Development and Characterization of Readout Electronics for a Monolithic Active Pixel Sensor

Matias Gjestvang Greaker

Thesis submitted for the degree of Master in Electronics and Computer technology
60 credits

Department of Physics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2019
Development and Characterization of Readout Electronics for a Monolithic Active Pixel Sensor

Matias Gjestvang Greaker
© 2019 Matias Gjestvang Greaker

Development and Characterization of Readout Electronics for a Monolithic Active Pixel Sensor

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo
Abstract

The ALPIDE is a radiation detector developed for the upgrade of the ALICE experiment at CERN. To be able to characterize this detector, and to do radiation measurements with it outside the ALICE environment, an easy to use readout system is needed. The work done in this thesis covers the design and development of a readout system for a single ALPIDE. In doing so various readout solutions have been discussed laying the foundation for specific readout system requirements. Particular emphasis was put on two VHDL modules which were developed to give a physical abstraction layer when interfacing with the ALPIDE. Both modules were implemented on an FPGA, and a complete readout system was realized. This readout system was successfully tested, and cluster size formation in the ALPIDE was studied using a radioactive source. The developed system had some shortcomings, notably handling the 320 Mb/s data transmitted from the ALPIDE. Here a continuous data transmission would result in a FIFO overflow.

Acknowledgments

I want to thank Ketil Røed, my supervisor, for helpful feedback and guidance. I would also like to thank Qasim W. Malik for aiding me with lab experiments, showing me around at CERN, and for many valuable discussions about the ALPIDE. Also, a big thanks to the people at ELAB for always answering my questions. Lastly, I like to thank Mads and Tonje for help with the writing.
Contents

1 Introduction ................................................. 7
  1.1 Objective ........................................ 9
  1.2 Thesis outline ................................ 9

2 Background: Radiation and Detectors .................. 11
  2.1 Radiation and definitions .......................... 11
  2.2 Interaction of radiation with matter .............. 12
    2.2.1 Heavy charged particles ................. 13
  2.3 Physics of semiconductor radiation detectors .... 16
    2.3.1 Semiconductor junction .................. 16
    2.3.2 Semiconductor junction detector ....... 19
  2.4 Semiconductor detectors .......................... 21
    2.4.1 Detectors for position measurement ....... 22

3 Background: ALPIDE ......................................... 25
  3.1 Background ........................................ 25
  3.2 Pixel Matrix ........................................ 26
    3.2.1 Pixel cell ................................ 26
    3.2.2 Priority Encoder ............................ 30
  3.3 Digital Periphery .................................... 30
    3.3.1 Taking a snapshot .......................... 30
    3.3.2 Operation modes ............................ 32
  3.4 Interface .......................................... 35
    3.4.1 Control Port ................................ 35
    3.4.2 Parallel Data Port ......................... 36
    3.4.3 Serial Data Port ............................ 38

4 Readout System Proposal ................................. 41
  4.1 Existing systems .................................... 41
    4.1.1 ALICE ITS .................................. 41
    4.1.2 ITS Prototype Readout Unit ............... 43
    4.1.3 DAQ board .................................. 45
  4.2 Requirements ....................................... 47
  4.3 Summary .............................................. 50

5 Readout System Implementation .......................... 51
  5.1 Overview ........................................... 51
  5.2 Hardware ............................................ 51
    5.2.1 ALPIDE Carrier ............................. 51
H Hardware-specific driver: Alpide Controller 109
  H.1 alpide_controller.h ....................................... 109
  H.2 alpide_controller.c ....................................... 111

I Hardware-specific driver: Alpide Data 115
  I.1 alpide_data.h ............................................... 115
  I.2 alpide_data.c ............................................... 116

J Alpide Controller Module VHDL 119
  J.1 Controller .................................................. 119
  J.2 Master ....................................................... 126

K Alpide Data Module VHDL 133
  K.1 Top .......................................................... 133
  K.2 Synchronizer ................................................ 139
  K.3 Filter ........................................................ 142
  K.4 Concatenate .................................................. 145
  K.5 FIFO .......................................................... 148

L Measurements from ALPIDE 153
Chapter 1

Introduction

In modern physics, the study of subatomic particles\(^1\) has become an increasingly popular field. Many of today’s biggest physics experiments, such as CERN, Fermilab, NSRL, all revolve around the study of the subatomic scale. The historic scientific success this field has had can explain its increasing popularity. Not only has the discoveries of subatomic particles led to scientific success in the field of particle physics, but it has accelerated discoveries in many other areas of science. E.g., the discovery of the electron, by J.J. Thomson in 1897, can be said to have laid the foundation of the modern revolution in computer technology \(^2[2]\).

As the understanding of subatomic particles is important for groundbreaking research, a good way to detect these particles become critical. Moreover, as physics experiments become more complex, demanding higher precision measurements, more complex and precise detectors are necessary. Often, the construction of new physics experiments demands that several technologies need to be pushed beyond their existing boundaries. For detectors, this typically means specialized detectors being more radiation-resistant and able to cope with more data at higher speeds \(^3[3]\). These specialized detectors usually also require specialized readout electronics, enabling measurements to be done successfully.

At Conseil Européen pour la Recherche Nucléaire (CERN) the study of elementary particles\(^2\) are being done by colliding subatomic particles together \(^3[3]\). Subatomic particles are accelerated close to the speed of light by the Large Hadron Collider (LHC) and are brought together at one of four collision points \(^3[3]\). These four collision points each have a detector: ATLAS, ALICE, CMS and LHC-B \(^5[5]–[8]\). Each detector is specifically designed to study different physics phenomenons. The A Large Ion Collider Experiment (ALICE) detector is focused on the study of quark-gluon plasma, a state of matter which existed just after the Big Bang \(^3[3]\).

To increase measurement precision in ALICE an upgrade of the detector is scheduled in 2019-2020 \(^6[6], [9]\). With this upgrade a new radiation detector called the ALICE Pixel Detector (ALPIDE) will be installed \(^10[10]\). This detector is developed specifically for the radiation environment at ALICE. And, specialized readout electronics are being developed specifically for the ALPIDE and the ALICE environment. The ALPIDE also offers interesting features for experiments done outside the ALICE detector. Such experiments, however, require suitable readout electronics to be developed for the ALPIDE.

---

\(^1\)Piece of matter/energy that is smaller than an atom \(^1[1]\).

\(^2\)The ultimate constituents of matter \(^4[4]\).
Figure 1.1: The LHC, a 27 km particle accelerator. The LHC and the four detectors (ATLAS, ALICE, CMS and LHC-B) are located 100 m underground [11].

Figure 1.2: A prototype of the ALPIDE, measuring 15 mm × 30 mm. Reprinted from [12], with permission from J. W. van Hoorne.
1.1 Objective

The main goal for this thesis is to explore the ALPIDE while contributing to future ALPIDE projects undertaken at the University of Oslo (UiO). In order to do so, this thesis’ chief objective will be to develop a readout system towards a single ALPIDE. The second-order objective will be to highlight general specification needs for readout electronics towards the ALPIDE. Together, these two objectives will highlight features of the ALPIDE and give insight into the requirements for readout electronics. This will provide a perspective on the development of readout systems towards the ALPIDE that will benefit future designs.

1.2 Thesis outline

In this thesis a readout system towards a single ALPIDE chip will be developed. The thesis will also discuss radiation measurements done with this readout system, demonstrating the readout system’s functionality. The thesis is organized as follows:

- **Chapter 2** provides a general theoretical background for radiation and semiconductor radiation detectors. The chapter also documents some of the latest technologies within semiconductor detectors.

- **Chapter 3** gives an explanation of the ALPIDE. The chapter focuses on aspects of the detector which are important for the development of readout electronics.

- **Chapter 4** proposes a readout system towards a single ALPIDE. The chapter also presents the discussion about the requirements needed for the readout system.

- **Chapter 5** documents the readout system developed in this thesis. Here the parts of the readout system are discussed.

- **Chapter 6** evaluates the developed readout system and the ALPIDE. The chapter also discuss the radiation measurements done with the readout system.

- **Chapter 7** summarizes the work done in this thesis, presenting my conclusion.
Chapter 2

Background: Radiation and Detectors

This chapter covers the theoretical backgrounds for this thesis. The chapter seeks to give the reader an introduction to the general physics of radiation detectors and evaluate the pros and cons of different radiation detector technologies. As the ALPIDE is mainly constructed of silicon, the chapter has limited the discussion to semiconductor radiation detectors [9].

The first two sections will review radiation and how radiation interacts with matter. The last two sections will discuss semiconductor radiation detectors. The main sources for this chapter are the following books: "Radiation Detection and Measurement" by Glenn F. Knoll [13], "Techniques for Nuclear and Particle Physics Experiments" by W.R. Leo [14], "Solid State Electronic Devices" by Ben. G. Streetman [15] and "Semiconductor Radiation Detectors" by G. Lutz [16].

2.1 Radiation and definitions

Radiation is defined as the “emission or propagation of energy in the form of waves or particles” [17]. According to Knoll [13], radiation can broadly be categorized into four categories:

- Fast electrons
- Heavy charged particles
- Electromagnetic radiation
- Neutrons

The divide above is a consequence of the different physical aspects each category has. That is, for example when a beta particle (fast electron) radiates into matter, as opposed to a neutron, only the beta particle experience a coulomb force with the substrate’s nuclei and electrons. Although the categories differ, many of the same concepts and rules also apply to all categories. An important measurement for all types of radiation is how much energy the radiating particle/wave gives off to the irradiated material. It is these concepts that will be in focus throughout this chapter.
2.2 INTERACTION OF RADIATION WITH MATTER

Energy
The measurement of radiation energy is electron volt (eV). One electron volt is defined as the kinetic energy that is gained by an electron accelerated in a potential of 1 V. That is [13]:

\[ 1 \text{ eV} = 1.602 \times 10^{-19} \text{ J}. \] \hspace{1cm} (2.1)

Typically in semiconductor radiation detectors, only energies above the ionization energy are of interest. The ionization energy is the minimum required energy to produce ionization in the material. Radiation with energy greater than the ionization energy classifies as ionizing radiation. Throughout this thesis, we will restrict our discussion to ionizing radiation. This will be further discussed in section 2.2 and section 2.3.

Radioactivity
Another measurement of interest is radioactivity. Radioactivity is a process were unstable nuclei disintegrates and forms new nuclei, spontaneously. Such disintegration usually results in the emission of radiation. The official unit of radioactivity is defined as the Becquerel (Bq), here [18]:

\[ 1 \text{ Bq} = 1 \text{ disintegration/s}. \] \hspace{1cm} (2.2)

An older unit of activity, and still used is the Curie (Ci). This is defined as [18]:

\[ 1 \text{ Ci} = 3.7 \times 10^{10} \text{ Bq} = 3.7 \times 10^{10} \text{ disintegration/s}. \] \hspace{1cm} (2.3)

Knowing the radioactivity becomes useful when estimating the number of particles radiating each second. For example, having an alpha source with radioactivity of 5 kBq would roughly correspond to 5000 alpha particles radiated from the source each second. Thus, if a detector were covering the whole surface area of the radiating source, we would expect to detect 5000 alpha particle each second. These kinds of calculations can be important for the configuration of a detector. This is further discussed in Chapter 3.

2.2 Interaction of radiation with matter

Understanding the interaction of radiation with matter is important for the understanding of radiation detectors. This understanding gives a better knowledge of the expected sensitivity for different detectors, and thus gives insight on how to construct/configure detectors. As radiation penetrates matter, it sees matter for its components, i.e., as a construct of electrons and nuclei. How radiation reacts with matter will depend on the radiation energy, radiation type and the type of material. These factors determine the expected main types of interactions. For example, taking the classical Rutherford experiment [19], an alpha particle entering a gold foil may collide electromagnetically with an electron, collide elastically with the gold nuclei, or even be absorbed in a nuclear reaction to produce secondary radiation. However, these interactions occur with a certain probability. E.g., in the Rutherford experiment, only 1 in 20,000 alpha particles underwent an elastic collision with the
2.2. INTERACTION OF RADIATION WITH MATTER

As described in the last section, radiation could broadly be categorized in four categories. Which expected interaction process depends on which category we find the radiation in. Generally, the most common interaction process for heavy charged particles and photons (electromagnetic radiation), is electromagnetic interaction, typically, inelastic collisions with the material electrons [14]. For uncharged radiation, e.g., neutrons, interactions involving the strong nuclear force dominates. Uncharged radiation collisions also involves processes of weak and electromagnetic interactions [14]. Nevertheless, in most practical cases of interest, the interactions result in a transfer of energy from the incident particle to the material’s nuclei or electrons [13].

In this thesis, the focus will be on heavy charged radiation. Though calculations differ for other types of radiation, the concept of calculating the energy deposited in the material stays the same. Moreover, as the energy of radiation also plays an important role in which of the interaction processes is involved, we will restrict the discussion throughout this thesis to energies from a few keV and higher. This range is typical for nuclear and particle physics [14]. At last, as will be discussed in section 2.3, the material involved in this thesis will be semiconductor materials. Typically silicon and germanium.

2.2.1 Heavy charged particles

Heavy charged particles are particles with a mass of one atomic mass unit or greater [13]. For heavy charged particle radiation, the two main effects characterizing the interaction with matter is:

- Inelastic collision with electrons.
- Elastic collision with the nuclei.

Usually, the quantity of interest is how much energy the traversing particle loses to the material. Here, inelastic collisions with electrons are almost solely the contributor to energy loss [14]. This is because in most materials the nucleus is much heavier than the incident particle. Thus very little energy is transferred from the incident particle to the substrate in an elastic collision with the nucleus.\(^1\)

In inelastic collisions, the amount of energy loss per collision is usually a small fraction of the particle’s kinetic energy. However, the number of collisions is large. Hence the cumulative energy loss in dense matter is substantial. Since the number of collisions is large, the variance of energy loss per path length is small. One can therefore meaningfully work with the average energy loss per path length. This is called the stopping power, or the rate of energy loss, and is given by the Bethe-Bloch formula [14]:

\[
-\frac{dE}{dx} = 2\pi N_a r_e^2 m_e c^2 \rho \frac{Z^2}{A} z^2 \left[ \ln \left( \frac{2m_e \gamma^2 v^2 W_{\text{max}}}{I^2} \right) - 2\beta^2 \right]
\]  

\(1\) More precise, 1 in 20,000 alpha particles underwent an elastic collision changing the incident alpha particle trajectory more than 90° [19].

\(2\) This will not be true for example with alpha particle radiating in hydrogen, here the mass of the nucleus is less than the alpha particle. However, throughout this thesis we are considering semiconductor materials, in which this assumption holds.
2.2. INTERACTION OF RADIATION WITH MATTER

Figure 2.1: Figure (a) shows the rate of energy loss (in keV µm⁻¹) as a function of particle kinetic energy. Figure (b) shows the maximum range (in µm) as a function of kinetic energy. Both figures are produced by theoretical calculations. The particles were passing through silicon. Reprinted from [20], with permission from ©Elsevier.

Here:
- \( r_e \): classical electron radius = \( 2.817 \times 10^{-13} \) cm;
- \( m_e \): electron mass;
- \( N_a \): Avogadro’s number = \( 6.022 \times 10^{23} \) mol⁻¹;
- \( I \): mean excitation potential;
- \( Z \): atomic number of absorbing material;
- \( A \): atomic weight of absorbing material;
- \( \rho \): density of absorbing material;
- \( z \): charge of incident particle in units of \( e \);
- \( \beta = \frac{v}{c} \) of the incident particle;
- \( \gamma = \frac{1}{\sqrt{1 - \beta^2}} \);
- \( W_{\text{max}} \): maximum energy transfer in a single collision.

Equation 2.4 is usually measured in eV g cm⁻³, or eV cm⁻¹ if the expression is divided by \( \rho \). The Bethe-Bloch formula usually also includes two correction parameters, called the density effect correction and the shell correction, these are important for high and low energies respectively [14]. For this thesis, however, no direct calculations with the Bethe-Bloch formula will be done, and the corrections are therefore omitted.

In Figure 2.1a, the average rate of energy loss is shown as a function of kinetic
2.2. INTERACTION OF RADIATION WITH MATTER

Figure 2.2: The Bragg curve, where the rate of energy loss is plotted with respect to particle depth. The data is from a simulation in SRIM [21]. Here 1000 alpha particles with an initial energy of 5.5 MeV radiating in silicon were simulated.

energy for various particles in silicon. In the figure the stopping power for charged particles in silicon was computed using equation 2.4 [20]. Using these computations, one can further calculate the average energy loss in a finite sample by integration. That is, a sample with a thickness $L$, will lose an average energy $\Delta E$ in the sample by the formula:

$$\Delta E = \int_0^L \frac{dE}{dx} \, dx$$  \hspace{1cm} (2.5)

Equation 2.5 is important to understand the sensitivity of radiation detectors, as the equation predicts how much energy will be transferred from the radiation to the detector material. Moreover, it gives a good indication of the detector’s sensitivity for a particular radiating particle.

Another prediction is the range of the incident particle. This is how far particles penetrate before losing all their energy. The range $R$ indicates how thick a substrate must be if we want to stop the radiation completely. Using the Bethe-Bloch formula, this can roughly be calculated as:

$$R = \int_0^{E_r} \left( \frac{dE}{dx} \right)^{-1} \, dE.$$  \hspace{1cm} (2.6)

Here $E_r$ represent the incoming particle’s energy. The range as a function of $E_r$ can be seen in Figure 2.1b for various particles in silicon. The calculations were done by [20]. For example, deduced from the graph in Figure 2.1b a 10 MeV proton will lose all of its energy in 700 $\mu$m of silicon.

**Bragg peak**

As can be seen from Figure 2.1a the stopping power increase as the kinetic energy of the particle decrease, which means that for a particle slowing down in matter,

---

3Roughly because the Bethe-Bloch formula is only valid in a certain energy range. Therefore, when integrating from zero, the formula is not valid. Also, the particle doesn’t take a straight path, but a zigzag path through the substrate. However, the main idea is the same, and the corrections are out of scope for this thesis.

15
the particle will deposit more of its energy at the end of its path rather than at its beginning. This is shown in Figure 2.2, here the stopping power of an alpha particle is plotted versus penetration depth. The data is from a simulation done in SRIM [21]. The spike in Figure 2.2 is called the Bragg peak and is where the alpha particle deposits its remaining energy.

**Ionization, direct or indirect**

Energy transferred from the incident particle to the material can cause ionization or excitation. In semiconductor detectors, ionization is of interest, as this will create electron-hole pairs. It should, however, be noted that ionization can further be categorized into two categories: direct- or indirect ionization. For direct ionization, the incoming particle collides with an orbiting electron causing ionization. Whereas for indirect ionization, enough energy is transferred to the electron such that the electron itself can cause secondary ionizations. For the latter, the creation of electron-hole pairs will not only happen in the incident particle trajectory but also at the periphery of the trajectory. This effect can be significant when one wants an exact position measurement of the incident particle. As a shift of ionization for the incident particle trajectory might lower the spatial resolution.

2.3 Physics of semiconductor radiation detectors

Semiconductor radiation detectors are based on crystalline substrates usually of silicon or germanium [14]. The basic principle of a semiconductor detector is to collect electron-hole pairs caused by ionizing radiation. These electron-hole pairs are collected by an electric field which generates a voltage change which in turn can be measured. In this section, a short introduction on the physical principles governing a semiconductor detector will be explained. The section will be simplified and not focus on real-life implementation.

2.3.1 Semiconductor junction

All semiconductor detectors depend on a semiconductor junction, more commonly called a diode [14]. This junction form when two regions of semiconductor material with different doping concentrations are neighboring.

**Barrier potential**

Take for example a Si substrate with a p-type region and an n-type region as shown in Figure 2.3. In this configuration, an internal barrier potential $V_0$ will form. Given the doping concentration $N_a$ of the p-type and the doping concentration $N_d$ of the n-type region, the potential can be calculated by the following equation [15]:

$$V_0 = \frac{kT}{q} \ln \frac{N_a N_d}{n_i^2}, \quad (2.7)$$

where $k$ is the Boltzmann’s constant, $T$ is the temperature, and $n_i$ is the intrinsic concentration of electrons in the substrate.

