (m_axi_areset = '0' or init_txn_pulse = '1') then
516               -- reset condition
517               write_done <= '0';
518             else
519               if (M_AXI_BVALID = '1' and axi_bready = '1') then
520                 --The write_done should be associated with a bready response
521                  write_done <= '1';
522               else
523                  write_done <= '0';
524               end if;
525             end if;
526           end if;
527         end process;
528
529
530       --/*
531       -- Check for last read completion.
532       -- */
533         process(M_AXI_ACLK)
534         begin
535           if (rising_edge (M_AXI_ACLK)) then
536             if (m_axi_areset = '0' or init_txn_pulse = '1') then
537               read_done <= '0';
538             else
539               if (M_AXI_RVALID = '1' and axi_rready = '1') then
540                 --The read_done should be associated with a read ready response
541                  read_done <= '1';
542               else
```

```
543                 read_done <= '0';
544               end if;
545             end if;
546           end if;
547         end process;
548 
549 
550       -- Register and hold any data mismatches, or read/write interface errors
551         process(M_AXI_ACLK)
552         begin
553           if (rising_edge (M_AXI_ACLK)) then
554             if (m_axi_areset = '0' or init_txn_pulse = '1') then
555               error_reg <= '0';
556             else
557               if (write_resp_error = '1' or read_resp_error = '1') then
558                 --Capture any error types
559                 error_reg <= '1';
560               end if;
561             end if;
562           end if;
563         end process;
564 
565 
566 
567 
568 
569       --    axi read/write process
570       -- set read_write
571             -- set axi_awaddr <=
572             -- set axi_wdata <=
573           --OR
574             -- set axi_araddr <=
575           --set init_axi_txn <= '1';
576 
577             -- wait for read_done = '1'
578           --OR
579             -- wait for write_done = '1'
580 
581           -- get results from  my_read_data if applicable
582 
583         -- process to configure the spi ip core and transfer data via spi.
584         process(M_AXI_ACLK)
585         begin
586           if (rising_edge (M_AXI_ACLK)) then
587             if (config_transfer = '0') then
588             if (m_axi_areset = '0' ) then
589                 config_state <= IDLE;
590             else
591                 case (config_state) is
592                     when IDLE =>
593                         if (config_start_pulse = '1') then             -- setup trigger
594                             config_state <= RESET_SPI;
595                         end if;
596                     when RESET_SPI =>
597                         if (config_state_wait = 0) then
```

```
598                             read_write <= '1';
599                             axi_awaddr <= x"00000040";
600                             axi_wdata <= x"0000000A";
601                             init_txn_pulse <= '1';
602                             config_state_wait <= 1;
603                         elsif (config_state_wait = 1) then
604                             init_txn_pulse <= '0';
605                             if (write_done = '1') then
606                                 config_state_wait <= 2;
607                             end if;
608                         elsif (config_state_wait = 2) then
609                             config_state_wait <= 3;
610                         elsif (config_state_wait = 3) then
611                             config_state <= ENABLE_TRANSMIT_INTERRUPT;
612                             config_state_wait <= 0;
613                         end if;
614                     when ENABLE_TRANSMIT_INTERRUPT =>