This potential is formed due to the large carrier concentration gradient at the
2.3. PHYSICS OF SEMICONDUCTOR RADIATION DETECTORS

Figure 2.3: As p-type material and n-type material are joined a depletion region is formed, this region give rise to an electric field $\mathcal{E}$. The p-type region is doped with acceptors, and the n-type region is doped with donors, per definition. In the figure only the doped atoms are shown as an illustration [22].

junction. That is, electrons from the n-type region will diffuse into the p-type region and holes will diffuse from p to n. As electrons diffusion from n to p they will recombine with holes, this recombination will leave behind uncompensated donor ions and acceptor ions. Consequently, an internal electric field $\mathcal{E}$ will build up. The electric field will be in the direction n to p, see Figure 2.3, and induce a drift current opposite to that of the diffusion current. Equilibrium is reached when the drift current equals that of the diffusion current. At this equilibrium the electric field is constant, and because of this field, there is an internal barrier potential $V_0$.

Depletion region

The region with uncompensated donors and acceptors is called the depletion region, see Figure 2.3. Assuming a uniform doping concentration and an abrupt junction, the electric field $\mathcal{E}$ will only exist inside this region.\(^4\) This region width, $W$, will depend on the concentration of the n and p dopant and on the applied voltage $V$ over the junction. The width can be expressed by the following formula [15]:

\[
W = \left[\frac{2eV_j}{q}\left(\frac{N_a + N_d}{N_aN_d}\right)\right]^{1/2},
\]

(2.8)

here $V_j = V_0 - V$.

\(^4\)Throughout this chapter a uniform doping concentration will be assumed, as well as an abrupt junction profile. It is, however, worth noting that this is not the case in real-life implementations. Here both the doping concentration and the junction exhibits a certain profile.
Figure 2.4: A pn-junction with a reverse voltage $V$ applied. $W$ is the depletion width when $V < 0$ V and $W_0$ is the width when $V = 0$ V. As can be seen, $W$ increases with an applied reversed-bias voltage.

The depletion region permittivity is $\epsilon$, and $q$ is the electronic charge. Illustrated in Figure 2.4 is a pn-junction with reverse-bias voltage i.e., a negative voltage to the p-side. This reverse voltage $V$, is by definition negative with respect to equation 2.8, and thus gives an increase in $V_j$ subsequently increasing $W$. That is, by applying a negative voltage to the p-side of the pn-junction, the depletion region will increase. As we will see, this is an important feature for semiconductor detectors as it enables the control of the depletion width.

It is worth noting how equation 2.8 behave when the doping concentration on one side is heavily doped compared to the other side. If $N_d >> N_a$ equation 2.8 becomes:

$$W \approx \left[\frac{2\epsilon V_j}{qN_a}\right]^{1/2}. \tag{2.9}$$

The depletion width will now solely depend on the acceptor doping concentration as well as the reverse bias voltage. Moreover, since the electric field inside the depletion region requires an equal amount of charge on both the p side and n side, the depletion region will primarily extend into the p side of the junction. This can qualitatively be understood by remarking that electrons must go further into the p region to "discover" a hole for recombination since the concentration of holes is much lower than that of electrons. Such a junction is referred to as a n$^+$p-junction and is common in semiconductor detectors [23]–[25]. Actually, the ALPIDE (our lead "role" in this thesis) has a n$^+$p-junction [12].

**Junction capacitance**

Before explaining how a pn-junction is used for radiation detection, an essential property of the pn-junction should briefly be noted. The capacitance $C$ of a pn-junction is analogous to that of a parallel plate capacitor [15]. That is, the pn-junction capacitance can be expressed as follow:

$$C = \epsilon \frac{A}{W}, \tag{2.10}$$

where $A$ is the junction cross-section and $W$ is the depletion width given by equation 2.8 or equation 2.9.
2.3. PHYSICS OF SEMICONDUCTOR RADIATION DETECTORS

Figure 2.5: A pn-junction working as a radiation detector. Radiation with enough energy will create electron-hole pairs. The electron-hole pairs created in the depletion region will be affected by the electric field $\mathcal{E}$, and consequently contribute to a charge collection on each side of the pn-junction.

### 2.3.2 Semiconductor junction detector

The semiconductor junction explained in the last section can be used to detect radiation. When a particle passes through the depletion region of a pn-junction the overall effect will be the creation of electron-hole pairs. These electron-hole pairs will function as carriers inside the depletion region. Electron-hole pairs inside the depletion region will be swept by the electric field $\mathcal{E}$ to each side of the junction, as shown in Figure 2.5. This will result in a charge build up on each side of the junction, which in turn can be measured.

#### Ionization energy

As discussed in section 2.2.1 the production process of electron-hole pairs can be direct or indirect. Regardless the overall effect is the generation of electron-hole pairs and the quantity of interest is the ionization energy, $w$. This quantity is the average energy expended by the incoming particle to generate one electron-hole pair. Experimental results have shown this quantity to be largely independent of the incident radiation energy [13]. Table 2.1 shows ionization energies for silicon and germanium. Here the results are obtained using 5.3 MeV alpha particles as incident radiation [26]. For further discussion, we will also assume that the ionization energy is independent of particle types. This shows experimentally to be partly true for other light ions, electrons and protons [27]–[30]. However, it should be noted that a large difference in $w$ is measured for heavy ions or fission fragments [13].

<table>
<thead>
<tr>
<th></th>
<th>Si</th>
<th>Ge</th>
</tr>
</thead>
<tbody>
<tr>
<td>300 K</td>
<td>3.62 eV</td>
<td></td>
</tr>
<tr>
<td>77 K</td>
<td>3.76 eV</td>
<td>2.96 eV</td>
</tr>
</tbody>
</table>

Table 2.1: Average ionization energy for Si and Ge [26].

#### Charge collection

Given the assumption that $w$ is constant for a given substrate, and assuming that the depletion region is thick enough to stop all particles completely. The number of electron-hole pairs created can be calculated as $E_r/w$, here $E_r$ is the energy of the radiating particles. Introducing also a collection efficiency $n$, the charge $Q$ collected
on each side of the pn-junction can be calculated by:

\[ Q = n \frac{E_r}{w}. \]  

This together with equation 2.10 gives an observed voltage \( V \) over the pn-junction:

\[ V = \frac{Q}{C} = n \frac{E_r}{wC} = \frac{nW}{w\epsilon A} E_r. \]  

The voltage generated by equation 2.12 is directly proportional to the radiation energy \( E_r \).\(^5\) Giving a good way to measure radiation energy by a voltage change.

If the depletion region is not thick enough to completely stop all particles, the radiation energy \( E_r \) cannot be used to calculate the number of electron-hole pairs. Instead the energy loss \( \Delta E \), given by equation 2.5, must be used. Such calculations give insights to the thickness needed to obtain a certain voltage.

**Recombination**

The charge collection efficiency \( n \) represents the ratio of collected charges over created charges. A lower \( n \) will lower the voltage response, as can be seen from equation 2.12, and in turn, affect the sensitivity of the radiation detector. Incomplete collection \((n < 1)\) occurs in particular when the particle trajectory is longer than, or outside, the depletion region. This is illustrated in Figure 2.6. Electron-hole pairs created outside the depletion region will not experience the electric field, and their movement is exclusively due to diffusion. Excess carriers outside the depletion region will recombine after a characteristic lifetime. These carriers, therefore, have the probability of not contributing to charge collection.

As an example, if an electron-hole pair is created a distance \( x \) from the depletion region in a silicon pn-junction. The probability of the electron diffusing to the depletion region is given by \([15]\):\(^6\)

\[ P(x) = e^{-x/L_n}. \]  

Here \( L_n \) is the diffusion length for electrons in silicon. \( L_n \) is the average distance an electron diffuses before recombining. Electron-hole pair created, e.g. a distance \( x = L_n \) from the depletion region will only have a 37% chance of contributing to the charge collection. Hence reducing the charge collection efficiency \( n \) and therefore the voltage response. Thus the sensitivity of the radiation detector is reduced as a consequence of the creation of electron-hole pairs outside the depletion region.

For this reason, it becomes apparent that a bigger depletion region can be favorable to improve the sensitivity of the semiconductor detector. This can be achieved by applying a reversed bias voltage as discussed in section 2.3.1, and has been shown to significantly increase the charge collection efficiency for the ALPIDE \([31]\), \([32]\).

\(^5\)Assuming that the efficiency \( n \) is independent of the energy \( E_r \).

\(^6\)Here only motion orthogonal to the depletion region is considered. If motion in three dimensions were taken into account, the diffusion length \( L_n \) in the direction of the depletion region would be reduced and thereby also reducing the collection efficiency.
2.4 Semiconductor detectors

As discussed in the book “Semiconductor Radiation Detectors” by Gerhard Lutz [16], semiconductor materials have some unique properties compared to other materials that make them suitable for detection of ionizing radiation. To illustrate the uniqueness a comparison to gaseous detectors is listed in the bullet points below. This comparison will act as a short introduction for this section. Here later in the section, some specific semiconductor detectors will be discussed. The comparison is partially quoted by the comparison in [16, page 79]. The comparison is made for silicon detectors.

- Higher sensitivity. The low ionization energy for silicon detectors ($\approx 3.6\text{ eV}$) compared to gaseous detectors ($\approx 30\text{ eV}$) causes a large number of charge carriers per unit energy expended by the incident particle.

- Higher resolution. The high density ($\rho = 2.33\text{ g cm}^{-3}$) leads to more energy loss per traversed length of the incident particle. This can be seen from equation 2.4, here $\frac{dE}{dx} \propto \rho$. This property makes it possible to build thin detectors with good sensitivity. Further, the large energy loss per traversed length also minimizes the length of secondary ionization. Thus making the position measurement more precise.

- Faster response time. Carriers in silicon can move freely. This makes for a fast charge collection and detectors can be used in high-rate environments. This is actually one of the reasons the ALPIDE chip will be utilized at ALICE [33].

- New types of detectors. Semiconductor detectors have the possibility of creating complex field configurations by doping. Allowing for completely new types of detectors with no gaseous analog.

Figure 2.6: Illustration of drift and diffusion. Only the electron-hole pairs inside the depletion region experience drift. Only a fraction of the pairs created outside the depletion will contribute to charge collection.
2.4. SEMICONDUCTOR DETECTORS

Better integration. As the detector and the readout electronics can be built from the same material, their integration into a single device is possible.

Ease of production. Silicon is highly available due to its popularity in integrated circuits (ICs) manufacturing. The availability reduces the price, and the use of already established process technologies make the production of detectors easier.

As the detector discussed in this thesis, the ALPIDE chip, is mainly a position detector, this section will only discuss semiconductor detectors for position measurements. That is detectors measuring the spatial position of the particle’s trajectory.

2.4.1 Detectors for position measurement

By splitting a detector in multiple smaller detectors, a signal due to an incident particle will correspond to a spatial position. This section will look at some typical position-sensitive detectors.

Diode strip detectors

A diode strip (also called microstrip) detector divides the detector into strips. The position of the incident particle is given by the location of the strip showing the signal. Figure 2.7a shows an illustration of a strip detector. This figure is taken from [34], here a strip detector with a spatial resolution of 5 µm was demonstrated. This demonstration showed that strip detectors, and in general semiconductor detectors, were promising detectors for high-energy particle physics [14], [34].

As electron and holes are swept to opposite sides of the wafer, it is possible to use both charge carriers for position measurements. Such a detector is called a double-strip detector. This detector, seen in Figure 2.7b, have strips on both sides of the wafer, orientated orthogonal to each other to give both x and y coordinates for the incident particle.

Silicon strip detectors have been adopted in the tracking of high-energy particles and as well in X-ray imaging [36]. E.g., strip detectors of the design produced by the Center of Industrial Research in Oslo [37], were used, with very satisfying results,
in the DELPHI\textsuperscript{7} experiment at the electron-positron collider (LEP) at CERN [16], [40].

**Pixel detectors**

A pixel detector consists of individual diodes laid out in a grid. This approach reduces the active detector surface area compared to diode strip detectors, and hence reduces capacitance. The reduction in capacitance is illustrated by equation 2.10. As a consequence, the intrinsic noise is reduced [41]. Moreover, the reduction in noise also makes the detector more radiation hard. The detector becomes more radiation hard as a consequence of the good signal to noise ratio, which in turn enables a thinner detector construction. These are among several advantages for pixel detectors mentioned in [41].

The main disadvantage for pixel detectors is the need for individual electrical connections to each pixel. A common solution to this problem is to separate the readout chip and pixel detector and connect them with flip-chip solder bonding. This is called a *hybrid pixel sensor* and a cross-section of one pixel is seen in Figure 2.8a. Another solution is to integrate the readout electronics with the detector. This is called a *monolithic pixel sensor*. Such a solution give rise to several construction problems. For instance, usually lower resistivity silicon is used in ICs manufacturing than for silicon detectors [13], or, the in-pixel circuitry might function as a charge collection electrode [42]. In Figure 2.8b a monolithic pixel sensor is illustrated. This is actually an illustration of the ALPIDE design. Here one can see deep p-wells being utilized to shield the readout PMOS transistor from the sensor element, thus not making it collect charge from the incoming radiation.

Although the construction of a monolithic pixel sensor presents many problems, the integration of detector and its electronics further increases the noise performance [16]. These kinds of detectors have thus gained high popularity and are therefore the chosen design when the ALICE experiment at CERN upgrades in 2019-2020 [33]. This design is named the ALPIDE design and will be the focus of the next chapter.

\textsuperscript{7}DELPHI stands for "Detector with Lepton, Photon and Hadron Identification" and was a detector at CERN operating between the year 1989 and 2000 [38], [39].
Figure 2.8: Figure (a) shows the cross-section of a hybrid pixel sensor. An incident particle will generate electron-hole pairs which in turn will generate a charge collection on the diode implant. This can then be read out by the readout chip. Figure (b) shows the cross-section of a monolithic pixel sensor. Here, the sensor diode and the readout circuit are integrated into one piece of silicon. Reprinted from [12], with permission from J. W. van Hoorne.
Chapter 3

Background: ALPIDE

This chapter aims to give the reader a qualitative understanding of the ALICE Pixel Detector (ALPIDE). The chapter will go into depths about parts of the sensor that are relevant for this thesis. These parts are in particular the readout functionality, that is its interface to off-detector electronics, and the front-end pixel circuitry.

This chapter is split into four main sections. In the first section a short background for why the ALPIDE was developed and for which purpose will be given. The sections Pixel Matrix, Digital Periphery and Interface will then follow. The order of these three sections is intended so that the reader will get an introduction to the ALPIDE chip from the bottom-up. That is, the section Pixel Matrix will first go through how a particle interacts with the ALPIDE, and how this interaction is converted to an electrical signal which is stored digitally. The section Digital Periphery will describe how that digital signal gets handled, and how different configurations of the ALPIDE chip will affect the handling of this signal. At last, the section Interface will go into detail about how data is transmitted from and to the ALPIDE chip.

The last section, Interface, is particular important for the development of the readout system done in this thesis. Almost all of the content in this chapter is taken from the Alpide Operations Manual [43], unless otherwise noted.

3.1 Background

The ALPIDE is a new sensor designed for the upgrade of ALICE experiment. ALICE is one of the detectors in the LHC ring at CERN. Here ALICE is designed to study the physics of strongly interacting matter, using proton-proton, proton-nucleus and nucleus-nucleus collisions [44]. The upgrade of ALICE includes an upgrade of its Inner Tracking System (ITS), which is situated close to the collision point. This upgrade is scheduled to take place during the long shutdown of the LHC in 2019-2020 [9] and will use the ALPIDE design for the new ITS. The ALPIDE chip is a Monolithic Active Pixel Sensors (MAPS) and is fabricated in 180 nm TowerJazz’s CMOS Image Sensor technology [9], [45].

The new Inner Tracking System (ITS), a 10 $\text{m}^2$ silicon tracking detector corresponding to more than twelve billion pixels, consist of seven layers [46]. The ITS is shown in Figure 3.1. The three innermost layers are called Inner Barrel layers, and the four outer layers are called Outer Barrel. As the innermost layers can expect a higher particle hit rate the ALPIDE chips in the Inner Barrel are configured slightly
3.2 Pixel Matrix

The ALPIDE chip consist of $1024 \times 512$ pixel cells each measuring at $29.24 \times 26.88 \text{µm}$ [43]. The pixels are grouped two and two in columns with a circuit called Priority Encoder, see Figure 3.2. The main functionality of this circuit is to distribute signals from and to the pixels, and as well ensure that only the pixels containing a hit will be read out. Thereby the name “priority”. The Pixel Matrix can be split into two parts, the pixel cell, and the priority encoder.

3.2.1 Pixel cell

Each pixel cell contain a collection diode, a front-end amplifier, a discriminator and a memory, see Figure 3.2. The collection diode is a $n^+p$-junction which is depict in Figure 3.4, here the p of the junction is the Epitaxial Layer $P$- and the n is the NWELL DIODE. The white layer in Figure 3.4 is the depletion region. The depletion region can be controlled by the substrate bias, as discussed in Chapter 2, to increase the charge collection [31]. This is done by setting a negative voltage to the Epitaxial Layer in Figure 3.4, more on this below. As can be seen from

![Figure 3.1: The seven layers of ALICE new Inner Tracking System (ITS). Each tile represents one ALPIDE chip. [10, Figure 1.1).](image)
3.2. PIXEL MATRIX

The transistors in Figure 3.4 each pixel hosts in-pixel circuitry, this is the circuitry depicted in Figure 3.3. These transistors are protected by deep p-wells to protect e.g. the PMOS transistor to function as a charge collection diode [31].

Signal of an incident particle

Following the signal of an incident particle a hit will be stored in the following manner (best read with Figure 3.3):

1. The incident particle will generate electron-hole pairs, as shown in Figure 3.4. Electrons entering the depletion region will be collected at the NWELL DIODE.

2. The collection of electrons will lower the potential at the pixel input, PIX_IN. This is visualized in the first graph in Figure 3.3. The PIX_IN input is continuously reset by $V_{RST}$, see Figure 3.4, so the voltage drop will only be temporary.

3. The voltage drop at PIX_IN will be inverted and amplified. Seen in the second graph in Figure 3.3. While this amplification is above a predefined threshold, signal OUT_D is high. The signal OUT_D is digital and indicates that a hit has been detected on the respective pixel cell.

4. The storage of hit data to in pixel memory will only be stored if the STROBE signal is high. How this is set high will be discussed in section 3.3.

---

*Zero suppression is the removal of unnecessary zeros. In our case, non pixel hits [47].*
5. If STROBE is high, a digital ‘1’ is written to one of the three “Hit Storage Latches” in Figure 3.3. These latches are called Multi Event Buffer (MEB) and functions as a circular buffer. They allow three storages of pixel hits without any extra readout or data loss. More on the MEB below.

Some important properties of the pixel cell should be emphasized. The threshold value, which decides if OUT_D goes high is a global value. That is, all pixels have a common threshold value. Further, when a STROBE signal is asserted it is asserted globally to all pixels, and this is referred to as a frame or a snapshot. This means that the ALPIDE operates in global shutter mode. That is, all the pixel hits are stored simultaneously. This stands in contrast to detectors operating in rolling shutter mode, here for example columns are captured one by one. An advantage with rolling shutter architecture is that only a small number of transistors are needed inside the pixel. However, a disadvantage with rolling shutter architecture is that the time resolution is fundamentally constrained by the number of columns. E.g. for the ALPIDE with 1024 columns, the theoretical time resolution could be no higher than $1024/40 \text{ MHz} \approx 25 \mu\text{s}$.\footnote{40 MHz is ALPIDE’s expected clock frequency. More on this later.} This restriction is undesirable as the ITS has a minimum time resolution requirement of $30\mu\text{s}$ [9]. For the ALPIDE the time resolution is $2\mu\text{s}$ [9].

Lastly, as illustrated by the steps above, the front-end amplifier and the discriminator are continuously active, hence the name Monolithic Active Pixel Sensors (MAPS).

**Multi Event Buffer (MEB)**

The MEB are implemented to reduce dead-time within the pixel cell. When a snapshot is taken, one MEB uses approximately 600 ns to store a pixel hit [10, page 99]. And the readout of a hit takes about 50 ns [43]. A MEB is therefore at least busy for 650 ns following a snapshot, introducing a dead-time. By implementing three latches, this dead-time is drastically reduced [10]. However, it should be noted that in a case where all the three latches are filled in less than 650 ns, a dead-time will be introduced. The handling of such cases is discussed in section 3.3.2.

**Reverse substrate bias**

A voltage can be applied to the deep pwells of the pixel cells as well to the substrate. The overall effect is an applied back bias voltage to the collection diode $C_d$, see Figure 3.4. This back bias voltage will throughout this thesis be referred to as ALPIDE’s back bias voltage, or simply $V_{BB}$. Here $V_{BB}$ will relate to the reverse-bias voltage $V$ of the collection diode as follow [12]:

$$V = V_{BB} + V_{RST},$$

(3.1)

where $V_{RST}$ is the pixel reset voltage. Thus, by applying a negative voltage to $V_{BB}$, the depletion region inside the pixel cell will increase, see equation 2.8. Consequently increasing the charge collection efficiency, as explained in section 2.3.2.
Figure 3.3: Block diagram of one ALPIDE pixel cell. The Collection Diode is where the hit is registered. VPULSE_* is used to generate test hits. The Pixel analog Front end amplifies the signal and if the signal is above a predefined threshold (THR), OUT_D is set high. OUT_D is then written to an empty MEB if STROBE is high. [43, Figure 1.2].

Figure 3.4: A simplified cross-section of an ALPIDE pixel. The N-WELL DIODE is the Collection Diode in Figure 3.3 and $C_d$ in this figure. A particle hit will generate electron-hole pairs, electrons entering the depletion region will then be collected by the N-WELL DIODE. This will result in a voltage drop. Reprinted from [12, Figure 3.3], with permission from J. W. van Hoorne.
3.3 Digital Periphery

This section documents the data flow in the ALPIDE. The section will highlight how different operation modes of the ALPIDE affect this data flow.

3.3.1 Taking a snapshot

As discussed in the previous section: pixels are arranged together two and two in columns separated by a priority encoder. An important feature of the Priority Encoders is to distribute the STROBE signal, as the STROBE signal is responsible for taking a snapshot of all the pixels. The STROBE signal can be generated in two ways:

- **Internally**: here the STROBE signal is generated by an internal sequencer. The frequency and the duration of the STROBE signals are set in ALPIDE’s configuration registers. After the right configurations are set, the internal sequencer can be started by a single external TRIGGER command, as explained in the bullet point below.

- **Externally**: here the STROBE signal is generated after the reception of an external TRIGGER command. The external TRIGGER command is sent by sending one of the following bytes over the control port: 0xB1, 0x55, 0xC9, 0x2D. More on the control port and TRIGGER command in section 3.4.1.

Figure 3.5 illustrates the data flow within the ALPIDE chip. After the pixel hit is stored in MEB the data gets read out by the Region Readout Unit (RRU). This data is then organized into data packets in the Top Readout Unit (TRU). As illustrated by Figure 3.5 this happens by appending “Frame info” to the data. Here “Frame info” is stored in the Frame FIFO, this FIFO is used to keep track of frames. One entry in the FIFO corresponds to one frame, that is, one STROBE signal will give one entry. The frame FIFO is 64-word deep which means that the ALPIDE can in principle store up to 64 frames at the same time.

**Busy signal**

If the STROBE has a high frequency, entries will be stored more rapidly in the Frame FIFO. If also the number of pixel hits is significant this will lead to a build
3.3. DIGITAL PERIPHERY

Figure 3.5: Block diagram showing the data flow in the ALPIDE chip. Reprinted from [48, Figure 2], with permission from S.V. Nesbo.

up off entries in the Frame FIFO, as data is not offloaded fast enough. And, when
the FIFO has 48 entries a busy data word will be transmitted off chip. As discussed
in the next section, this busy data word is also transmitted if one of the MEB is
saturated. The intention is that the off-detector electronics will reduce the trigger
rate when a busy word i detected. Avoiding the Frame FIFO or the MEB to overflow.

Strobe gap and strobe length

As mentioned above, the STROBE gap, how long the STROBE signal is low, and
the STROBE length, how long the STROBE signal is high, can be configured. These configurations have effects on the capturing of pixel hits. E.g., as the OUT_D signal is typically high for $\approx 5\mu s$ [43], a STROBE configuration with multiple high and low within that time will capture multiple hits from the same particle. Or, if the STROBE length is long, but the STROBE gap is short, a gap happening in the middle of a pixel hit will result in two pixel hits captured from the same particle hit. Figure 3.6 illustrates these two scenarios. These effects are important to consider when setting the STROBE length and STROBE gap, as they are usually both unwanted effects. By setting the STROBE gap longer than $5\mu s$ both effects can be avoided.

Pixel hits per frame

Moreover, the STROBE length will also determine the expected number of pixel hits each frame. This number can be important as it will indicate which frame rate we can operate the ALPIDE chip on. This comes from the fact that only a given amount of pixel hits can be transmitted off-chip in a given time without resulting in a Frame FIFO or MEB overflow, thereby losing data. E.g. in article [48] simulations were done with frame rates at 100 kHz with a hit density of 19.5 hits/cm$^2$, here about 99.6% of the total triggers were accepted. That is, about 0.4% of the total frames

---

This is set in ALPIDE’s configuration register 0x0005 and 0x0006 [43].
3.3. DIGITAL PERIPHERY

Figure 3.6: Figure (a) shows the same hit being captured twice as a consequence of the strobe gap being too short. Figure (b) shows a hit being captured multiple times as both the strobe gap and the strobe length is too short.

were not recorded, and data was lost as a consequence.

For this reason, calculating the expected pixel hits per frame can indicate which frame rate we can operate on. Take for example a radioactive alpha source of 0.01 $\mu$Ci which radiate uniformly. Assuming that the disintegrations of the source and the radiated number of particles are equal, and for the sake of simplicity, assuming that the ALPIDE chip would receive half of the radiated particles. The ALPIDE chip would receive:

$$\frac{1}{2} \times 0.01 \mu\text{Ci} = 1850 \text{ particles/ s}.$$  

That is, 1.85 particles/ ms. Thus having a strobe length of 1 ms would mean that the expected number of pixel hits each frame would be:

$$\langle \text{hits/frame} \rangle = 1.85 \text{ particles/ ms} \times 1 \text{ ms} \times N_{cz},$$

Here $N_{cz}$ is the cluster size, that is the number of pixel hits per particle. Setting e.g $N_{cz} = 13 \text{ hits/particles}$, we get:

$$\langle \text{hits/frame} \rangle = 1.85 \text{ particles/ ms} \times 1 \text{ ms} \times 13 \text{ hits/particles} \approx 24.$$  

Thus, using the result from [48], here the hits per frame would be roughly $19.5 \text{ hits/ cm}^2 \times (30 \text{ mm} \times 15 \text{ mm}) = 87.75$, one would expect the ALPIDE to be operating safely at 100 kHz for this radiation environment.\footnote{This value is not randomly chosen, this was about the mean cluster size in one of the measurements done later in this thesis, see Figure L.4.}

In general one can write the expected pixel hits per frame as:

$$\langle \text{hits/frame} \rangle = \alpha R S_I N_{cz}, \tag{3.2}$$

here $\alpha$ represents the portion of particles radiated hitting the ALPIDE chip. $R$ is the radioactivity and $S_I$ the strobe length.

3.3.2 Operation modes

As mentioned earlier the ALPIDE has numerous registers enabling monitoring of internal DACs, configuration settings, readout of internal memory, etc. Although this thesis will not focus on these registers, some important configuration distinctions should be made. The different readout modes and ALPIDE’s module identifiers are two of these.

\footnote{The value $30 \text{ mm}$ and $15 \text{ mm}$ are the dimensions of the ALPIDE.}
3.3. DIGITAL PERIPHERY

Triggered mode

OUT_D

STROBE

write MEB A B C

read MEB A

BUSY

Figure 3.7: ALPIDE configured in triggered mode. OUT_D corresponds to the pixel being hit or not. As illustrated by the red cross, STROBE is supposed to go high writing to buffer A, however, as the readout of buffer A is not finished, this strobe is skipped. A skipped strobe is transmitted as an empty frame. Busy is asserted when there are no more free MEB.

Readout modes

Important for the storage of pixel hits is the difference between the chip configured in continuous or triggered mode. The difference between these two modes lies in the logic handling a situation where the MEB is full, and the corresponding pixel has received a hit.

- **Triggered**: This mode prioritizes stored pixel hits in the MEB over new pixel hits. If a new STROBE is scheduled, but the MEB are full, the STROBE is skipped. That is, a whole frame will be skipped. The skipped frame will nevertheless get an entry in the Frame FIFO, and will be transmitted as an empty chip data packet.

- **Continuous**: In continuous mode newly received hits are priorities over stored pixel hits in the MEB. This is done by ensuring that there is at least one latch free in the MEB. If a pixel hit is being written to the last free MEB, the chip will interrupt the ongoing frame readout to free up one MEB. The data packet associated with the interruption will then only contain data up until the interruption. This data loss is reported in one of the flags in the corresponding data packet trailer.

These two modes are illustrated and further explained in Figure 3.7 and Figure 3.8. The two modes are intended for two different use cases [43].

The triggered mode is intended for use cases where the duration of the STROBE signal is configured to be short, typically to a few hundredths of nanoseconds [43]. The STROBE gap is set to be long enough to ensure no multiple captures, as shown happening in Figure 3.6b. A longer STROBE gap will ensure a longer time for the readout of the MEB. Hence an empty frame, as illustrated in Figure 3.7, is less likely to happen. Moreover, triggered mode is intended to be used with the external strobe generation, thus when a busy signal is detected, due to almost full MEB, the off-detector electronics can cease its external triggering for a moment.
3.3. DIGITAL PERIPHERY

Continuous mode

![Diagram of Continuous Mode]

Figure 3.8: In continuous mode, when writing to the last free buffer C, buffer A is
forcefully deleted. This will cause partial data readout, but ensure that no frames
are skipped. Busy is asserted when there are only one free MEB and de-asserted
when there are two free.

Whereas for the continuous mode the STROBE signal is typically configured to
be longer (a few microseconds) and the gap between consecutive strobes shorter,
keeping capture time maximum. In the continuous mode, no frames are skipped,
but data may be lost. This mode is typically used when the internal sequencer is
utilized, as it then does not exist any immediate method of slowing down the trigger
rate.

That is, all considered, triggered mode is typically used when capturing all the
data in a frame is more important than having a deterministic timing of frames.
Whereas the continuous mode is typically used when having a deterministic timing
of frames is more important than capturing all the data.

Module Identifiers

Each ALPIDE chip has a corresponding chip id. The chip id is set by pulling the
CHIPID[6:0] ports either high or low. The id is used to both identify the chips
topological locations, when multiple chips are interconnected, and to set the chip’s
module. The ALPIDE chip can be configured to three different modules: Inner
Barrel Chip, Outer Barrel Master or Outer Barrel Slave. Which of these modules
the chip is configured to should correspond to which layer the chip is residing on
inside the ITS. The reason for the distinctions is that in the inner layers of the ITS
a much higher pixel hit rate is expected. Thus, faster data transmission of pixel hit
data is needed. For this reason, the module the chip is configured to will determine
which data interface is used for transmitting data. Under is a summary of the
consequence for different module configurations with respect to data transmission.

- **Inner Barrel Chip, Outer Barrel Master**: Data is transmitted over the
  serial data port. The data rate is: 1.2 Gb/s, 600 Mb/s or 400 Mb/s.

- **Outer Barrel Slave**: Data is transmitted over the parallel data port, this
  port gives a data rate of 320 Mb/s.
Table 3.1: Some of ALPIDE’s physical port names with their respective interface names. Table data taken from [43, Table 2.1].

<table>
<thead>
<tr>
<th>Physical Port</th>
<th>Interface</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCLK_P</td>
<td>System clock</td>
<td>INPUT</td>
<td>Outer Barrel Master clock input</td>
</tr>
<tr>
<td>MCLK_N</td>
<td>System clock</td>
<td>INPUT</td>
<td>Outer Barrel Master clock input</td>
</tr>
<tr>
<td>DCTRL_P</td>
<td>Control Port</td>
<td>BIDIR</td>
<td>Differential Control Port</td>
</tr>
<tr>
<td>DCTRL_N</td>
<td>Control Port</td>
<td>BIDIR</td>
<td>Differential Control Port</td>
</tr>
<tr>
<td>DATA[7:0]</td>
<td>Parallel Data Port</td>
<td>BIDIR</td>
<td>Outer Barrel local bus</td>
</tr>
<tr>
<td>CHIPID[6:0]</td>
<td>Chip id</td>
<td>INPUT</td>
<td>Topological- and module identifier</td>
</tr>
<tr>
<td>HSDATA_P</td>
<td>Serial Data Port</td>
<td>OUTPUT</td>
<td>Serial data output</td>
</tr>
<tr>
<td>HSDATA_N</td>
<td>Serial Data Port</td>
<td>OUTPUT</td>
<td>Serial data output</td>
</tr>
</tbody>
</table>

*Ports ending with P/N are differential lines.*

### 3.4 Interface

The ALPIDE has numerous input/output (I/O) ports for data transmission, inter-chip communication, analog monitoring, etc. In this thesis, the focus has been to interface with one ALPIDE chip. Thus some interfaces, as well as some of their attributes, will not be discussed. The interfaces that will be in focus in this thesis are the ones shown in Figure 3.2. Namely: **Control Port**, **Parallel Data Port** and **Serial Data Port**. The former interface is responsible for communication between the off-detector readout electronics and ALPIDE’s configuration registers. This interface is needed to transmit the TRIGGER command. The two latter interfaces are used for data transmission. In Table 3.1 an overview of the physical port names, with corresponding interfaces are shown.

Before discussing the above interfaces individually it should be noted that the ALPIDE chip runs on an external clock. This clock is received on a differential input port and is ALPIDE’s only clock source, see Table 3.1. The clock will throughout this thesis be referred to as the system clock. The clock’s expected frequency is 40 MHz, this frequency is the frequency that a bunch of protons has around the LHC [43]. Hence called the LHC bunch frequency.

#### 3.4.1 Control Port

The differential control port, or just called the control port, is a bi-directional port, supporting serial signaling at 40 Mb/s on differential lines. The control line serves two purposes:

1. Access to ALPIDE’s configuration and monitor registers and as well the in chip memory.
2. Send TRIGGER commands or other types of commands.

The communication protocol is a protocol synchronized to the system clock. The off-detector electronics initiate the transactions over the control line. That is, the off-detector electronics operates as master and the ALPIDE chip as slave. The ALPIDE chip encodes all characters transmitted over the control line in Manchester encoding.
encoding. The off-detector readout electronics can choose whether or not to encode in Manchester encoding. The rule is regardless: the ALPIDE chip samples the control line (DCTRL_P and DCTRL_N) at the system clock’s rising edge. Manchester encoding can, however, be disabled in one of ALPIDE’s configuration registers.\footnote{In register address 0x0010, the CMU and DMU configuration register, bit 5 enables/disables Manchester encoding over the control port \cite[page 41]{43}.} This is done throughout this thesis, and no Manchester encoding is used by the ALPIDE chip nor the readout electronics. This is done as short electrical lines are being used, and the readout electronics control the system clock. Thus no clock recovery is needed.

### Control Port data format

In Figure 3.9 the different valid transaction formats over the control line is seen. As illustrated in the figure the OPCODE is the first byte to be sent. The OPCODE communicates to the ALPIDE chip which transaction format to expect. In Table 3.2 the different valid OPCODES with their corresponding formats are listed. Table 3.2 illustrates that for example the ALPIDE can be reset by sending a single opcode (GRST) over the control line.

The bus turnaround in Figure 3.9 represents a phase when the off-detector readout electronics releases the bus and the ALPIDE chip takes over the bus or vice versa. This phase follows strict timing specifications and, as we will see, is a good reason for utilizing Field Programmable Gate Arrays (FPGA) as off-detector electronics.

#### 3.4.2 Parallel Data Port

The parallel data port, or referred to as DATA[7:0], is an 8-bit data bus intended for inter-chip communication. This bus is used in the Outer Barrel of the ITS. In such a configuration there will be one Outer Barrel Master and several Outer Barrel Slaves, illustrated in Figure 3.10. Outer Barrel Slaves will transmit hit data to the
Table 3.2: Valid opcodes with their corresponding data format. Table data taken from [43, Table 3.1].

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Hex value</th>
<th>Purpose</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIGGER</td>
<td>0x55</td>
<td>Send external trigger</td>
<td>Trigger command</td>
</tr>
<tr>
<td>GRST</td>
<td>0xD2</td>
<td>Chip global reset</td>
<td>Broadcast command</td>
</tr>
<tr>
<td>PRST</td>
<td>0xE4</td>
<td>Pixel matrix reset</td>
<td>Broadcast command</td>
</tr>
<tr>
<td>PULSE</td>
<td>0x78</td>
<td>Pixel matrix pulse</td>
<td>Broadcast command</td>
</tr>
<tr>
<td>BCRST</td>
<td>0x36</td>
<td>Bunch counter reset</td>
<td>Broadcast command</td>
</tr>
<tr>
<td>DEBUG</td>
<td>0xAA</td>
<td>Sample states</td>
<td>Broadcast command</td>
</tr>
<tr>
<td>RORST</td>
<td>0x63</td>
<td>Readout reset</td>
<td>Broadcast command</td>
</tr>
<tr>
<td>WROP</td>
<td>0x9C</td>
<td>Start write transaction</td>
<td>Unicast/Multicast read</td>
</tr>
<tr>
<td>RDOP</td>
<td>0x4E</td>
<td>Start read transaction</td>
<td>Unicast read</td>
</tr>
</tbody>
</table>

b Or: 0xB1, 0xC9, 0x2D.

Figure 3.10: An illustration of the Outer Barrel setup. Each block represents an ALPIDE chip configured as Outer Barrel Master or Outer Barrel Slave. The Outer Barrel Master receives hit data over the local data bus from the Outer Barrel Slaves. This data is then serialized and transmitted over the Serial Data Port to off-detector electronics.

Outer Barrel Master in turn. The Outer Barrel Master will then transmit this data over the Serial Data Port. As is illustrated in Figure 3.10 the parallel data port is a bidirectional line, here every chip listens for their turn to transmit data to the Outer Barrel Master.

The local data bus has two different operating modes. The bus can either use 8 lines at 40 MHz, or 4 lines at 80 MHz. The latter is called Double Data Rate (DDR) transmission as the chip will transmit data on both rising and falling clock edge. The former is called Single Data Rate (SDR) transmission. DDR is enabled by default. However, throughout this thesis SDR will be utilized, this is because faster lines are more error-prone with respect to signal integrity, and the extra 4 lines needed for SDR does not present any spacial problems.

Even though the parallel data port is intended for inter-chip communication it also works well for data offloading when interfacing with one chip. The parallel data port offers a considerable slower data interface than the serial data port. Thus, it makes for an easier electrical connection and the use of for example high-speed transceivers are not needed. This will further be discussed in section 4.2.
Figure 3.11: Example of a data packet transmitted over DATA[7:0]. Each data packet corresponds to one frame. The end of the data packet is marked by the CHIP TRAILER word.

Parallel data format

The parallel data port operating in SDR sends one byte each clock cycle. Each byte is part of a data word. The data words are listed in Table 3.3. The transmission of pixel hit data happens through the transmission of data packets. Each data packet corresponds to a frame taken by the ALPIDE. In Figure 3.11 an example data packet can be seen. As illustrated, each data packet starts with a CHIP HEADER and ends with a CHIP TRAILER, here the REGION HEADER and DATA SHORT/DATA LONG data words in between contain the pixel hit information.

Clustering

The DATA LONG data word is only transmitted if clustering is enabled in ALPIDE’s configuration registers. This data word can compress up to 8 neighboring pixel hits. Hence, the total amount of data the readout electronics need to handle will be reduced. How much data compression clustering will give will be discussed in section 6.2. See Appendix A for a detailed explanation of the DATA LONG data word.