615                         if (config_state_wait = 0) then
616                             read_write <= '1';
617                             axi_awaddr <= x"00000028";
618                             axi_wdata <= x"00000004";
619                             init_txn_pulse <= '1';
620                             config_state_wait <= 1;
621                         elsif (config_state_wait = 1) then
622                             init_txn_pulse <= '0';
623                             if (write_done = '1') then
624                                 config_state_wait <= 2;
625                             end if;
626                         elsif (config_state_wait = 2) then
627                             config_state_wait <= 3;
628                         elsif (config_state_wait = 3) then
629                             config_state <= DISABLE_INTERRUPT;
630                             config_state_wait <= 0;
631                         end if;
632                     when DISABLE_INTERRUPT =>
633                         if (config_state_wait = 0) then
634                             read_write <= '1';
635                             axi_awaddr <= x"0000001C";
636                             axi_wdata <= x"00000000";
637                             init_txn_pulse <= '1';
638                             config_state_wait <= 1;
639                         elsif (config_state_wait = 1) then
640                             init_txn_pulse <= '0';
641                             if (write_done = '1') then
642                                 config_state_wait <= 2;
643                             end if;
644                         elsif (config_state_wait = 2) then
645                             config_state_wait <= 3;
646                         elsif (config_state_wait = 3) then
647                             config_state <= DESELECT_SLAVE;
648                             config_state_wait <= 0;
649                         end if;
650                     when DESELECT_SLAVE =>
651                         if (config_state_wait = 0) then
652                             read_write <= '1';
```

```
653                             axi_awaddr <= x"00000070";
654                             axi_wdata <= x"FFFFFFFF";
655                             init_txn_pulse <= '1';
656                             config_state_wait <= 1;
657                         elsif (config_state_wait = 1) then
658                             init_txn_pulse <= '0';
659                             if (write_done = '1') then
660                                 config_state_wait <= 2;
661                             end if;
662                         elsif (config_state_wait = 2) then
663                             config_state_wait <= 3;
664                         elsif (config_state_wait = 3) then
665                             config_state <= READ_STATUS_REG;
666                             config_state_wait <= 0;
667                         end if;
668                     when READ_STATUS_REG =>
669                         if (config_state_wait = 0) then
670                             read_write <= '0';
671                             axi_araddr <= x"00000060";
672                             init_txn_pulse <= '1';
673                             config_state_wait <= 1;
674                         elsif (config_state_wait = 1) then
675                             init_txn_pulse <= '0';
676                             if (read_done = '1') then
677                                 config_state_status_register <= my_read_data;
678                                 config_state_wait <= 2;
679                             end if;
680                         elsif (config_state_wait = 2) then
681                             config_state_wait <= 3;
682                         elsif (config_state_wait = 3) then
683                             config_state <= WRITE_STATUS_REG;