Busy data word

As seen in Table 3.3 two data words, BUSY ON and BUSY OFF, can also be transmitted over the parallel data port. This indicates to the off-detector electronics if the busy signal is high or low and is transmitted as soon as possible without splitting data words in the data packet. That is, the signal can be transmitted in between data words but not, for example, split a CHIP EMPTY FRAME data word. This busy word reflects the busy signal discussed in the previous section. As discussed, this busy signal would enable the off-detector electronics to slow down triggers to prevent data loss.

3.4.3 Serial Data Port

The serial data port is a high-speed serial data port utilized by chips configured as Inner Barrel Chips or Outer Barrel Masters. ALPIDE chips configured to Inner Barrel Chips will only utilize the serial data port for data transmission. As mentioned earlier the data port can be configured to a line rate of 1.2 Gb/s, 600 Mb/s or 400 Mb/s. However, the line utilizes 8b10b data encoding and the effective data rate, therefore, becomes 960 Mb/s. The data format is the same as that of the parallel

More information on the encoding of pixel hits can be found in Appendix A.
data port, seen in Table 3.3, with one exception. A comma word is also transmitted over the serial data line. The comma word is only transmitted when there are no data packets to transmit and is used for clock recovery and synchronization to the serial data stream.

Table 3.3: Data format used on the parallel data bus and the serial data line. Table data taken from [43, Table 3.37].

<table>
<thead>
<tr>
<th>Data word</th>
<th>Bit length</th>
<th>Value (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>8</td>
<td>1111_1111</td>
</tr>
<tr>
<td>CHIP HEADER</td>
<td>16</td>
<td>1010&lt;CHIPID[3:0]&gt;&lt;bunch_counter[10:3]&gt;</td>
</tr>
<tr>
<td>CHIP TRAILER</td>
<td>8</td>
<td>1011&lt;readout_flags[3:0]&gt;</td>
</tr>
<tr>
<td>CHIP EMPTY FRAME</td>
<td>16</td>
<td>1110&lt;CHIPID[3:0]&gt;&lt;bunch_counter[10:3]&gt;</td>
</tr>
<tr>
<td>REGION HEADER</td>
<td>8</td>
<td>110&lt;region_id[4:0]&gt;</td>
</tr>
<tr>
<td>DATA SHORT</td>
<td>16</td>
<td>01&lt;encoder_id[3:0]&gt;&lt;addr[9:0]&gt;</td>
</tr>
<tr>
<td>DATA LONG</td>
<td>24</td>
<td>00&lt;encoder_id[3:0]&gt;&lt;addr[9:0]&gt;0&lt;hit_map[6:0]&gt;</td>
</tr>
<tr>
<td>BUSY ON</td>
<td>8</td>
<td>1111_0001</td>
</tr>
<tr>
<td>BUSY OFF</td>
<td>8</td>
<td>1111_0000</td>
</tr>
</tbody>
</table>
Chapter 4

Readout System Proposal

This chapter discusses the requirements for a readout system towards a single ALPIDE. The chapter will lay the foundation for the readout system developed in this thesis, and a discussion of the pros and cons for different setups will be made. First, section 4.1 will present ALPIDE’s existing readout solutions as these solutions worked as guidance for this thesis. Section 4.2 will then discuss the requirements for the readout system developed in this thesis. The last section 4.3, provides a summary.

4.1 Existing systems

This section goes through some of the key aspects of the already developed/proposed readout systems for the ALPIDE. This section discusses parts of these systems that are important for the development done in this thesis, particularly the interfaces and the hardware utilized.

4.1.1 ALICE ITS

As the ALPIDE is developed for the upgrade of ALICE’s ITS, as discussed in section 3.1, it is natural to investigate ITS’ proposed readout system. As mentioned, the ALPIDE chip will be arranged on layers surrounding the collision center at ALICE, this can be seen in Figure 3.1. Each layer can further be divided into staves, here seen in Figure 4.1. Depending on which layer the stave is residing on, the ALPIDE chips will utilize different readout solutions. For the Inner Barrel staves, each ALPIDE chip will transmit data independently over the serial data port. One Inner Barrel stave will house 9 ALPIDE chips, and consequently, there will be a total of 9 serial data lines. Whereas for the Outer Barrel staves, 7 ALPIDE chips will be grouped together with the local data bus, as explained in section 3.4.2. Each group will have a dedicated Outer Barrel Master chip who transmit the data over the serial data port. On an Outer Barrel stave, there will exist 8 or 14 of these groups, surmounting to a total of 8/14 serial data lines [33].

Interfaces

Regardless of the stave configuration, data is transmitted off the staves using the serial data port. The only difference for the ”off-stave” readout electronics, called
4.1. EXISTING SYSTEMS

Figure 4.1: Illustration of staves in the ITS. Two different staves are used depending on the positioning in the ITS. Inner Barrel to the left and Outer Barrel to the right. Data transmitted from the staves are transmitted over the serial data port. [33, Figure 1.3].

Figure 4.2: Architecture of the Readout Unit (RU) receiving the data transmitted from a stave. A RU connected to an Inner Barrel stave will receive 9 serial data lines. Whereas a RU connected to an Outer Barrel stave will receive 8/14 serial data lines. The RU will host FPGAs. [46, Figure 9].
the Readout Unit (RU), will be the number of serial data lines. This is illustrated in Figure 4.2. The figure illustrates the proposed architecture of the RU which will be connected to each stave. Here there will be three different interfaces between the RU and the staves: Clock line, serial data port, and control port.

The RU will be about 4m to 5m from the staves [33]. Thus, when transmitting 1.2 Gb/s over the serial data line, proper impedance matching between the stave and the cable is critical. The performance of the cable is therefore important. Currently the cable of choice is the Samtec AWG30 Twinax "Firefly" [33, page 121]. However, for a single chip readout system, with no requirements for the distance between the ALPIDE chip and the off-detector electronics, the Firefly cable might be excessive, this is discussed further below.

Readout Unit (RU)

The readout electronics will consist of the RU shown in Figure 4.2. There will be one RU for each stave, each of them hosting FPGA’s to operate the control line and handle the data stream over the serial data line [46]. The RU will then transit the data to a unit called the Common Readout Unit (CRU) by optical links [49], see Figure 4.2. This unit will not be discussed, but it should be noted that the optical links have a speed of 3.2 Gb/s. The high data rate of the optical links is important to prevent data ”pile-up” in the RU.

RU Firmware

The main firmware tasks handled by RU can be summarized by the block diagram in Figure 4.3. As illustrated some key firmware modules handles the interface between the stave and the RU. These modules are, in particular, the modules highlighted as 1 and 2 in Figure 4.3. The former module handles the control line, that is, it implements the protocol discussed in section 3.4.1. This module also issues trigger commands, as these are issued over the control line. The latter module handles the data stream received from the serial data port, as explained in section 3.4.3. Following the data flow of the serial data in Figure 4.3, the data is directly transited to the CRU. Lastly, as highlighted by block 3, is the module handling the reception of triggers from the CRU. These triggers are, depending on the configuration, issued as trigger commands by block 1.

4.1.2 ITS Prototype Readout Unit

The ITS Prototype Readout Unit was developed to address different R&D tasks concerning the development of the RU. This board hosts a Xilinx Kintex-7 FPGA and utilizes the Samtec FireFly cables for the connection between the board and the ALPIDE [49]. Here an adopter card was made for a FireFly connection, which can be seen in Figure 4.4.

This board will not be discussed in detail as there were little knowledge publicly available at the time of writing this thesis. Further, as this board was specifically developed to address R&D tasks of the RU, this board was considered to be not well suited for the development of an independent platform. Nevertheless, the Adopter slave card, depicted in Figure 4.4, would be available to the Electronics section at
4.1. EXISTING SYSTEMS

From/To slaves

From/To CRU

Figure 4.3: Block diagram of the main firmware tasks handled by the RU. Block 1 handles the control line. Block 2 handles the serial data line. Block 3 handles the input of triggers from the CRU. Reprinted from [50, Slide 11], with permission from ALICE.

UiO [90], and was considered to be used in the readout system developed in this thesis.

Adopter slave

The adopter slave card, shown in Figure 4.4, is used to convert the PCIe x8 connector of the ALPIDE Carrier card, explained in the next section, to the FireFly cable. This enables the connection of the ALPIDE chip to the Prototype RU board. An Adopter slave card could be acquired and make it possible to develop a system towards this card. This would imply that a development board with a FireFly connector had to be purchased or that a converter card from FireFly to some other I/O standard had to be obtained. The best alternative for a development board hosting a FireFly connector was found to be the Xilinx Virtex UltraScale+ FPGA VCU118 Evaluation Kit, costing $6,995 [52, visited 6/2018]. However, due to its high price, and its somewhat disproportionate specification for a system interfacing with one ALPIDE chip, this option was not pursued. In addition, the solution of creating, for example, an FPGA Mezzanine Card (FMC) to FireFly converter board was examined. However, this task was considered superfluous as one could then instead create a FMC to PCIe x8 card, directly connecting to the ALPIDE Carrier card, thus skipping the FireFly cable and the Adopter slave card. As will be discussed, this was the solution chosen, but using simple PMOD headers instead of FMC.\footnote{PMOD is a standard developed by Digilent for Peripheral Modules. The standard uses 100mil spaced, 25mil square, pin-header style connectors [53].}
4.1.3 DAQ board

Under the development of the ALPIDE chip, test systems were made to test prototypes of the ALPIDE. One of these systems is called the pALPIDEfs test system [54]. This system consists of a DAQ board and an ALPIDE Carrier card. The system can be seen in Figure 4.5. The DAQ board is developed by Istituto Nazionale di Fisica Nucleare (INFN) [54], [55], and the ALPIDE Carrier card is developed by CERN. This system was obtained and was successfully used in radiation tests (see Figure 5.17). That said, the board did present some difficulties. Firstly, little knowledge was available of the board’s hardware or firmware at the time of this thesis. As a result of this, the way radiation measurements were done was to utilize a software framework made by CERN [56]. This method works well for doing radiation measurements, but gives little insight into the development of a new readout system. Secondly, utilizing the software framework made by CERN [56] was not ideal as it was under development and therefore not properly documented. Thirdly, the lack of hardware and firmware documentation makes the DAQ board inadequate for characterizing the ALPIDE chip with respect to readout speeds and data handling. This is because the low-level handling of data is "hidden". For these reasons the development of a new readout system will hopefully bring new insights to the ALPIDE’s readout functionalities.

It should be noted that one way of controlling the DAQ board is to use a signal generator to generate trigger signals. This way of operating the ALPIDE is a reflection of the RU, where a trigger is received by block 3, which in turn is issued as trigger command by block 2, see Figure 4.3. The need for a signal generator is in principle, superfluous for taking snapshots. Reason being the FPGA on the DAQ board can issue trigger commands without the signal generator. Therefore, when developing a readout system, that is not meant towards the ITS, this way of triggering is excessive.
4.1. EXISTING SYSTEMS

Figure 4.5: The ALPIDE Carrier card (left) connected to the DAQ card (right).

**Interfaces**

Since there exists no firmware documentation for the DAQ board it becomes complicated to know which interface is being utilized for data offloading, the parallel data port or the serial data port. By inspecting the DAQ board, one can see that the board hosts an Altera Cyclone IV E FPGA, product line EP4CE40. This type of FPGA has no transceivers, thus reading from the serial port at 1.2 Gb/s is unlikely. The FPGA product line does however, host dedicated LVDS pairs. Here the LVDS interface can function up to 840 Mb/s [57]. As the serial data port is a LVDS pair, see Appendix B, the DAQ board could utilize the serial data port configured to 600 Mb/s or 400 Mb/s. However, measurements were done with an oscilloscope which indicated that the DAQ board utilized the parallel data lines for data offloading, see Appendix D Figure D.2.

**ALPIDE Carrier**

The ALPIDE Carrier card is a holding card for the ALPIDE chip. As seen from the previously mentioned readout systems, they all interface towards this card, and as seen from Figure 4.5, the DAQ board is specifically designed to interface towards this card. The basic functionality of the ALPIDE Carrier card is to host the ALPIDE chip and route ALPIDE’s signals directly to its PCIe x8 male connector. It is important to note that the ALPIDE chip does not use PCIe protocol, but simply uses the connector as a form factor. Meaning there are no apparent reasons for limiting the readout system to a PCIe x8 connector. However, as the ALPIDE chip available to the Electronics section at UiO came attached to the ALPIDE Carrier card, the system developed in this thesis will also be made towards this card. The ALPIDE Carrier card is further explained in section 5.2.1, where the readout system developed in this thesis is presented.
4.2 REQUIREMENTS

Table 4.1: Specific requirements for the readout system, as an outcome of the discussion in this chapter.

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.1</td>
<td>Use the ZedBoard as readout board.</td>
</tr>
<tr>
<td>R.2</td>
<td>Develop towards the ALPIDE Carrier card.</td>
</tr>
<tr>
<td>R.3</td>
<td>Develop a HDL module implementing the control port protocol.</td>
</tr>
<tr>
<td>R.4</td>
<td>Develop a HDL module handling data packets from the parallel data port.</td>
</tr>
<tr>
<td>R.5</td>
<td>Handle the data streams without reducing ALPIDE’s frame rate.</td>
</tr>
<tr>
<td>R.6</td>
<td>Lines should handle up to 40 MHz switching speeds.</td>
</tr>
<tr>
<td>R.7</td>
<td>Develop analyzing software, analyzing data streams.</td>
</tr>
<tr>
<td>R.8</td>
<td>Develop a PCIe x8 to PMOD converter board.</td>
</tr>
<tr>
<td>R.9</td>
<td>Power the ALPIDE from the ZedBoard.</td>
</tr>
<tr>
<td>R.10</td>
<td>Control ALPIDE’s $V_{BB}$ voltage.</td>
</tr>
</tbody>
</table>

4.2 Requirements

This section discusses requirements for the readout system towards the ALPIDE. The section arguments for some of the design choices done in Chapter 5. The general guidelines for the readout system are summarized below.

1. Inexpensive.
2. Simple.
3. Independent.
4. Communicate over the control interface.
5. Handle data streams (Offload pixel hit data).

Each of these guidelines are discussed in turn in the next subsections. The discussion is meant to lay the foundation for specific requirements for the readout system. The specific requirements, as a result of the discussion, are summarized in Table 4.1.

Inexpensive

The readout system should be as inexpensive as possible without sacrificing significant functionality. For this reason the ZedBoard was chosen as it could be purchase for $475 [58, visited 6/2018], and hosted PMOD headers. This sets R.1 in Table 4.1.

Simple

The readout system developed in this thesis should be as simple as possible without sacrificing significant functionality. This guideline comes from the fact that we want to keep all possibilities open for what the system developed in this thesis can be used for in the future. For this reason, the readout system should be easily modified, and time should be spent on parts of the readout system that are a necessity for future readout system. For this reason, the readout system was developed towards the ALPIDE Carrier card, as creating a new host card for the ALPIDE chip would
both be time expensive and perhaps superfluous for future systems. This sets R.2 in Table 4.1.

To note, interfacing towards the ALPIDE Carrier card would be in contradiction to guideline 3. But, this card functioned almost solely as a host card, and would not "hide" any deeper functionalities in difference to the DAQ board. Moreover, the ALPIDE Carrier card schematics and PCB layout were openly available, making development done towards this card easier [59]. Using the ALPIDE Carrier card meant that a converter board should be made, enabling a connection between the ZedBoard and the ALPIDE Carrier board. Therefore a converter board was made in this thesis, converting the PCIe x8 connection of the ALPIDE Carrier card to the PMOD connectors of the ZedBoard. This sets R.8 in Table 4.1.

The ALPIDE chip requires a 1.8 V power supply [43]. And, as using an external power supply source is impractical for lab experiments, the ALPIDE should be powered from the ZedBoard. This sets R.9 in Table 4.1. Moreover, the readout system shall also be able to control the back bias voltage, that is the $V_{BB}$ voltage discussed in section 3.2. This sets R.10 in Table 4.1.

**Independent**

Developing an independent system was one of the key motivations for developing a new readout system for the ALPIDE. This requirement originated from the issues with the DAQ board, here sparse documentation of the board existed, thus making measurements time-consuming and insufficient. Moreover, much of the documentation found on other readout units, such as the ITS Prototype RU discussed in section 4.1.2, was internal documentation at CERN and not available for third-party developers. Even the documentation of the ALPIDE chip, found in [43], was as of the time of writing not complete, making development time-consuming. One of the main goals for this thesis was in general to highlight parts of ALPIDE’s functionalities, which were not well documented, making it easier for future developments. Therefore, relying on CERN internal documentation was considered unsatisfactory. For these reasons, the readout system was emphasized throughout this thesis to be as independent as possible from CERN while being well documented for future use. As a consequence of this, most of the development done in this thesis should be from the ground up, that is, not relying on any other sources. This meant that both of the Hardware Description Language (HDL) modules, discussed in the next two subsections, and the analyzing software should be fully developed in this thesis. Thus setting R.3, R.4 and R.7 in Table 4.1.

**Communicate over the control interface**

To control the ALPIDE, it is necessary to be able to communicate over the control interface, as this interface enables the control of all of ALPIDE’s configuration registers and the trigger command, see section 3.4.1.

A functional control interface does not per se require an FPGA as used by the existing readout solutions. E.g., in the CERN summer school project [60], an Arduino was used to communicate over the control interface. Using an Arduino for communicating over the control port require the ALPIDE chip to operate on a lower clock speed than its expected 40 MHz speed [43]. This is because the Arduino clock speed is 16 MHz. Hence, the theoretical maximum of the I/O toggle speed is 16 MHz.
This solution would then make the maximum frame rate of the ALPIDE chip lower. Using a faster processor, with I/O toggle speeds of 40 MHz, would overcome this issue. However, such an implementation requires the full usage of the processor during a read or write transaction. In other words, triggering in parallel to handle data streams from the ALPIDE becomes complicated. Moreover, utilizing an FPGA is also favorable with respect to the control interface protocol, as this protocol follows strict timing specifications. For example, the bus turnaround phase in Figure 3.9 should happen exactly 5 clock edges after the last byte was transmitted. An FPGA also offer considerable more I/Os than a normal micro-controller, which is needed for this thesis as it will utilize the parallel data port which by itself consumes 8 I/O ports.

Therefore, considering the discussion above, utilizing an FPGA would be the best solution. As discussed in section 4.1, this is also the solution for all of the existing readout systems mentioned. This means that a firmware module implementing the control port protocol should be developed. This module shall be developed in-house. This lays the foundation for R.3 in Table 4.1. Since the ALPIDE would be running on a 40 MHz clock, the control line needs to handle switching speeds of 40 MHz. This sets requirement R.6 in Table 4.1.

Handle data streams

To evaluate pixel hit data, the readout system must be able to offload data packets from the ALPIDE. This can be done in three different ways.

First, data packets can be read out over the control interface. This is done by reading directly from ALPIDE’s internal FIFO. However, as seen by Figure 3.9, a read transaction over the control line has overhead. Meaning that the actual pixel hit data read would be about 15% of the total transaction. And, as the data in the internal FIFO is stored in 24-bit wide data words, two read transaction has to be done to read one data word. This will lower the actual data read to about 11%. This data word can, in addition, only exist of, e.g. an 8-bit CHIP HEADER data word, amounting then to a total of about 4% of actual data read. Nevertheless, considering the best case scenario of 11%, the effective data rate over the control interface, operating on 40 MHz, would be 4.4 Mb/s, and in the worst-case scenario 1.6 Mb/s. This data rate is considerably slower than the serial data port 1.2 Gb/s and the parallel data port 320 Mb/s. Besides, utilizing the control interface for data offloading would not make it possible to do data offloading parallel to triggering. This option was for these reason considered a non-optimal solution for data offloading.

Second, the serial data port offers the fastest data offloading option. The serial data port would give an effective data rate of 960 Mb/s, 480 Mb/s or 320 Mb/s, considering the 8b10b encoding. This is also the intended data offloading interface for the ALPIDE chip, meaning that utilizing this interface should give maximum

\[ \text{Utilizing two processors would make this task possible. However, this complicates software development significantly.} \]

An example code where this was done can be found on this thesis GitHub page [59].

\[ \text{The UNICAST READ transaction takes ideally 105 clock cycles, transmitting 1 bit each clock cycle, the data read consist of 16 bits. Therefore, } 16/105 \approx 15\%. \text{ See Appendix D, Figure D.1 for measurement.} \]
performance. As normal FPGA I/O ports generally will not work at these data rates, an FPGA with transceivers would have to be required. However, buying a relatively expensive development board before knowing the exact necessity of the readout system would go against guideline 1. And, as the ZedBoard [62] could be used with the parallel data port, as explained next, this option was put aside.

Third, the parallel data port is meant for inter-chip communication and has two operation configurations, as explained in section 3.4.2. Either it is configured to use 8 data lines operating at 40 MHz or to use 4 data lines operating at 80 MHz. Regardless, the data rate will be 320 Mb/s. The parallel interface consequently offers a relatively low switching speed signal with a relatively high data rate. It actually delivers the same data rate as the serial data port operating at its lowest speed. Thus, this interface gives the option to utilize normal FPGA I/O ports without dropping to the data rate of the control port, namely 4.4 Mb/s. The downside is that a total of 8 data lines need connection, in contradiction to the two differential lines for the serial data port. However, as previously mentioned, most FPGA’s has no problem with 8 I/O ports. Hence, the parallel data port was the interface chosen for the readout system, as this enabled the system to be developed on the ZedBoard [62], also fulfilling guideline 1.

As the data stream coming from the ALPIDE chip has its own defined protocol, a HDL module resembling that of block 2 in Figure 4.3 should be developed. This should be developed in-house to fulfill guideline 3. This sets the foundation for R.4 in Table 4.1. The data stream coming from the ALPIDE chip would be at a data rate of 320 Mb/s. This means that the ZedBoard should be able to handle this data rate without saturating. That is, the ZedBoard should be able to store the data packets, then transmit the packets to a computer without sacrificing ALPIDE’s frame rate or losing data. This sets R.5 in Table 4.1.

4.3 Summary

This chapter discussed the requirements for a single ALPIDE chip readout system. To summarize the outcome of the discussion, each requirement is listed in Table 4.1. The next chapter will develop a readout system which fulfills these requirements.
Chapter 5

Readout System Implementation

This chapter documents the readout system developed in this thesis. First, an overview of the system will be given. Following, each part of the system will be explained in detail. Then, in section 5.5 the verification methodology, as well as some specific verification examples, will be discussed. Lastly, a summary will be given.

5.1 Overview

A picture of the readout system developed in this thesis can be seen in Figure 5.1. The system was designed towards the ALPIDE Carrier card. Here a converter card was built to handle the interface between the ALPIDE Carrier card and the ZedBoard. This converter card, named the PCIe x8 to PMOD card, was developed towards the PMOD connectors of the ZedBoard. In the Zynq-7020 SoC Programmable Logic (PL) two VHDL modules, named Alpide Controller and Alpide Data, were implemented enabling communication with the ALPIDE. These modules were developed as memory mapped modules to the Zynq-7020 SoC Processor System (PS), which was programmed to initiate tests on the ALPIDE chip. The data gathered from the ALPIDE chip was in turn transmitted to a computer, using the ZedBoard’s UART connection. This data was then analyzed on a computer. Measurements done with the readout system will be discussed in section 6.1.

5.2 Hardware

In this section, the readout system’s different hardware components are explained. The readout system is pictured in Figure 5.1. The ALPIDE Carrier card and the ZedBoard will be discussed relatively briefly, whereas the PCIe x8 to PMOD card will be discussed in more detail as this card was developed in this thesis.

5.2.1 ALPIDE Carrier

As the ALPIDE available was attached to the ALPIDE Carrier card, the hardware design was chosen to be implemented towards this card. The ALPIDE Carrier card was developed by CERN, and the schematics and the PCB layout were available at [63]. This made the development towards this card significantly easier. Schematics
5.2. HARDWARE

Figure 5.1: The readout system developed in this thesis. 1: The ALPIDE Carrier card. 2: The PCIe x8 to PMOD card, developed specifically for this thesis. 3: UART connection, used for data transfer to computer. 4: Power connector. 5: The Zynq-7020 SoC. The green development board is the ZedBoard.
Figure 5.2: A block diagram of the PCIe x8 to PMOD board. Only the parallel data lines, DATA[7:0], need level shifting. The control line and system clock is level shifted to the 3.3 V ref by the ALPIDE Carrier board.

and PCB layout can also be found on this thesis GitHub page [59]. The ALPIDE Carrier card uses wire bonding to connect to ALPIDE’s pads and routes those signals out to the PCIe x8 male connector. The parallel data port, that is DATA[7:0], is directly routed from the ALPIDE chip to the PCIe x8 connector. However, the clock line and the control line are both routed through integrated circuits (ICs). These ICs, here two Multipoint-LVDS line Drivers and three Transceivers, are used to convert the bi-directional differential control line (see section 3.4.1), to three single-ended unidirectional lines. The differential clock line is converted to a single-ended line. This makes for an easier connection, as the ICs take care of specific electrical requirements needed for connecting to the ALPIDE chip. Meaning, the I/Os of the Zynq-7020 SoC does not need to function as differential pairs, be bi-directional or have the required voltage level of the ALPIDE’s I/Os. These requirements, as well as the ALPIDE Carrier card’s ICs, are explained in more detail in Appendix B.

5.2.2 PCIe x8 to PMOD

Since the ALPIDE Carrier card had a PCIe x8 connector which was specifically made for the DAQ board in Figure 4.5, a self-made converter had to be made to utilize the ZedBoard. This converter, named the PCIe x8 to PMOD, acts as a converter from the PCIe x8 connector to three PMOD headers. A block diagram of the board can be seen in Figure 5.2. The choice of using the PMOD connectors was made because of their simplicity.

Requirements and Design

The main design requirements for the PCIe x8 to PMOD board are listed in Table 5.1. The bullet points below address each requirement and its respective solution.

- **D.1**: The ZedBoard’s PMOD connectors are all routed to I/O Banks on the Zynq-7020 which are driven to 3.3 V, therefore, a voltage level shifter is needed. This voltage translation has to be between ALPIDE’s 1.8 V CMOS level and
Table 5.1: Design requirements for the PCB design of the PCIe x8 to PMOD board.

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1</td>
<td>Convert between ALPIDE I/Os and ZedBoard I/Os electrical characteristics.</td>
</tr>
<tr>
<td>D.2</td>
<td>Interface to the PL-side of the Zynq-7020.</td>
</tr>
<tr>
<td>D.3</td>
<td>Handle up to 40 MHz switching speeds.</td>
</tr>
<tr>
<td>D.4</td>
<td>Power the ALPIDE from the PMOD connectors.</td>
</tr>
<tr>
<td>D.5</td>
<td>Enable control over ALPIDE’s $V_{BB}$ voltage.</td>
</tr>
<tr>
<td>D.6</td>
<td>Simple PCB design, enabling etching at the UiO electronics lab [64].</td>
</tr>
</tbody>
</table>

The PMOD’s 3.3 V CMOS level. A level shifter is only needed for the parallel data lines, as the clock line and the control line is level shifted by the ALPIDE Carrier board (see Appendix B). The 8-Bit dual-supply configurable voltage translator, SN74AXCH8T245 [65] was chosen. This chip was convenient as it could level shift all 8 data lines. The chip’s propagation delay for the setup chosen was in the range 0.5 ns to 5 ns. For 5 ns that would be 20% of the 25 ns system clock period [65]. However, this was not of major concern as the data sampling of the parallel data lines would be asynchronous to the system clock, which will be further explained in section 5.3.2. Moreover, the chip came in TSSOP package which helped fulfilled requirement D.6.  

- **D.2**: Four PMOD connectors on the ZedBoard interface to the PL-side of the Zynq-7020. Each PMOD connector has 12 pins in total. Eight pins that connects towards the PL, and 2x2 towards ground and power [62]. The control line, clock line and parallel data lines together need 12 pins. If we also add the functionality of turning the power towards the ALPIDE chip on and off, and the clock on and off, we get a total of 14 lines which needs to be connected to a PMOD connector. Two PMOD connectors are then needed. The choice in the design was to use three PMOD connectors. However, the last one was used only for stability. See Appendix E Figure E.4 for the schematic layout.  

- **D.3**: The clock line, control line and the parallel data lines all run at 40 MHz. This requires some termination resistance to ensure signal integrity. In the PMOD headers used in the ZedBoard, no termination resistors are implemented. However, the other PMOD connectors on the ZedBoard uses a termination resistor of 200 $\Omega$. This was tested, and a termination resistor of 200 $\Omega$ was added to the clock and control line. Tests done with respect to transmission integrity are found in Appendix C.  

- **D.4**: The PMOD connectors on the ZedBoard each have two power pins, routed to 3.3 V [66]. As the ALPIDE runs on a 1.8 V supply, a voltage regulator is necessary [43]. Here the low-dropout regulator, TPS773618 [67], was chosen. ALPIDE’s maximum current consumption is not well documented, probably due to the fact that current consumption largely depends on particle hit rate and means of data transmission. However, some papers have noted ALPIDE’s expected power consumption [9], [68]. Using the maximum power estimate,  

---  

1TSSOP stands for “Thin-shrink small-outline package”, and the package has legs which enable hand soldering.
5.2. HARDWARE

Table 5.2: Components used for the PCIe x8 to PMOD circuit.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
<th>Datasheet</th>
</tr>
</thead>
<tbody>
<tr>
<td>SN74AXCH8T245</td>
<td>Voltage level-shifter</td>
<td>[65]</td>
</tr>
<tr>
<td>TPS773618</td>
<td>Low-dropout regulator</td>
<td>[67]</td>
</tr>
<tr>
<td>PCIE-098-02-F-D-RA</td>
<td>PCIe x8 female connector</td>
<td>[70]</td>
</tr>
<tr>
<td>IO pin header</td>
<td>PMOD standard male pin headers</td>
<td>[71]</td>
</tr>
</tbody>
</table>

≈150 mW, from these papers, a back of the hand calculation can be done to estimate the maximum current consumption to:

\[
\frac{P}{V} = I \Rightarrow \frac{150 \text{ mW}}{1.8 \text{ V}} \approx 83 \text{ mA}.
\]

This makes the TPS773618 well in range as it can handle up to 400 mA [67]. It should also be noted that a much lower current consumption is expected, as our setup will not utilize the serial data port, a unit having an expected power consumption alone of 68 mW [68].

- **D.5:** ALPIDE’s $V_{BB}$ voltage is routed to a pin header on the PCIe x8 to PMOD board. This enables external control over the $V_{BB}$ voltage. The pin header, and thereby $V_{BB}$, can also be shorted to ground with a jumper, see Appendix E Figure E.2.

- **D.6:** As the circuit was reasonably simple, it would be an advantage with regards to time and re-design issues, to etch the PCB at the Electronics workshop at the Department of Physics, UiO [64]. This resulted in some limits concerning design and manufacturing. E.g., the PCB had to be a 2-layer design, and all the components should be hand soldered.

The components used in the final design is listed in Table 5.2. In Figure 5.3 and Figure 5.4 the bottom and the top of the assembled PCIe x8 to PMOD circuit board can be seen. The circuit board was designed with the CADSTAR program suit [69]. Schematics and PCB layout can be found in Appendix E.

5.2.3 ZedBoard

The ZedBoard is a development kit for the Xilinx Zynq-7000 All Programmable SoC. The kit was chosen as the Zynq-7000 offered Programmable Logic (PL) and it was reasonably priced. Using an FPGA\(^2\) was, as mentioned in section 4.2, a requirement for the readout system. The Zynq-7000 model on the ZedBoard was a Zynq-7020 [72]. This model had a total of 53,200 Look-Up Tables (LUTs) and offered a dual-core ARM Cortex-A9 processor [73]. Moreover, the ZedBoard also offered enough I/Os towards the PL-side to support the implementation of the modules set by requirements R.3 and R.4 in Table 4.1.

\(^2\)FPGA and PL are used interchangeably throughout this thesis.
5.2. HARDWARE

Figure 5.3: Bottom of assembled PCIe x8 to PMOD circuit board.

Figure 5.4: Top of assembled PCIe x8 to PMOD circuit board.
5.3 Firmware

In Figure 5.5 a block diagram of the top level design can be seen. The design is split in two separate modules: Alpide Data and Alpide Controller. The former module handles the data packets received over the parallel data port and stores those packets in block-ram (BRAM). The latter module implements the communication protocol over the control line. Both modules are extended to be compatible with the AXI4 Interconnect Intellectual Property (IP) [74]. This allows the modules to be connected to the Zynq-7020 PS-side as memory mapped modules. The choice of splitting the design into two distinct modules was made because both modules were self-reliant with respect to the ALPIDE chip. As discussed in section 4.1.1, this was also done for the proposed firmware solution at CERN. Furthermore, two distinct modules made testing easier as each module could be tested separately.

Both modules were written in VHDL, simulated and tested in Questa [75], and implemented in the Vivado Design Suit 2018.2 [76]. The VHDL code for each module can be found in Appendix J and K. The modules were written with the general guideline to be as device-unspecific as possible. That is, the modules should be easily implemented on other platforms. The testbench for each module is described in section 5.5. The next two subsections will discuss each module individually. Section 5.3.3 will then discuss the implementation of these modules on the ZedBoard.

5.3.1 Alpide Controller

The Alpide Controller module implements the communication over the control port. The control port, as discussed in section 3.4.1, follows a protocol defined in the Alpide Operations Manual [43].

All though the protocol was described in the Alpide Operations Manual [43], the Bus Functional Model (BFM) designed by Ola S. Grøttvik [51], [77] was of great guidance during the development of the Alpide Controller module. Grøttvik had designed a BFM together with a simulation with the intent of understanding ALPIDE’s principle of operation [78]. This simulation was designed around a behavioral model of the ALPIDE chip, called the Alpide Light Weight Model (ALWM) [79]. The BFM and the ALWM were in general not synthesizable, but worked well as functional reference during the design of the Alpide Controller module.

Design

The Alpide Controller is split into sub-modules. This is illustrated in Figure 5.6. The Alpide Controller runs on the 40 MHz system clock. Below is a short description of each sub-module. The VHDL code can be found in Appendix J.

---

3BRAM is memory located within the PL fabric. The Zynq-7020 has 4.9 Mb of BRAM [73].
4AXI stands for “Advanced eXtensible Interface” and makes the development of 3rd partner IP’s easier [74].
5A BFM, in the world of VHDL, is a set of procedures and functions simulating bus transactions. The BFM is usually non-synthesizable code.
Figure 5.5: A block diagram of the Zynq-7020 layout with the Alpide Data and Alpide Controller modules implemented. Clock lines are marked with the clock frequency.

Figure 5.6: An illustration of the different sub-modules in the Alpide Controller module.
5.3. FIRMWARE

Master

The master sub-module takes care of the actual writing and reading of the control line. The sub-module is implemented as a Finite State Machine (FSM). Since the ALPIDE chip will read the control line on the rising edge of the system clock the FSM changes state on the falling edge. This is to ensure a stable signal during write transactions. The reading of the control line is, however, done on the rising edge as this is the expected protocol standard [43].

Controller

The controller sub-module implements the correct transaction format corresponding to an OPCODE, see Figure 3.9. The sub-module is implemented as a FSM and changes the master sub-module states as a “marionette” to implement the correct transaction format. That is, the controller module decides if the master module should go into a write state or a read state.

AXI4 interface (Alpide Controller)

The Alpide Controller module was wrapped with an AXI4 interface, allowing the module to be memory mapped to the processor. The memory mapped registers can be found in Appendix F. Vivado generates the AXI4 interface VHDL code following the AXI4 interface standard [80]. This code is therefore not discussed in this thesis. However, it should be noted that some user logic was added to the generated code. E.g., user logic was added to ensure valid clock domain crossing, as the Alpide Controller module ran on a 40 MHz clock and the AXI4 interface on a 240 MHz clock. The full AXI4 interface code can be found on this thesis GitHub page [59].

As illustrated in Figure 5.6 two signals were also added to the AXI4 interface module: ALPIDE enable and System clock enable. The former signal was routed directly to the enable pin of the Low-dropout regulator on the PCIe x8 to PMOD board, facilitating control of ALPIDE’s power supply. The latter signal was routed directly to one of ALPIDE Carrier’s ICs, facilitating control of the clock input toward the ALPIDE chip (see Appendix B).

5.3.2 Alpide Data

The Alpide Data module receives bytes over the parallel data port. These bytes are part of data words, as discussed in section 3.4.2. The parallel data port (DATA[7:0]) is clocked by the ALPIDE chip on the rising edge of the system clock. However, as the system clock originates from the Zynq-7020, the parallel data received will have an unavoidable skew due to the propagation delay between the ALPIDE chip and the Zynq-7020. This delay is implemented in the ALWM, the behavioral model of the ALPIDE chip, and can be seen in Figure 5.8. The voltage translator implemented in the PCIe x8 to PMOD board, see section 5.2.2, is also expected to increase this skew further. To both ensure signal integrity and not to favor any particulate setup with respect to the distance between the ALPIDE chip and the Zynq-7020 Soc, the Alpide Data module was designed to receive data asynchronously. This would improve modularity and signal integrity.
5.3. Firmware

Figure 5.7: An illustration of the different sub-modules in the Alpide Data module.

Figure 5.8: A snapshot of a simulation with the ALWM and Alpide Data module. As illustrated the parallel data port has a transient value of 0x5F. The Synchronizer sub-module ensures that no transient data is read. A skew of the parallel data port can also be seen, here the parallel data port changes after the rising edge of the system clock.

Design

An overview of the Alpide Data module design can be seen in Figure 5.7. The Alpide Data module runs at a 160 MHz clock. This gives the module 4 rising edges each parallel data byte. Below each sub-module of the Alpide Data module is explained. The VHDL code for each component can be found in Appendix K.

Synchronizer

See Figure 5.9 for a flowchart diagram of the Synchronizer module. The Synchronizer primarily waits for a bit change on the parallel data line. A change indicates that a new data word is transmitted from the ALPIDE chip. When that change is stable for two clock cycles, the byte gets sent to the Filter. This is done to avoid any transient data reading. Transient data is expected, and the ALPIDE chip will sometimes write 4-bits of the byte before the other 4-bits [43]. This is also implemented in the ALWM and is illustrated in Figure 5.8. Moreover, in the special case where two equal consecutive bytes are transmitted from the Synchronizer module, the Filter will check if the ALPIDE chip normally unequal due to the data word predefined prefix bit string. However, in the case where, e.g. the bunch counter equals that of the region header, two equal bytes will be sent over the parallel data port.

Filter

The Filter makes sure that only valid data words get written to the FIFO. That is, data words listed in Table 3.3. The Filter also ensures that not multiple data words get written to the FIFO. In the case where two equal consecutive bytes gets transmitted from the Synchronizer module, the Filter will check if the ALPIDE chip

---

6This is indeed a special case. Looking at the valid data words in Table 3.3 the bytes are normally unequal due to the data word predefined prefix bit string. However, in the case where, e.g. the bunch counter equals that of the region header, two equal bytes will be sent over the parallel data port.
intends this. The Filter does so by looking at the type of data words transmitted before and after. For example, if the Synchronizer sends two consecutive REGION HEADER data words, the Filter will know this was an error as this is not the protocol standard. Moreover, the Filter will make sure that when the ALPIDE is in IDLE, only one IDLE word gets written to the FIFO, not filling the FIFO with IDLE words.

**Concatenate**

The Concatenate component concatenates the bytes to 32-bit data words before writing them to the FIFO. The bytes are saved in little-endian format.

**FIFO**

The FIFO is written in VHDL and is written after the template found in the Vivado Synthesis User Guide [81, page 116], which utilizes BRAM on the FPGA. This VHDL template utilizes BRAM when the code is synthesized and implemented. However, the template has only been tested on the Zynq-7020. The current implementation sets the BRAM in the FIFO to be 65535 deep, that is \(65535 \times 32b \approx 260\) kB. This utilizes about 45% of the available BRAM in the Zynq-7020, see Table 5.3. The size of the FIFO can be changed by adjusting the generic of the FIFO sub-module (see Appendix K), but a maximum of 560 kB are available in the Zynq-7020. The FIFO controller was implemented after the design example by Pong P. Chu in [82, page 279].