684                             config_state_wait <= 0;
685                         end if;
686                     when WRITE_STATUS_REG =>
687                         if (config_state_wait = 0) then
688                             read_write <= '1';
689                             axi_awaddr <= x"00000060";
690                             -- bits 3 and 4 describt clock phase and clock polarity. In this case they are
691                             axi_wdata <= (x"000000E6" OR config_state_status_register);
692                             init_txn_pulse <= '1';
693                             config_state_wait <= 1;
694                         elsif (config_state_wait = 1) then
695                             init_txn_pulse <= '0';
696                             if (write_done = '1') then
697                                 config_state_wait <= 2;
698                             end if;
699                         elsif (config_state_wait = 2) then
700                             config_state_wait <= 3;
701                         elsif (config_state_wait = 3) then
702                             config_done_pulse <= '1';
703                             config_state <= FINISH;
704                             config_state_wait <= 0;
705                         end if;
706                     when FINISH =>
707                         config_done_pulse <= '0';
```

```
                        config_state <= IDLE;
                    end case;
                end if;
            else
                if (m_axi_areset = '0' ) then
                    transfer_state <= IDLE;
                else
                    case (transfer_state ) is
                        when IDLE =>
                           if (transfer_start_pulse = '1') then
                               transfer_state <= SET_DATA_REGISTER;
                           end if;
                        when SET_DATA_REGISTER =>  -- set data register
                           if (transfer_state_wait = 0) then
                                read_write <= '1';
                                axi_awaddr <= x"00000068";
                                axi_wdata <= spi_send_data;
                                init_txn_pulse <= '1';
                                transfer_state_wait <= 1;
                            elsif (transfer_state_wait = 1) then
                                init_txn_pulse <= '0';
                                if (write_done = '1') then
                                    transfer_state_wait <= 2;
                                end if;
                            elsif (transfer_state_wait = 2) then
                                transfer_state_wait <= 3;
                            elsif (transfer_state_wait = 3) then
                                transfer_state <= SET_SLAVE_REGISTER;
                                transfer_state_wait <= 0;
                            end if;
                        when SET_SLAVE_REGISTER => --set slave select register
                          if (transfer_state_wait = 0) then
                                read_write <= '1';
                                axi_awaddr <= x"00000070";
                                axi_wdata <= x"FFFFFFFE";
                                init_txn_pulse <= '1';
                                transfer_state_wait <= 1;
                            elsif (transfer_state_wait = 1) then
                                init_txn_pulse <= '0';
                                if (write_done = '1') then
                                    transfer_state_wait <= 2;
                                end if;
                            elsif (transfer_state_wait = 2) then
                                transfer_state_wait <= 3;
                            elsif (transfer_state_wait = 3) then
                                transfer_state <= READ_CONTROL_REG;