**AXI4 interface (Alpide Data)**

The Alpide Data module was wrapped in an AXI4 interface. The memory mapped registers can be found in Appendix G. As mentioned earlier, the VHDL code for the AXI4 interface is generated by Vivado. Nevertheless, some user logic was added to the AXI4 interface, in particular, user logic handling the clock domain crossing and the interrupts. Two interrupts were added to the Alpide Data module. These were

---

**Figure 5.9: Flowchart of the Synchronizer sub-module.**

---
added to indicate to the Zynq-7020 PS when the FIFO was half-full and full (see Appendix G).

5.3.3 Implementation

The Alpide Controller module and the Alpide Data module were implemented in the Zynq-7020 PL using the Vivado Design Suit 2018.2. Both modules were packaged as custom IPs after the Xilinx user guide [83]. This step is strictly not necessary to implement the modules in the Zynq-7020 PL, but was done to make future usage of the modules more consistent, at least for Xilinx users. The modules, as well as the packaged IPs, can be found on this thesis GitHub page [59].

The Block Design functionality of the Vivado Design Suit was used during implementation. In Figure 5.10, the final block diagram design of the Alpide Controller and Alpide Data module can be seen. As seen from the figure, the control line is split into three signals: DCTRL_IN, DCTRL_EN, and DCTRL_OUT. These signals are converted to the bi-directional control port by the ALPIDE Carrier board, see Appendix B. Moreover, as previously noted, the signals ALPIDE_EN (ALPIDE enable) and MCLK_EN (System clock enable) were added to the Alpide Controller module. Respectively controlling the power towards the ALPIDE chip and the enabling of the system clock.

The Alpide Data module in Figure 5.10 has two interrupts, named half_full_intr and full_intr. These interrupts are routed directly to the interrupter input of the Zynq-7020 PS. The parallel data port (DATA[7:0]) is routed directly to the Alpide Data module.

AXI4 Interconnect speed

The ALPIDE chip will be outputting data at a maximum of 320 Mb/s. Thus, the FIFO needs to be at least able to be popped at this rate to prevent overflow. Since the Alpide Data module was wrapped in an AXI4 interface, the AXI4 Interconnect speed becomes crucial to obtain the wanted readout rate. For this reason, the AXI Interconnect was connected to the maximum clock frequency that Vivado would allow during implementation, which was 240 MHz. Ideally, this would allow to pop the FIFO at a rate of 32-bit/(1/240 MHz) = 7680 Mb/s, however, as will be discussed in Chapter 6 the final readout speed is much slower. This is also expected as there will be overhead in the AXI4 protocol as well as in the drivers handling the readout.

Resources and Constrains

Constraints were applied to the Vivado project to set the correct pin mapping and correct clock handling. The constraint file can be found on this thesis GitHub page [59]. Table 5.3 shows the resources used for the final design. This table illustrates the minimum resources needed for future systems using the Alpide Controller and Alpide Data module.
Table 5.3: Utilization report post-implementation for Zynq-7020. Data was taken from Vivado for the final design.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>1033</td>
<td>53200</td>
<td>1.94</td>
</tr>
<tr>
<td>LUTRAM</td>
<td>62</td>
<td>17400</td>
<td>0.36</td>
</tr>
<tr>
<td>FF</td>
<td>1422</td>
<td>106400</td>
<td>1.34</td>
</tr>
<tr>
<td>BRAM</td>
<td>64</td>
<td>140</td>
<td>45.71</td>
</tr>
<tr>
<td>IO</td>
<td>14</td>
<td>200</td>
<td>7.00</td>
</tr>
<tr>
<td>BUFG</td>
<td>5</td>
<td>32</td>
<td>15.63</td>
</tr>
<tr>
<td>MMCM</td>
<td>1</td>
<td>4</td>
<td>25.00</td>
</tr>
</tbody>
</table>

Figure 5.10: A screen-shot of the Block Diagram implementation of the Alpide Controller module and the Alpide Data module.
5.4. SOFTWARE

This section will discuss the software that was programmed on the Zynq-7020 ARM-based processor. Section 5.4.1 will focus on the Hardware Abstraction Layer (HAL), that is the Alpide Controller and Alpide Data hardware-specific drivers. In section 5.4.2, some special software considerations regarding the readout system are noted. But first, a short overview of the software application will be given.

The software application was developed in Xilinx SDK 2018.2 [84] as a standalone application. Meaning that no operating system was used. The chosen language was C [85]. The basic functionality of the application was to initiate tests on the ALPIDE chip and transmit the received data from the ALPIDE over UART to a computer. A flowchart diagram of the software application can be seen in Figure 5.11. The program waits for bytes received over UART from a computer and when a byte is received the program initiates the corresponding test for that byte. If no bytes are received the program checks the Alpide Data module’s FIFO, if the FIFO is not empty, the program will pop the FIFO and transmit the popped values to the computer over UART. The data is then analyzed on the computer, as will be explained in section 5.4.3. The data transmitted to the computer is the raw data, that is the data words shown in Table 3.3. The choice of not decoding the data words to, for example, pixel hits consisting of column and rows before transmitting the data was done to get more flexibility when analyzing the data. This is further discussed in 5.4.3.

Figure 5.11: A simplified flowchart diagram of the standalone application running on the Zynq-7020 ARM-based processor.
5.4. SOFTWARE

![Figure 5.12: A block diagram showing the two key Python programs. The zedboard_communication.py and the alpide_data_analyzer.py.](image)

5.4.1 Hardware Abstraction Layer (HAL)

As the Alpide Controller module and the Alpide Data module are memory mapped modules, hardware-specific driver functions were created to interface with these modules. This enabled easier communication between the modules and the software written on the Zynq-7020 ARM-based processor. These functions were created from the register map of the Alpide Controller and Alpide Data module, found in Appendix F and G. The driver functions can be found in Appendix I and H.

5.4.2 Special considerations

When capturing snapshots with the ALPIDE chip, the chip should first be initialized in a certain manner. How this is done is explained in [43, page 75], and was done during the initialization of the readout system. The code doing so can be found on this thesis GitHub page [59]. However, since the readout system in this thesis used the parallel data port, which is not the intended data offloading port, a slight adjustment to the initialization procedure was needed. As mentioned in section 3.4.2, each ALPIDE connected to the parallel data port transmits data packets in turn to the Outer Barrel Master. But for the readout system in this thesis, we wanted the ALPIDE to transmit data over the parallel data port continuously, and not wait for turn. This was achieved by configuring the Control Management Unit and the Data Management Unit in the ALPIDE [43]. This can be seen done in code Listing 5.1, here the driver function of the Alpide Controller module is used to write to ALPIDE’s configuration register 0x0010.

```python
1 writeALPIDE(chipID, 0x0010, 0x0030);
2 /* Writes initial token to 1, disables Manchester encoding and double
3  * data rate. Setting previous chip ID to 0x0 (my chipID), this way
4  * ALPIDE will continuously transmit over parallel data port.
5 */
```

Listing 5.1: Part of initialization procedure.

5.4.3 Python programs

As the standalone application transmitted raw data words to the computer, decoding was needed to analyze the data as pixel hits. This decoding was done in Python [86]. A Python program was written to initialize tests over UART and receive the data. The two key Python programs, the zedboard_communication.py and the alpide_data_analyzer.py, are illustrated in Figure 5.12. The former program handles
the UART communication between the ZedBoard and the computer. The latter program decodes the raw data words into pixel hit data. That is, it saves the data in a text file with a format of column, row and hit id. Here the column and row is the address of the pixel hit, and the hit id is the number of the received data packet. Since each data packet corresponds to one frame, all the hits with the same hit id has happened during the same frame. Both programs, together with examples of usage can be found on this thesis GitHub page [59].

As mentioned earlier, the choice of not decoding the data on the Zynq-7020 SoC was done also to be able to analyze the raw data. For example, this allowed the analysis of ALPIDE’s clustering functionality, discussed in section 6.2.1.

5.5 Verification

Throughout the development process, a large emphasize was put towards the verification of the Alpide Data and Alpide Controller module. In this section, the verification of these modules is discussed. But first, as design and verification require some methodology, the methodology adopted in this thesis is described. Then in section 5.5.2 and 5.5.3 the testbenches for the Alpide Controller and Alpide Data module is described respectively. Lastly, in section 5.5.4, some of the tests done on the readout system after implementation are discussed.

5.5.1 Methodology

In all RTL designs, it is essential to conduct comprehensive verification. This should be done parallel to designing as detecting and resolving design defects are generally more time consuming later in the development phase. In Figure 5.13, an illustration of the design flow used in this thesis is shown. As illustrated the testbench and the RTL design is developed in parallel.

The general idea of breaking each module into sub-modules and then testing the sub-modules individually was used throughout the development. This makes for both easier RTL design and testbench design. The testbenches discussed later in this section are the final testbenches. That is, the testbenches testing the Alpide Controller and Alpide Data modules. Following is a short description of key methods/components used during verification of the Alpide Controller and Alpide Data modules.

UVVM

As the testbench and the RTL design grow it becomes important to have an easily extensible and maintainable testbench. For this reason the Universal VHDL Verification Methodology (UVVM) library was utilized [87]. UVVM is a library for making structured VHDL-based testbenches developed by Bitvis [88]. The UVVM library offers good error detection logging, making a self-checking testbench easier.

---

7A testbench is a program, usually written in VHDL or (System) Verilog, to verify the RTL design.
5.5. VERIFICATION

![Diagram of design flow](image)

**OSVVM**

As it is hard to think of all the possible tests one can do in a testbench (or actually implementing all possible test), a good testbench should specifically test some corner cases and then do extensive testing randomly. For this reason the Open Source VHDL Verification Methodology (OSVVM) library was utilized [89]. OSVVM offers intelligent random coverage [89], accomplishing complete functional coverage faster.

**Alpide Light Weight Model (ALWM)**

In order to thoroughly test each module, both testbenches were tested with the ALWM. The ALWM is a digital simulation model of the ALPIDE chip created by [79]. The model was created to aid firmware/software development [79]. Pixel hits are generated randomly to imitate radiation. During verification and design, this module worked as an assistive module to verify correct functionality. As can been seen from the testbench descriptions below, the ALWM was used in both testbenches.

**Support procedures**

Packages were written in VHDL to help the modularity of the testbench design. These packages contain a set of procedures making, for example, reading and writing to the Alpide Controller easier. Thereby increasing the readability and flexibility of the testbench design. These packages can be found on this thesis GitHub page [59].

5.5.2 Testbench: Alpide Controller

In Figure 5.14 a block diagram of the final testbench used for verification of the Alpide Controller module is shown. In the figure the Test sequencer initiates different transactions between the Alpide Controller module and the ALWM. A simplified behavior model of the ALPIDE Carrier card was made to reflect the readout system. The data returned by the Alpide Controller is then controlled with the expected data return of the current test sequence. The testbench can be found at [59].

The Test sequencer first tested specific corner cases, e.g., resetting the ALWM and reading back reset values from the configuration registers. Then the Test sequencer ran a functional coverage test writing random values to random configura-
5.5. Verification

Testbench: Alpide Controller

Support procedures

Test sequencer

DUT: Alpide Controller

ALPIDE Carrier
behaviour model

ALWM

Data return

Controlled

Figure 5.14: Block diagram of the testbench utilized to verify the Alpide Controller module (Device Under Test (DUT)).

tion registers and reading those values back. If the value written was not the same as the Data return an error would be asserted.

5.5.3 Testbench: Alpide Data

The final testbench for the Alpide Data module can be seen in Figure 5.15. This testbench actually verifies the whole readout system, and it was, therefore, important that the Alpide Controller module was verified in advance, isolating potential errors to the Alpide Data module.

The Test sequencer initiates external triggers towards the ALWM to take snapshots, as discussed in section 3.3.1. These snapshots are sent as data packets over the parallel data port (DATA[7:0]) which are received by the Alpide Data module. The Test sequencer then pops the data packets received by the Alpide Data module and store those in a text file. This text file then contains the raw data words discussed in 3.4.2. The file is then decoded, as described in section 5.4.3, and if no errors are found a hitmap is produced. This hitmap represents randomly generated hits produced by the ALWM.

5.5.4 Physical testbench

After both modules, Alpide Controller and Alpide Data, were verified with the testbenches mentioned above, both modules were implemented on the Zynq-7020 SoC. Then, to verify the whole readout system, tests were written in C on the standalone application. These tests would function much in the same way as the testbenches explained above, however, on the physical system. This was done to test for potential errors originating from, e.g., signal integrity, constrain issues or other unforeseen issues.

The system was able to read and write over the control port, and the lines were found to handle switching speeds of 40 MHz (see Figure D.1 for an oscilloscope measurement of a read transaction over the control port). Data packets received over the parallel data port were received without errors. This data was transmitted over
Figure 5.15: Block diagram of the testbench utilized to verify the Alpide Data module (Device Under Test (DUT)). The hitmap is produced from random generated hits by the ALWM.
5.6. SUMMARY

Figure 5.16: A hitmap plotted from data taken with the readout system. The ALPIDE was pulsed in a chess pattern (only visible by the enlarged square). The plot reveals dead pixels (marked by red circles) and dead columns. The dead columns addresses are: 16, 17, 22, 23, 448 and 449.

UART and successfully analyzed using the Python program described in section 5.4.3. Moreover, the ALPIDE chip was successfully powered from the ZedBoard using the voltage regulator on the PCIe x8 to PMOD board.

To further validate the readout system, specific pixels in the ALPIDE were digitally pulsed over the control port.\(^8\) This would detect if there existed discrepancies between the pulsed pixel addresses and the received data from the parallel port. No discrepancies were found. However, this test showed that the ALPIDE used in this thesis had dead columns and dead pixels. That is, areas on the chip that could not register pixel hits. This is shown in Figure 5.16, here a chess pattern has been pulsed to the ALPIDE.

Further, the readout system was tested under radiation. This is documented in section 6.1. The data from the radiation tests were also compared to radiation tests with the DAQ board. Here the data from the DAQ board also showed no hits at column address: 16, 17, 22, 23, 448 and 449, see Figure 5.17. This indicated that the DAQ board and the readout system developed in this thesis handled pixel hit data in the same manner.

5.6 Summary

This chapter documented the development of the readout system towards a single ALPIDE chip. The requirements in Chapter 4 Table 4.1 served as guidelines for the development. To fulfill the requirements two VHDL modules were developed and implemented, the *Alpide Data* and *Alpide Controller* module. This development involved extensive testing and testbenches were consequently written in VHDL to verify each module. Further, one converter card was made specifically to connect the ALPIDE chip and the ZedBoard, namely the *PCIe x8 to PMOD*. This task involved PCB design and general knowledge of signal integrity for valid data transmission. On Zynq’s ARM-based processor a standalone application was programmed, this

\(^8\)Digitally pulsed pixels will forcedly write a hit to their MEB. How to puls pixels can be found on this thesis GitHub page under the software folder [59].
Figure 5.17: A heatmap plotted from data taken with the DAQ board under gamma radiation. Data from the DAQ board also shows dead columns.

involved creating hardware drivers for the two VHDL modules as well as general embedded programming. At last, a Python program was created to analyze the raw data transited from the ALPIDE to a computer. The Python program would convert the transited raw data to pixel hits with column and row addresses, which could further be analyzed.

Overall, the implementation of the readout system, can be said to have been a success. Reason being the system worked, and no functional errors were found. The performance of the system will be discussed in the next chapter.
Chapter 6

Results

To evaluate the accomplishment of this thesis’ chief objective, namely to develop a readout system for doing radiation measurements with the ALPIDE, radiation measurements were done with the readout system. This is documented in section 6.1. In section 6.2 the performance of the developed readout system will be discussed. This discussion will highlight restrictions, found during measurements in section 6.1 and during implementation in Chapter 5, with the readout system. Consequently, section 6.2 aims to fulfill this thesis second-order objective: to highlight general specification needs for readout electronics. Lastly, in section 6.3 a summary will be given.

6.1 Radiation measurements

This section documents the objective, methods and results related to the radiation measurements done with the readout system developed in this thesis. The measurements were done at the UiO at the Electronics section’s lab [90].

6.1.1 Objective

The objective of the measurements was to demonstrate the feasibility of doing lab experiments with the developed readout system in this thesis. This was done by measuring cluster size, defined as the number of pixel hits neighboring, for different distances between the radiation source and the ALPIDE. The measurements were also done to obtain data which were used in the discussion in section 6.2.

6.1.2 Equipment

The equipment can be seen in Figure 6.1. The ALPIDE was placed directly under an Am-241 source, marked by outline 1 in Figure 6.1a. The source had a nominal activity of 5 kBq and would radiate alpha particles with energies at roughly 5.5 MeV [91]. A knob was used to adjust the distance between the ALPIDE chip and the source, marked by outline 2 in Figure 6.1a. The readout from the ALPIDE chip was done using the readout system developed in this thesis. The whole setup, seen in Figure 6.1a, was closed off in an aluminum box to prevent disturbance from other sources of radiation, this can be seen in 6.1b. The stand holding the source and
6.1. Radiation Measurements

Figure 6.1: Figure (a) shows the test setup used. Outline 1 was where the Am-241 source was placed. Outline 2 shows the knob to adjust the distance between the source and the ALPIDE chip. Figure (b) shows the aluminum box being closed with the test setup inside. The box was used to protect against other sources of radioactivity.

The aluminum box was designed by UiO's instrumentation lab on request by Qasim Waheed, a PhD candidate at UiO [92].

6.1.3 Method

The strobe length was set to 1 ms. Using equation 3.2, with $\alpha$ estimated to $\alpha = 1/2$, this would give about 2 particle hits per frame. Here $\alpha$ was largely overestimated, hence we would expect to see less than two alpha particles actually hitting the ALPIDE chip each frame. By setting the strobe length long, we would capture the full evolution of the cluster [93, page 49]. That is, pixel hits happening at different times but from the same particle hit will be captured in the same frame due to the long strobe length. Capturing a pixel at the beginning or end of the strobe signal is also not wanted, as this might only capture the cluster partially. Therefore, by using a longer strobe signal the rate of partial clusters will be reduced. E.g. if the pulse charge of a cluster stays for about $8 \mu$s [93], the chance of this being captured at one of the strobe edges is $2 \times 8 \mu$s/1 ms = 1.6%. We can therefore expect 1.6% of the cluster sizes to not be their true size, an amount considered tolerable. Moreover, the substrate bias of the ALPIDE chip was connected to ground, that is $V_{BB} = 0 \text{ V}$.

A total of 100,000 triggers were sent to the ALPIDE chip. The chip was configured in triggered mode using external triggering, see section 3.3. This would ensure that partial data was not captured if the trigger rate got too high and would allow the readout system to control the trigger rate. The rate would be changing as the readout system transmitted the raw data to a computer for each frame before sending a new trigger command to the ALPIDE. This was done to ensure that the internal
memory of the Zynq-7020 would not overflow because of the AXI Interconnect speed or the UART connection speed, this issue will be discussed in section 6.2. Moreover, the STROBE gap was made sure to always exceed 5 µs, ensuring that no hits would be captured twice, discussed in section 3.3.1.

Measurements were first done without the source present. This was done to remove potential faulty pixels, that is pixels showing hits regardless of irradiation or not. Faulty pixels were then masked in the ALPIDE chip. Measurements were then done with the source. The source started at a distance of 5 mm from the ALPIDE chip and was then moved in one millimeter steps to a distance of 20 mm, amounting to a total of 16 measurements. Each measurement was done with and without clustering, giving data for the discussion in section 6.2. For each frame, the cluster size was analyzed in Python. The Python code would count the number of neighboring pixel hits in a cluster and store all the clusters and their sizes in a list. The cluster size distribution for each distance could then be analyzed. The Python code can be found at this thesis GitHub page [59].

To supplement the discussion, a simulation of the experiment was done in SRIM [21]. SRIM is a simulation tool which calculates features of the transport of ions in matter [21]. The tool is limited and the simulations were mainly done to calculate the expected energies of the alpha particles. This is discussed in more detail in section 6.1.5.