                                transfer_state_wait <= 0;
                            end if;
                        when READ_CONTROL_REG => --read control reg
                           if (transfer_state_wait = 0) then
                                read_write <= '0';
                                axi_araddr <= x"00000060";
                                init_txn_pulse <= '1';
                                transfer_state_wait <= 1;
                            elsif (transfer_state_wait = 1) then
```

```
763                             init_txn_pulse <= '0';
764                             if (read_done = '1') then
765                                 transfer_temp_register  <= my_read_data;
766                                 transfer_state_wait <= 2;
767                             end if;
768                         elsif (transfer_state_wait = 2) then
769                             transfer_state_wait <= 3;
770                         elsif (transfer_state_wait = 3) then
771                             transfer_state <= WRITE_CONTROL_REG;
772                             transfer_state_wait <= 0;
773                         end if;
774                     when WRITE_CONTROL_REG => --write control reg
775                       if (transfer_state_wait = 0) then
776                             read_write <= '1';
777                             axi_awaddr <= x"00000060";
778                             axi_wdata <= (x"FFFFFEFF" and transfer_temp_register);
779                             init_txn_pulse <= '1';
780                             transfer_state_wait <= 1;
781                         elsif (transfer_state_wait = 1) then
782                             init_txn_pulse <= '0';
783                             if (write_done = '1') then
784                                 transfer_state_wait <= 2;
785                             end if;
786                         elsif (transfer_state_wait = 2) then
787                             transfer_state_wait <= 3;
788                         elsif (transfer_state_wait = 3) then
789                             transfer_state <= READ_STATUS_REG_WHILE;
790                             transfer_state_wait <= 0;
791                         end if;
792                     when READ_STATUS_REG_WHILE => --read status reg
793                         if (transfer_state_wait = 0) then
794                             read_write <= '0';
795                             axi_araddr <= x"00000064";
796                             init_txn_pulse <= '1';
797                             transfer_state_wait <= 1;
798                         elsif (transfer_state_wait = 1) then
799                             init_txn_pulse <= '0';
800                             if (read_done = '1') then
801                                 transfer_temp_register  <= my_read_data;
802                                 transfer_state_wait <= 2;
803                             end if;
804                         elsif (transfer_state_wait = 2) then
805                             transfer_state_wait <= 3;
806                         elsif (transfer_state_wait = 3) then
807                             if ((transfer_temp_register and x"00000004") = x"00000000" ) then
808                                 transfer_state <= READ_STATUS_REG_WHILE;
809                             else
810                                 transfer_state <= DESELECT_SLAVE;
811                             transfer_state_wait <= 0;
812                             end if;
813                         end if;
814                     when DESELECT_SLAVE => --deselect Slave
815                       if (transfer_state_wait = 0) then
816                             read_write <= '1';
817                             axi_awaddr <= x"00000070";
```

```
818                             axi_wdata <= x"FFFFFFFF";
819                             init_txn_pulse <= '1';
820                             transfer_state_wait <= 1;
821                         elsif (transfer_state_wait = 1) then
822                             init_txn_pulse <= '0';
823                             if (write_done = '1') then
824                                 transfer_state_wait <= 2;
825                             end if;
826                         elsif (transfer_state_wait = 2) then
827                             transfer_state_wait <= 3;
828                         elsif (transfer_state_wait = 3) then
829                             transfer_state <= READ_DATA;
830                             transfer_state_wait <= 0;
831                         end if;
832                     when READ_DATA =>
833                         if (transfer_state_wait = 0) then
834                             read_write <= '0';
835                             axi_araddr <= x"0000006C";
836                             init_txn_pulse <= '1';
837                             transfer_state_wait <= 1;
838                         elsif (transfer_state_wait = 1) then
839                             init_txn_pulse <= '0';
840                             if (read_done = '1') then
841                                 spi_receive_data  <= my_read_data;
842                                 transfer_state_wait <= 2;
843                             end if;
844                         elsif (transfer_state_wait = 2) then
845                             transfer_state_wait <= 3;
846                         elsif (transfer_state_wait = 3) then
847                             transfer_state <= FINISH;
848                             transfer_state_wait <= 0;
849                             transfer_done_pulse  <= '1';
850                         end if;
851                     when FINISH =>
852                         transfer_state <= IDLE;
853                         transfer_done_pulse  <= '0';
854                 end case;
855             end if;
856           end if;
857           end if;
858         end process;
859
860
861
862
863         process(M_AXI_ACLK)
864         begin
865           if (rising_edge (M_AXI_ACLK)) then
866             if (m_axi_areset = '0' ) then
867                 spi_control_state <= IDLE;
868             else
869             case (spi_control_state) is
870                 when IDLE =>
871                     spi_control_state <= START_CONFIG;
872                 when START_CONFIG =>
```

```
873                         config_transfer <= '0';
874                         config_start_pulse <= '1';
875                         spi_control_state <= WAIT_FOR_CONFIG;
876                     when WAIT_FOR_CONFIG =>
877                         config_start_pulse <= '0';
878                         if (config_done_pulse  = '1') then
879                            spi_control_state <= START_TRANSFER;
880                         end if;
881                     when START_TRANSFER =>
882                         config_transfer <= '1';
883                         spi_send_data <= x"0000001A";
884                         transfer_start_pulse  <= '1';
885                         spi_control_state <= WAIT_FOR_TRANSFER;
886                     when WAIT_FOR_TRANSFER =>
887                         transfer_start_pulse <= '0';
888                         if (transfer_done_pulse   = '1') then
889                            spi_control_state <= FINISH;
890                         end if;
891                     when FINISH =>
892                         spi_control_state <= IDLE;
893                 end case;
894             end if;
895             end if;
896           end process;
897 
898         -- User logic ends
899 
900 end implementation;
901 
```

Listing 14: Code for "Deflection_Estimator"

Appendix B

Filter Coefficients

No.	low pass	high pass
1	-0.006671	0.07239
2	-0.007837	0.02015
3	0.009766	0.02329