6.1.4 Results

In Figure 6.3 a plot of all the pixel hits at a distance of 5 mm is shown. All the frames are laid on top of each other and the color represents the recurrence of pixel hits. Figure 6.4 shows the cluster size distribution for the hitmap in Figure 6.3. An example of two clusters can be seen in Figure 6.2. As can be seen from Figure 6.4 the cluster size distribution varies from 1 to 29 pixel hits. The distribution seems to have two different normal distributions, one around cluster size 21, and one around 7. Also, a large portion of single pixel hits was observed. This in total gives the distribution a large standard deviation. This can be observed in all the measurements for the different distances, see Appendix L.

Using the mean cluster size from each measurement, the mean cluster size was plotted versus distance between the source and the ALPIDE. This can be seen in Figure 6.5. The error bars are the standard deviation of the cluster size distribution. This plot suggests that there is a relation between the cluster size and the distance between the ALPIDE chip and the Am-241 source.

Lastly, the hit recurrence was plotted versus column/row pixel address. This is shown in Figure 6.6. The figure shows that the source was located somewhere above pixel address: column = 445, row = 290. The figure also shows the dead columns at column address 448 and 449. And, as can be seen from the plot some pixel hits was registered in these column addresses, indicating that some pixels in the columns are partially working. Nevertheless, the dead column was situated closed to where the ALPIDE received the most alpha particles. This could affect the cluster size distribution, as clusters in this area would not be their true size.

1Masked pixels will not be latched into ALPIDE’s MEB. How to mask pixels can be found on this thesis GitHub page under the software folder [59]. The faulty pixels can be found in Table L.1 in Appendix L.
Figure 6.2: Figure (a) shows a cluster of size 12. Figure (b) shows a cluster of size 24.

Figure 6.3: Hitmap for 100,000 triggers. The colorbar represents the hit recurrence. The measurement was taken with a distance of 5 mm between the Am-241 source and the ALPIDE chip. The stripe in the middle is from dead pixels in column 448 and 449.
Figure 6.4: Cluster size distribution for 100,000 triggers. The mean cluster size value is, $\mu = 14.20$, and the standard deviation is, $\sigma = 7.99$. The measurement was taken with a distance of 5 mm between the Am-241 source and the ALPIDE chip.

Figure 6.5: Mean cluster size as a function of distance between the Am-241 source and the ALPIDE chip. The error bars are the standard deviation of the cluster size distribution.
Figure 6.6: Hit recurrence versus column/row pixel address. The spike in the column hit recurrence plot is from the dead columns 448 and 449. The plot suggests that the Am-241 source was placed somewhere above pixel address: column = 445, row = 290.

6.1.5 Discussion

Considering the results from the measurements, a relation between the distance from the source and the cluster size was shown, see Figure 6.5. The cluster size is expected to decrease as the distance from the source increase. This stems from the fact that alpha particles will lose energy traveling through air, thus depositing less energy inside the ALPIDE. In Figure 6.7a, the energy of alpha particles with respect to distance from the source is plotted. E.g. after a distance of 5 mm the alpha particles will have an energy of 5059 keV, having lost $5.5 \text{ MeV} - 5059 \text{ keV} = 441 \text{ keV}$ to air. Further, knowing that the ALPIDE is about 50 $\mu$m thick [12], we know that all the remaining energy will be expended inside the ALPIDE chip, see Figure 6.7b. This would suggest that the created charge inside the ALPIDE can be calculated by equation 2.11, here $E_r$ would be the energy after air in Figure 6.7a. And, as the energy of the alpha particles after air in Figure 6.7a shows a relative linear relation, one could expect the cluster size to also vary linearly with the distance. However, as seen in Figure 6.5, the mean cluster size decrease more rapidly as the distance increase. One explanation for this could be that the ALPIDE has a metal layer of interconnects, which is not sensitive to radiation. Thus only a portion of $E_r$ will contribute to charge collection. This portion is illustrated by the shaded regions in Figure 6.7b. Using equation 2.5 integrating only over the shaded region, the energy deposited, which leads to charge collection can be calculated. This energy will equal the energy of the alpha particle after it has traversed the aluminum layer in the ALPIDE. The alpha particle energy after aluminum can be seen in Figure 6.7a by the dashed lines. As seen the dashed line decays slightly more rapid as the distance increase, and could help explain the slightly more steep decrease in cluster size as
6.1. RADIATION MEASUREMENTS

When looking at the cluster size distribution in Figure 6.4 one can see spikes of more frequent cluster sizes. These spikes all happen on even cluster sizes. This effect was also observed by a recent master thesis [94], here also even cluster sizes were shown to appear more frequent. The effect can be seen in all cluster size distributions, see Appendix L, and would suggest that the ALPIDE favor even cluster sizes. A closer investigation of this effect could be interesting.

Lastly, it should be emphasized that this experiment is not ideal if one wants to study the energy of alpha particles with respect to cluster size. The Am-241 source will for once radiate alpha particles in a spectrum of energies [91], this together with random fluctuations in air and different incident angles will give rise to a range of different deposited energies for the same distance. The SRIM simulations did not include different incident angles and radiation energies. The large uncertainty in the energy deposition can be confirmed by the large standard deviation in the cluster distributions, see Appendix L. Moreover, as was suggested in Simon K. Huiberts’ master thesis [93], the cluster size may also vary depending on where the particle hit on the pixel, which we have no control over in this experiment.

6.1.6 Conclusion

The readout system developed in this thesis proved to be sufficient for the radiation tests done in this section. All measurements were successfully captured, and no apparent faults were detected, e.g., from transmission integrity, firmware defects, or programming errors. The readout system, however, used a significant amount of time doing each measurement. This originated from the slow UART connection and the choice of waiting for the frame being transmitted to the computer before initiating a new trigger. As will be discussed in the section 6.2, this was a flaw with the readout system and would put limits to the frame rate the ALPIDE could operate at.

The mean cluster size from the measurements corresponded with the expected results, namely that the mean cluster size would decrease with respect to an increase in distance. However, the large portion of single pixel hits and the more frequent even cluster sizes were not expected. Both of these results would be interesting to investigate further.
Figure 6.7: Figure (a) shows the energy of alpha particles at distances from the source. The dashed line is the energy after the alpha particle has penetrated 10 µm of aluminum (Al). The energy deposition of the diamond/circle mark is shown in figure (b). Figure (b) shows the stopping power versus depth inside the ALPIDE. The ALPIDE was modeled with a 10 µm Al layer and a 40 µm Si layer. The shaded regions are the energy deposited in Si, the sensitive layer. Data is from simulations in SRIM with 1000 alpha particles [21].
Table 6.1: Specific requirements for the readout system which were not fully met or verified.

<table>
<thead>
<tr>
<th>#</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.5</td>
<td>AXI4 Interconnect and UART causes bottleneck.</td>
</tr>
<tr>
<td>R.10</td>
<td>$V_{BB}$ voltage control not verified.</td>
</tr>
</tbody>
</table>

6.2 Readout system evaluation

This section documents the performance of the readout system and highlights general aspects concerning requirements for readout electronics. The section discusses the developed system, here restrictions with the current design will be highlighted giving recommendations for future developments.

6.2.1 Requirements discussion

Two of the requirements listed in Table 4.1 were not fully met or verified. These requirements are listed in Table 6.1. Requirement R.10 was not verified as the ALPIDE Carrier card available did not enable control of the $V_{BB}$ voltage. That is, the $V_{BB}$ voltage was soldered to ground on the ALPIDE Carrier card. A re-solder of this connection was not prioritized due to time constraints. Requirement R.5 was not met as the ZedBoard could not handle a continuous readout speed of 320 Mb/s from the ALPIDE. This became apparent under the radiation measurements in section 6.1, here a solution which reduced the frame rate was chosen to prevent data loss.

Requirement R.5

Requirement R.5 highlighted some important general aspects concerning development towards a readout system for the ALPIDE chip. For the readout system in this thesis, R.5 meant that the ZedBoard needed to handle 320 Mb/s from the ALPIDE. As discussed, the AXI4 Interconnect speed was a concern with regards to this. To check how fast the Zynq-7020 was able to pop values from the FIFO, an experiment was done to check the average time used for a single pop. This was done using Zynq’s Cortex A9 private timer, here 500,000 pops from the FIFO were done giving an average speed of $\approx$74 Mb/s. This was below the acceptable rate of 320 Mb/s, and consequently, a FIFO overflow would occur if the ALPIDE transmitted data continuously. Figure 6.8 illustrates the different data rates. A solution to prevent overflow would be to lower ALPIDE’s frame rate, thereby lowering the transmitted effective parallel data port speed. This was done in section 6.1. However, this means that the readout system will restrain ALPIDE’s frame rate, which is unwanted.

A solution to this problem would be to send a train of triggers, here a train would be a sequence of consecutive triggers which all would be stored in the FIFO. Then afterward the FIFO could be popped and data transmitted over UART. The AXI4 Interconnect speed would not limit these triggers as they all would be stored in the FIFO. The frame rate for these triggers could, therefore, be as high as the ALPIDE chip could handle (as discussed in [48]). When using this method, it is interesting to know the average data size one would expect for a certain amount
of triggers. As this will set the maximum triggers one can have in a train. This will naturally also depend on the average pixel hits per frame. Using data from measurements done in section 6.1, when clustering was disabled and enabled, the data size per frame was plotted with respect to pixel hits per frame. As can be seen from Figure 6.9, these two variables has a linear relation. Doing a linear regression of the data points when clustering is enabled gives:

\[ y_c = 0.7x + 4.6 \] (6.1)

Here \( y_c \) is measured in B/frame and \( x \) in hits/frame. Setting \( x = \alpha RS_l N_{cz} \) (6.1) from equation 3.2 gives the relation:

\[ y_c = 0.7(\alpha RS_l N_{cz}) + 4.6 \]

Since each trigger is honored with one frame in the ALPIDE, multiplying this by the number of triggers gives:

\[ M_c = y_c \times \text{(triggers)} = \left[ 0.7(\alpha RS_l N_{cz}) + 4.6 \right] \times \text{(triggers)}, \] (6.2)

where \( M_c \) is the memory size in bytes when clustering is enabled. This equation gives a relation between the needed memory for a certain amount of triggers, under a known radiation environment. From this, we can calculate the maximum triggers we can have in a train. For example, let us say that we can maximum expect: \( \alpha RS_l N_{cz} = 100 \) hits/frame. Then, for the FIFO in the Alpide Data module, with memory size 260 kB, a maximum of:

\[ \text{triggers} = \frac{M_c}{0.7(\alpha RS_l N_{cz}) + 4.6} = \frac{260 \text{ kB}}{0.7 \times 100 + 4.6} \approx 3228, \] (6.3)

can be in a train. This calculation is done with data from a specific radiation environment, and deviations is expected for other environments. For example, as seen from Figure 6.9, a given hits/frame can give different B/frame. This will typically come from different cluster sizes, causing different data compression, see Figure 6.10. However, by doing the same calculations with clustering disabled one would get the worst scenario estimate, and equation 6.2 would then become:

\[ M_c = \left[ 3.1(\alpha RS_l N_{cz}) + 3.7 \right] \times \text{(triggers)}, \] (6.4)

where \( M_c \) is the memory size in bytes when clustering is disabled. Using this equation the maximum triggers in a train, calculated as in equation 6.3, would result to \( \text{triggers} \approx 828 \). Consequently, if one wanted a safe readout solution, one could

Figure 6.8: A block diagram illustrating the data rates in the Alpide Data module and in the the Zynq-7020. The UART connection has a data rate of 115 200 b/s. The AXI4 Interconnect an effective rate of 74 Mb/s. 

82
6.2. READOUT SYSTEM EVALUATION

Figure 6.9: Data points taken from radiation measurements. For each frame the number of bytes and number of hits were counted. The lines are linear regression fits to the data points.

enable clustering in the ALPIDE, but use $M_c$ as a FIFO size estimate. This would ensure that the FIFO would never overflow for the calculated number of triggers. It should also be emphasized that this estimation does not depend on which data port that is being used, the serial data port, or the parallel data port. The calculation principle above thus offers a general solution for FIFO size estimation for readout electronics towards the ALPIDE.

As a last note, the UART speed will set the maximum rate of trains. Here a memory of 260 kB will take theoretically 260 kB/115 200 b/s $\approx$ 18s to empty over the UART connection. Thus we would at least need to wait 18 s between each train.

Clustering compression

As seen from Figure 6.9, the data reduction as a result from clustering will depend on the number of hits/frame. That is, the regression lines in Figure 6.9 are almost equal when hits/frame $\to$ 0, but separates when hits/frame $\gg$ 0. To investigate what the average data compression would be under radiation, as a result of enabling clustering. The data compression for 100,000 triggers was calculated as:

$$\text{compression} = 100\% \times (1 - \frac{\text{Num bytes with clustering}}{\text{Num bytes without clustering}}).$$  (6.5)

The compression was plotted versus the average cluster size, seen in Figure 6.10. As seen in the figure the minimum compression happens when the cluster size is small. The compression then gradually increases as the cluster size gets larger. The maximum compression is about 57%. This would suggest that the FIFO size estimate $M_c$ is safer as the expected cluster size increase.
6.2. READOUT SYSTEM EVALUATION

![Data compression from enabling clustering](image)

Figure 6.10: Clustering compression with respect to mean cluster size.

### 6.2.2 Hardware discussion

Utilizing the hardware discussed in this thesis has no immediate discrepancies between the requirements discussed in Chapter 4 and the final results. The biggest drawback, however, is the limitation to the readout speed. Since the Zynq-7020 on the ZedBoard has no transceivers, the parallel data port was chosen for data offloading, maintaining a readout speed of 320 Mb/s. However, as previously mentioned, the serial data port would offer an effective readout speed of 960 Mb/s. Using this port would be optimal for the characterization of the ALPIDE, as this would allow testing the full range of ALPIDE's capacity. Nevertheless, the single chip readout system developed in this thesis sought out to create a fairly simple readout solution, and the parallel data port offered that without sacrificing significant readout speed. Moreover, highlighting how to interface towards the parallel data port was in itself more interesting, as this was not the intended data offloading interface. Hence, by using the parallel data interface, the intent was also to give this thesis more value.

**PCIe x8 to PMOD**

During the development of the PCIe x8 to PMOD board, an emphasis was put towards signal integrity. The board was constructed to be as short as possible, with the idea of delaying the signals to a minimum. However, this also meant that less space would be available. This resulted in no room for termination resistors for the parallel data lines, see schematics in Figure E.4. From oscilloscope measurements the parallel data lines were found to experience a slight ringing effect caused by improper impedance matching [95], see Figure C.2 in Appendix C. In retrospect, the solution of sacrificing termination resistors for PCB space was a poor decision, and termination resistors should be added in future designs. Nevertheless, this problem was not found to cause any functional errors for the readout system.
6.2. READOUT SYSTEM EVALUATION

ALPIDE Carrier

As a last limitation with respect to the hardware, it should be noted that using the ALPIDE Carrier card does not completely fulfill the requirement of being independent, as discussed in Chapter 4. However, as this was how the ALPIDE chip was available to the Electronics section at UiO [90] at the time of this thesis, the choice of developing towards this card meant that a complete readout system could be developed. Moreover, the hardware design requirements highlighted in this thesis were general, e.g. the voltage level shifting, power supply needs, etc. Thus the ALPIDE Carrier card does not represent any big drawbacks for future development using this thesis as reference but not the ALPIDE Carrier card.

6.2.3 Firmware discussion

The firmware developed in this thesis, the Alpide Controller and the Alpide Data module, together with the verification done on these modules were the most time-consuming tasks done in this thesis. In retrospect, an argument can be made that it would have been better to utilize already developed modules. For example, the firmware framework used by a team at the Department of Physics and Technology at the University of Bergen was made available during this master thesis [96] [97]. This firmware is a modified version of the firmware discussed in section 4.1.1, developed by CERN. Utilizing this firmware would probably have been less time-consuming. However, by developing a firmware framework in this thesis, proper understanding of ALPIDE’s functionality was needed. This helped when analyzing the required FIFO memory size and clustering compression. The development of the firmware in this thesis also helped to fulfill the requirement of being independent, as discussed in Chapter 4.

Alpide Controller

In the Alpide Controller module’s AXI4 Interface two signals were added, as discussed in section 5.3.1, controlling the power supply and the clock input. However, these two signals are board specific signals. That is, the signal controlling ALPIDE’s power supply is specific for the PCIe x8 to PMOD board, and the signal controlling the clock input is specific for the ALPIDE Carrier board. Therefore, the Alpide Controller module is less portable to new platforms since the two signals would be useless for future designs not using the ALPIDE Carrier and PCIe x8 to PMOD board. The module would still work, but the two signals should be ignored. Moreover, it also means that the Alpide Controller module is not strictly a module handling ALPIDE’s control interface, which might be confusing. A better solution would have been to create a dedicated System module. This module could have handled all the board specific tasks.

As a last remark about the Alpide Controller module, the module was developed specifically towards the ALPIDE Carrier card. That is, as can be seen from Figure 5.10, the module interfaces towards the ALPIDE Carrier’s ICs with the DCTRL_IN, DCTRL_EN, and DCTRL_OUT. But, in a readout solution were the ALPIDE Carrier card would not be used and for example, the ALPIDE I/Os would be connected directly to an FPGA’s I/Os, the Alpide Controller module as explained in this chapter would not work. Making it work in other configurations would involve
small modifications to the already developed module in this thesis. Thus, to make it easier, a module created for direct electrical connection with the ALPIDE I/Os was also developed during this thesis. This module was only verified with simulations, as explained in section 5.5.2, and was never physical tested on any development boards. The module can be found on this thesis GitHub page [59].

**Alpide Data**

The Alpide Data was developed to store data words coming from the ALPIDE chip into memory. This data could then later be read by the PS and analyzed on a computer. However, as seen in the firmware developed for the Readout Unit at CERN, see Figure 4.3. The module handling the data stream also contains an "Error/busy extraction" module. This module would immediately signal if a busy data word was received, and would therefore enable the off-detector electronics to immediately slow down the trigger speed, as discussed in section 3.4.2. This functionality is not present in the current version of the Alpide Data module and is an obvious flaw. The busy word would, in the readout system of this thesis, be detected when the busy word was popped from the FIFO. This can, however, happen significantly later than the busy word was transmitted from the ALPIDE chip, and in turn might cause unavoidable data loss. Adding a busy detection functionality to the Alpide Data module would therefore be highly recommended.

**6.2.4 Software discussion**

The standalone software application, seen in Figure 5.11, was developed to suffice with respect to the task of offloading raw data from the ALPIDE chip to the computer and initiate different tests. That is, the standalone application was not priorities due to time constraints and had many shortcomings. For example, when new tests were created, the whole system needed to be re-programmed. This meant that creating new tests or modifying tests were time-consuming and complicated. It would have been preferable to be able to adjust ALPIDE’s configurations during experiments more easily.

As mentioned, the UART connection becomes a bottleneck for the rate of trains which the readout system can handle. Utilizing the ZedBoard’s Ethernet port [62] would have minimizes this bottleneck. Nevertheless, for the tests done in this thesis, this was not a problem, as no measurements which required high frame rates were taken.

**6.2.5 Conclusion**

The readout system developed in this thesis was found to function and was successfully used in radiation measurements. There were however, two requirements in Table 4.1 which were not met or fully verified. One of these requirements, R.5, provided an interesting general perspective for readout electronics towards the ALPIDE chip. As the ALPIDE chip would be offloading data at a rate of 320 Mb/s, the read-out electronics needed to handle this data rate to prevent overflow. If this data rate could not be handled, a solution where trains of triggers were sent and then read out was proposed. If this solution were to be used, however, one needed a knowledge of the memory usage of triggers. This was estimated in Figure 6.9 and a
6.3 Summary

This chapter evaluated the readout system developed in this thesis. Radiation measurements were successfully done with the readout system showing that the system could be used for lab experiments. The system’s performance with respect to the requirements in Table 4.1 were then discussed. Two requirements were not reached. However, requirement R.5 offered a general aspect with regards to the handling of data for readout electronics. Thereby highlighting important considerations for future designs. After a discussion of the readout system was done, here proposed revisions were summarized in Table 6.2.