4	0.001364	0.027
5	-0.01856	0.03155
6	0.01703	0.03713
7	0.01411	0.04429
8	-0.04453	0.0539
9	0.0241	0.06742
10	0.05432	0.08835
11	-0.1155	0.1254
12	0.02834	0.2111
13	0.5352	0.6362
14	0.5352	-0.6362
15	0.02834	-0.2111
16	-0.1155	-0.1254
17	0.05432	-0.08835
18	0.0241	-0.06742
19	-0.04453	-0.0539
20	0.01411	-0.04429
21	0.01703	-0.03713
22	-0.01856	-0.03155
23	0.001364	-0.027
24	0.009766	-0.02329
25	-0.007837	-0.02015
26	-0.006671	-0.07239

Table B.1: Coefficients of filters rounded to 4th significant decimal places

Appendix C

ADAU1761 configuration

Register Adress	Register	Bit Name	Setting	Resulting Hex Data
4000	R0	CLKSRC INFREQ[1:0] COREN	0 = direct from MCLK pin 00 = 256 × sampling frequency 1 = core clock enabled	01
4015	R15	SPSRS LRMOD BPOL LRPOL MS	0 = Serial port sampling rate set in Register R17 0 = LRCLK mode 50% duty cycle 0 = BCLK polarity falling edge 0 = LRCLK polarity falling edge 1 = Serial data port master mode	01
4016	R16	BPF[2:0] MSBP LRDEL[1:0]	000 = 64 bit clock cycles per LRCLK audio frame 0 = MSB first in LRCLK frame 00 = Data delay from LRCLK edge 1 BCLK	00
4017	R17	ADOSR CONVSR[2:0]	0 = ADC oversampling ratio is 128x 000 = Converter sampling rate (DAC and ACD) 48kHz	00
401c	R22	MX3RM MX3LM MX3AUXG[3:0] MX3EN	0 = Mixer 3 right DAC input muted 1 = Mixer 3 left DAC input unmuted 0110 = Mixer input gain 0dB 1 = Mixer 3 enable	2d
401d	R23	MX3G2[3:0] MX3G1[3:0]	0000 = Mixer 2 bypasses gain control to Mixer 3 mute 0000 = Mixer 1 bypasses gain control to Mixer 3 mute	00
401e	R24	MX4RM MX4LM MX4AUXG[3:0] MX4EN	1 = Mixer 4 right DAC input muted 0 = Mixer 4 left DAC input unmuted 0110 = Mixer input gain 0dB 1 = Mixer 4 enable	4d
401f	R25	MX4G2[3:0] MX4G1[3:0]	0000 = Mixer 2 bypasses gain control to Mixer 4 mute 0000 = Mixer 1 bypasses gain control to Mixer 4 mute	00
4023	R29	LHPVOL[5:0] LHPM	111001 = Left Headphone output 0dB 1 = Left Headphone unmuted	e7

		HPEN	1 = Left Headphone volume control enabled	
4024	R30	RHPVOL[5:0] RHPM HPMODE	111001 = Right Headphone output 0dB 1 = Right Headphone unmuted 1 = enable headphone output	e7
4029	R35	HPBIAS[1:0] DACBIAS[1:0] PBIAS[1:0] PREN PLEN	00 = Headphone normal operation 00 = DAC normal operation 00 = Playback path normal operation 1 = Playback right channel enable 1 = Playback left channel enable	03
402a	R36	DACMONO[1:0] DACPOL DEMPH DACEN[1:0]	00 = DAC Stereo mode 0 = DAC polarity normal 0 = DAC de-emphasis filter disabled 11 = BOTH DAC enabled	03
402b	R37	LDAVOL[7:0]	00000000 = Left Channel Volume 0dB	00
402c	R28	RDAVOL[7:0]	00000000 = Right Channel Volume 0dB	00
40f2	R58	SINRT[3:0]	0001 = Serial input to DACs [L0, R0] ->[L, R]	01
40f9	R65	SLEWPD ALCPD DECPD SOUTPD INTPD SINPD SPPD	1 = Enable Codec slew cock 1 = enable ALC clock 1 = enable decimator resync clock 1 = enable serial routing output clocks 1 = enable interpolator resync clock 1 = enable serial routing intput clock 1 = enable serial port clock	7f
40fa	R65	CLK1 CLK0	1 = enable clock generator 1 1 = enable clock generator 0	03
40f4	R60	LRGP3 BGP2 SDIGP0	1 = port is LRCLK 1 = port is BCLK 1 = port is DAC_SDATA	00
40f8	R64	SPSR[2:0]	000 = serial port sampling rate is sampling rate	00
402d	R39	DACSDP[1:0] LRCLKP[1:0] BCLKP[1:0]	10 = no pullup or pulldown on DAC_SDATA 10 = no pullup or pulldown on LRCLK 10 = no pullup or pulldown on BCLK	aa
402f	R40	CDATP[1:0] CLCHP[1:0] SCLP[1:0] SDAP[1:0]	10 = no pullup or pulldown on CDATA 10 = no pullup or pulldown on CLATCH 10 = no pullup or pulldown on CCLK 10 = no pullup or pulldown on COUT	aa

Appendix D

Block Designs

D.1 Digital Controller Design

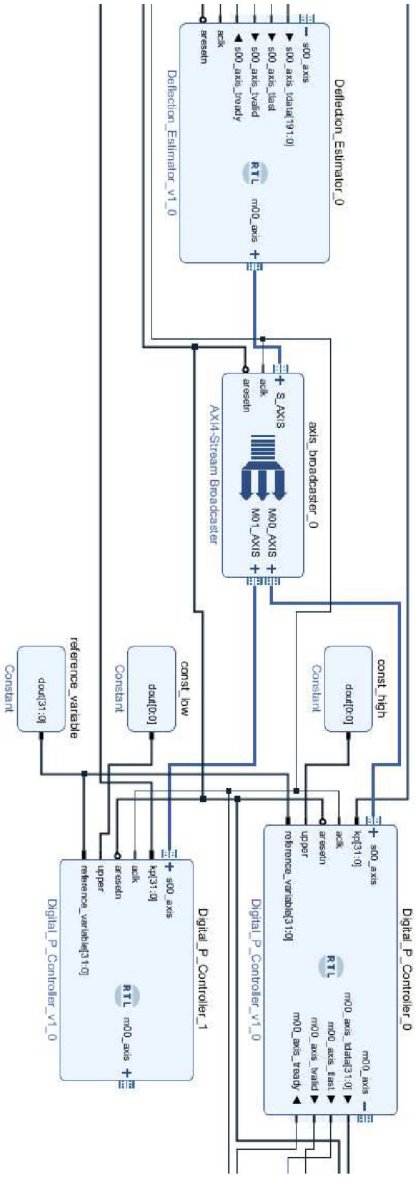

Figure D.1: Complete block design for digital controller

D.2 Validation Data Design

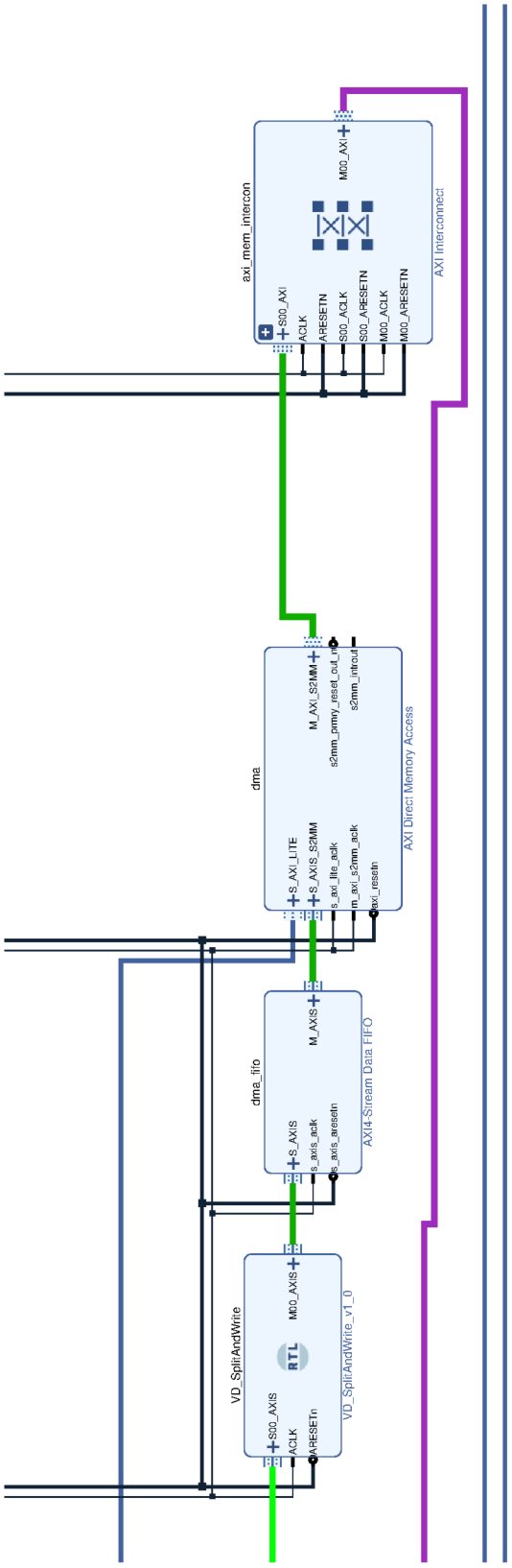

Figure D.2: The validation data block design

D.3 System Design

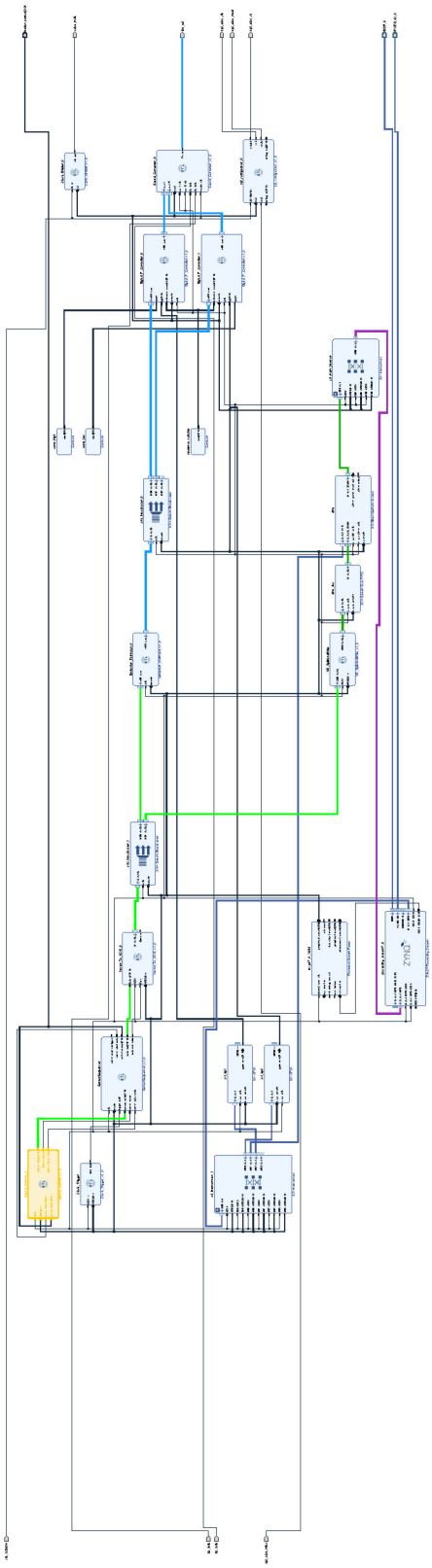

Figure D.3: The system main block design