---

Table 6.2: Proposed revisions for the readout system. Arranged from high priority to low priority.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Part</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Alpide Data module</td>
<td>Add busy detection functionality.</td>
</tr>
<tr>
<td>#2</td>
<td>ZedBoard</td>
<td>Utilize the Ethernet port for data offloading.</td>
</tr>
<tr>
<td>#3</td>
<td>PCIe x8 to PMOD</td>
<td>Termination resistors for the parallel data lines.</td>
</tr>
<tr>
<td>#4</td>
<td>Standalone application</td>
<td>Better standalone application.</td>
</tr>
</tbody>
</table>

A general relation was proposed in equation 6.2 and equation 6.4, one when clustering was enabled and one when clustering was disabled, respectively. These equations would give an indication of the number of triggers a readout system could handle at maximum frame rate. A solution where equation 6.4 would be used for trigger and memory estimate, and the ALPIDE would be configured with clustering enabled was proposed, as this solution would ensure that a memory overflow could never occur.

Further, the average compression from the clustering functionality of the ALPIDE was tested. This functionality was found to give a compression from 36% to 57% under 5.5 MeV alpha particle radiation.

To summarize the discussion of the parts of the readout system, Table 6.2 shows the proposed revisions for the system. The last revision proposal: ”Better standalone application” is perhaps a bit general. The reason for this was that not much time was spent on the standalone application due to time constraints. In general, the standalone application needed more features. However, listing these features are beyond the scope of this thesis and thus the general proposal.
Chapter 7

Conclusion

This thesis main goal was to explore the ALPIDE by developing a readout system towards a single ALPIDE chip. This goal was achieved, and radiation measurements were successfully done using the developed system. Whilst different solutions were investigated in the development process of the readout system, providing perspectives on the ALPIDE’s readout functionalities, two VHDL modules developed in-house were the core of the system. These modules, the Alpide Controller and the Alpide Data, gave the developer a physical abstraction layer when interfacing with the ALPIDE. Moreover, the two modules were device independent, meaning that the modules could be easily integrated onto new FPGA systems, making them suitable for future readout systems.

Notably, the developed readout system had some shortcomings with respect to data handling. For example, the maximum frame rate for the ALPIDE could only be held for about 3229 triggers, given an expected 100 hits/frame, before the FIFO of the Alpide Data module was filled. This was calculated from the relation in equation 6.2. This relation proved, however, to be an interesting general concept for readout electronics developed for the ALPIDE. As readout electronics either needed to offload data at the same rate as the ALPIDE was offloading data, or the readout electronics could send trains of triggers, here the train size could be calculated from equation 6.2. Moreover, a safe readout solution was suggested, here the FIFO size would be calculated from when clustering was disabled, as in equation 6.4, but the ALPIDE would have clustering enabled. This would ensure that no FIFO overflow occurred. The thesis also investigated briefly the average clustering compression of the ALPIDE, here it was shown to give up to 57% data compression under 5.5 MeV alpha particle radiation. These findings, together with the proposed revisions in Table 6.2, fulfilled this thesis second-order objective: to highlight specification needs for readout electronics.

In conclusion, this thesis achieved its chief objective and second-order objective, providing a readout system that can be of much use to further research with the ALPIDE.
7.1 Future work

Suggestions for future works are addressed in the bullet points below.

- Implement the readout system on the PYNQ board [98]. The readout system developed in this thesis is complicated to use due to the lack of a good standalone application. And as noted, modifying or creating new tests are time-consuming. By implementing the Alpide Data and Alpide Controller module on the PYNQ development board, the need for a standalone application would be redundant, as the PYNQ can easily be programmed in Jupyter Notebook using Python [98]. This framework would make prototyping new tests with the ALPIDE significantly easier. Moreover, the PYNQ utilizes Ethernet, hence revision proposal #2 in Table 6.2 would also be solved.

- Investigate busy probabilities with the developed system. Simulations of busy probabilities have been done in [48], however, it would be interesting to see if the same results could be obtained with the readout system.

- Investigate why even cluster sizes are favored. In all cluster size distributions, even cluster sizes were observed more frequent. This thesis does not offer any explanation of this. Investigating why this occurs would be valuable for the understanding of the ALPIDE.

- Investigate the large number of single pixel hits during radiation. Under radiation of alpha particles, a large number of single pixel hits were observed. These hits were not due to faulty pixels, as these had been masked, and no explanation of the large occurrence was given in this thesis.
Appendix A

ALPIDE pixel indexing scheme

This appendix explains the indexing scheme used by ALPIDE when transmitting data packets. That is, how to convert the raw data transmitted over the parallel data port or the serial data port to geographical locations for pixel hits.

The ALPIDE chip divides its pixel matrix into 32 regions. Here each region has 16 priority encoders, and each encoder has 1024 pixels. This is illustrated in Figure A.1. As can be seen from Figure 3.11 the REGION HEADER is transmitted first, then DATA SHORT or DATA LONG words will follow.

REGION HEADER

This word contains the region of the pixel hits. Thus every subsequent DATA SHORT/DATA LONG word will be within that region. If a region has no pixel hits the region is skipped.

DATA SHORT

The DATA SHORT word contains the geographical location of a single hit. Here the encoder_id[3:0] represents which encoder and the addr[9:0] represents the pixel address. See Table 3.3 and Figure 3.11.

DATA LONG

The DATA LONG word can compress up to 8 pixel hits in a hit map. This word is transmitted if clustering is enabled in the configuration registers. When the data word is transmitted it groups together hits neighboring. The DATA LONG word first bytes is a replica of the DATA SHORT word, see Table 3.3. This is the first pixel hit in the cluster. The next section of the DATA LONG word then contains the hit map. An example of a clustered hit is shown in Figure A.2.
Figure A.1: The pixel indexing scheme used for the ALPIDE pixel matrix.

Figure A.2: Example of a DATA LONG word. Here the cluster of 4 pixel hits are shown. [43, Figure 3.11].
Appendix B

ALPIDE electrical characteristics & ALPIDE Carrier ICs

In this appendix the electrical requirements for the ALPIDE’s I/Os are addressed. Also the ALPIDE Carrier card’s solution to these requirements are explained.

ALPIDE electrical characteristics

In Table B.1 ALPIDE’s electrical characteristics of the ports relevant for this thesis are summarized. The CMOS I/Os uses the 1.8 V standard. Each CMOS I/O pads has either a pull-up or pull-down resistor. The MCLK and DCTRL differential lines are designed after the MLVDS standard [43], [99].

ALPIDE Carrier ICs

In Figure B.1 a cut out of ALPIDE Carrier’s schematic is shown. The figure shows the two ICs (SN65MLVD200AD) utilized by the ALPIDE Carrier card to convert the control line and the system clock to single ended CMOS 3.3 V lines. These ICs helps to reduce the complexity of the readout system as the ZedBoard PMOD I/Os has CMOS 3.3 V lines [62]. Moreover, the MCLK_EN_3V3 line in Figure B.1 enables control of the clock signal towards the ALPIDE chip. This signal is routed to the Alpide Controller AXI4 interface.

Table B.1: ALPIDE’s ports electrical characteristics. Table data taken from [43, Table 2.1].

<table>
<thead>
<tr>
<th>Physical Port</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCLK_P</td>
<td>Differential (MLVDS)</td>
</tr>
<tr>
<td>MCLK_N</td>
<td>Differential (MLVDS)</td>
</tr>
<tr>
<td>DCTRL_P</td>
<td>Differential (MLVDS)</td>
</tr>
<tr>
<td>DCTRL_N</td>
<td>Differential (MLVDS)</td>
</tr>
<tr>
<td>HSDATA_P</td>
<td>Differential (LVDS)</td>
</tr>
<tr>
<td>HSDATA_N</td>
<td>Differential (LVDS)</td>
</tr>
<tr>
<td>DATA[7:0]</td>
<td>CMOS, internal pull-up</td>
</tr>
<tr>
<td>CHIPID[6:0]</td>
<td>CMOS, internal pull-down</td>
</tr>
</tbody>
</table>
Figure B.1: Top: The differential clock (MCLK\_P and MCLK\_N) is converted to MCLK\_CMOS\_3V3, with a clock enable signal (MCLK\_EN\_3V3). Bottom: The bi-directional differential control interface (DCTRL\_P and DCTRL\_N), converted to three unidirectional lines, here DCTRL\_EN\_3V3, DCTRL\_IN\_3V3 and DCTRL\_OUT\_3V3. All the signals are converted to a CMOS 3.3V level, making them suitable for connection with the ZedBoard’s PMOD headers. [63]
Appendix C

Transmission integrity

To ensure that the signals from the ZedBoard to the ALPIDE, and vice versa, was transmitted correctly, some form of signal integrity check had to be done.

System clock and control line

As the system clock and the control line both has a line frequency of 40 MHz the lines were terminated with the same resistance. For the system clock a simple comparison with different termination resistors were done, here a 200 Ω resistor was found to give the best result. In Figure C.1 a oscilloscope measurement can be seen with no termination and a 200 Ω termination resistor. As can be seen from the figure, the line with no termination gives unwanted ringing and overshoots the maximum voltage level of 3.3 V. The line with termination has no ringing and has a wanted voltage level. The apparent non-squared waveform of the system clock is assumed to be mostly due to the oscilloscopes bandwidth of 200 MHz, and was not found to be a problem in the readout system.

The system clock line and the control line were also properly length matched. This was important as the control line’s protocol is synchronized to the system clock.

Parallel data port

The parallel data port had 8 lines which presented some length matching problems as the PCB design was relative small. In Figure C.2 the two parallel data lines which had the biggest length difference were measured. The difference in propagation delay is measured to be around 1.6 ns and represents about 6% of one data cycle. This is not perfect and could be a source of error for data transmission. A greater effort towards proper length matching of the parallel data lines should be done in future designs.

Moreover, the parallel data lines were not terminated due to shortage of space. This results in unwanted ringing, as seen in Figure C.2, and should also be addressed in future designs. Nevertheless, non of these issues were found to give any functional problems with the system developed in this thesis.
Figure C.1: Line 1 shows the clock line with no termination. Line 2 shows the clock line with a 200 Ω termination resistor. Measurements was done with a Tektronix MSO 2024B 200 MHz Oscilloscope.

Figure C.2: The difference between two falling edges between two parallel data lines. The two lines with the biggest length difference were measured. Measurements was done with a Tektronix MSO 2024B 200 MHz Oscilloscope.
Appendix D

Oscilloscope measurements

This appendix contains various oscilloscope measurements done throughout this thesis.
Figure D.1: The yellow plot (1) shows the control line being driven from the Zed-Board. The blue plot (2) shows the respons from the ALPIDE. As can be seen a transaction over the control line takes about 2.6\(\mu\)s, which is equivalent to 105 clock cycles at 40 MHz. The Measurements was done with a Tektronix MSO 2024B 200 MHz Oscilloscope

Figure D.2: Two parallel data lines probed during a test with the DAQ board. The measurements shows two bits being sent over the parallel data line, thus indicating that the parallel data lines are being utilized for data offloading. The Measurements was done with a Tektronix MSO 2024B 200 MHz Oscilloscope
Appendix E

PCIe x8 to PMOD Schematic and Layout
Figure E.1: Top schematic layout of the PCIe x8 to PMOD board.

100
Figure E.2: PCIe x8: Schematic of the PCIe x8 connector.

101
Figure E.3: POWER and Voltage Translation: Schematic of the power supply (top) and the level translator (bottom).
Figure E.4: PMOD: Schematic of the PMOD connector.
Figure E.5: PCB layout of the PCIe x8 to PMOD board.
Appendix F

Alpide Controller register map

This appendix explains the AXI4 interface register map for the Alpide Controller module. Only the address offsets are shown as the base address might change each implementation. Extra notes are added about some of the register fields.

0x0 - Chip id & Opcode

Table F.1: Offset address 0x0. Register for chip id and opcode. (Read/Write)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>OPCODE</td>
<td>Opcode to write to ALPIDE chip</td>
</tr>
<tr>
<td>14:8</td>
<td>CHIPID</td>
<td>Which ALPIDE chip to read/write to</td>
</tr>
<tr>
<td>31:15</td>
<td>Not used</td>
<td>-</td>
</tr>
</tbody>
</table>

- The format of the transaction, as discussed in section 3.4.1, is decided by the OPCODE and is handled by the Alpide Controller module.

0x4 - Address & Data to write

Table F.2: Offset address 0x4. Register for writing. (Read/Write)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>DATA[7:0]</td>
<td>The LSB to write to ALPIDE</td>
</tr>
<tr>
<td>15:8</td>
<td>DATA[15:8]</td>
<td>The MSB to write to ALPIDE</td>
</tr>
<tr>
<td>23:16</td>
<td>REG ADDR[7:0]</td>
<td>The LSB address to write to</td>
</tr>
<tr>
<td>31:24</td>
<td>REG ADDR[15:8]</td>
<td>The MSB address to write to</td>
</tr>
</tbody>
</table>
0x8 - Data read & busy

Table F.3: Offset address 0x8. Register for reading. (Read only)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>DATA[7:0]</td>
<td>The LSB read from ALPIDE</td>
</tr>
<tr>
<td>15:8</td>
<td>DATA[15:8]</td>
<td>The MSB read from ALPIDE</td>
</tr>
<tr>
<td>22:16</td>
<td>CHIPID[7:0]</td>
<td>The ALPIDE chip who responded</td>
</tr>
<tr>
<td>23</td>
<td>busy</td>
<td>High when Alpide Controller is busy</td>
</tr>
<tr>
<td>31:24</td>
<td>Not used</td>
<td>-</td>
</tr>
</tbody>
</table>

- When the busy field is high no registers shall be changed. This indicates that the Alpide Controller is in the middle of a transaction, and might be reading or writing to the AXI4 interface registers.

0xC - Control register

Table F.4: Offset address 0xC. Control register. (Read/Write)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>start</td>
<td>Write high to start transaction</td>
</tr>
<tr>
<td>1</td>
<td>reset</td>
<td>Reset the Alpide Controller module</td>
</tr>
<tr>
<td>2</td>
<td>ALPIDE enable</td>
<td>Turn the ALPIDE on</td>
</tr>
<tr>
<td>3</td>
<td>System clock enable</td>
<td>Turn the clock signal towards ALPIDE on</td>
</tr>
<tr>
<td>31:4</td>
<td>Not used</td>
<td>-</td>
</tr>
</tbody>
</table>

- The start field is automatically driven low after the Alpide Controller has initiated the given transaction. There is thus no need to write the start signal low again.

- The ALPIDE enable and System clock enable signal must of course be set high before any transactions can be initiated.

- The Alpide Controller is reset while the reset field is high. This field shall be high for at least two system clock cycles (50 ns) to ensure that the reset is registered by the Alpide Controller.
Appendix G

Alpide Data register map

This appendix explains the AXI4 interface register map for the Alpide Data module. Only the address offsets are shown as the base address might change each implementation. Extra notes are added about some of the register fields.

0x0 - FIFO data

Table G.1: Offset address 0x0. Data currently in the FIFO. (Read only)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>FIFO data</td>
<td>Data in FIFO</td>
</tr>
</tbody>
</table>

0x4 - FIFO status

Table G.2: Offset address 0x4. The status of the FIFO. (Read only)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>full</td>
<td>High if FIFO is full</td>
</tr>
<tr>
<td>1</td>
<td>empty</td>
<td>High if FIFO is empty</td>
</tr>
<tr>
<td>2</td>
<td>half full</td>
<td>High if FIFO is half full</td>
</tr>
<tr>
<td>31:3</td>
<td>Not used</td>
<td>-</td>
</tr>
</tbody>
</table>

- The full and half full field are mirrors of the interrupt signal going to the Processor System (PS).
0x8 - FIFO control

Table G.3: Offset address 0x8. FIFO controller. (Read/Write)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rd</td>
<td>Pop the current value of the FIFO</td>
</tr>
<tr>
<td>1</td>
<td>enable write</td>
<td>Enable writing to the FIFO from ALPIDE</td>
</tr>
<tr>
<td>2</td>
<td>enable read</td>
<td>Enable reading of the FIFO from the PS</td>
</tr>
<tr>
<td>3</td>
<td>reset</td>
<td>Reset the FIFO</td>
</tr>
<tr>
<td>31:4</td>
<td>Not used</td>
<td>-</td>
</tr>
</tbody>
</table>

- Setting the rd field high will pop one 32-bit data word from the FIFO. The rd field is automatically set back to zero.

- The Alpide Data is reset while the reset field is high. This field shall be high for at least two of the internal Alpide Data clock cycles (12.5 ns) to ensure that the reset is registered by the Alpide Data module.

0xC - FIFO pop status

Table G.4: Offset address 0xC. FIFO pop status. (Read only)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>popped</td>
<td>This field is inverted each time data is popped</td>
</tr>
<tr>
<td>1</td>
<td>busy</td>
<td>When busy is high, rd should not be written high</td>
</tr>
<tr>
<td>31:2</td>
<td>Not used</td>
<td>-</td>
</tr>
</tbody>
</table>

- The popped field can be used to check if the FIFO was actually popped. This will change even though the FIFO is empty.

- The busy field is used to indicate when the FIFO can be popped. That is when the rd field can be set high.
Appendix H

Hardware-specific driver: Alpide Controller

This Appendix contains the code for the Alpide Controller driver. The header code is listed first, then the source code.

H.1 alpide_controller.h

/*
 * alpide_controller.h
 *
 * Created on: 17. okt. 2018
 * Author: matiasgg
 *
 * Description: Contains low level Alpide functions, implemented by the alpide_controller module
 */

#ifndef SRC_UTILS_ALPIDE_CONTROLLER_H_
#define SRC_UTILS_ALPIDE_CONTROLLER_H_

#include "xil_types.h"

// ALPIDE Registers (Only a selected subset)
#define ALPIDE_REG_COMMAND 0x0000
#define ALPIDE_REG_MODE_CTRL 0x0001
#define ALPIDE_REG_FROMU_CFG1 0x0004
#define ALPIDE_REG_FROMU_CFG2 0x0005
#define ALPIDE_REG_FROMU_CFG3 0x0006
#define ALPIDE_REG_CMUDMU_CFG 0x0010
#define ALPIDE_REG_DMU_FIFO_LO 0x0012
#define ALPIDE_REG_DMU_FIFO_HI 0x0013
#define ALPIDE_REG_DAC_VRESETD 0x0602
#define ALPIDE_REG_DAC_VCASN 0x0604
#define ALPIDE_REG_DAC_VPULSEH 0x0605
#define ALPIDE_REG_DAC_VPULSEL 0x0606
#define ALPIDE_REG_DAC_VCASN2 0x0607
#define ALPIDE_REG_DACITHR 0x060E
#define ALPIDE_REG_TEST_CTRL 0x0701

// ALPIDE Command codes

109
# define ALPIDE_CMD_TRIGGER 0x0055
# define ALPIDE_CMD_GRST 0x00D2
# define ALPIDE_CMD_PRST 0x00E4
# define ALPIDE_CMD_PULSE 0x0078
# define ALPIDE_CMD_BCRST 0x0036
# define ALPIDE_CMD_RORST 0x0063
# define ALPIDE_CMD_DEBUG 0x00AA
# define ALPIDE_CMD_WROP 0x009C
# define ALPIDE_CMD_RDOP 0x004E
# define ALPIDE_CMD_CMU_CLEAR_ERR 0xFF00
# define ALPIDE_CMD_FIFOTEST 0xFF01
# define ALPIDE_CMD_LOADOBDEFCFG 0xFF02
# define ALPIDE_CMD_XOFF 0xFF10
# define ALPIDE_CMD_XON 0xFF11
# define ALPIDE_MCD_ADCMEASURE 0xFF20

/* Writes data to regAddress over the control port. ChipID is the ALPIDE chip ID. */
void writeALPIDE(u32 chipID, u32 regAddress, u32 data);

/* Reads the data stored at register address regAddress from the ALPIDE chip with chipID. */
void readALPIDE(u32 chipID, u32 regAddress);

/* Sends an opcode trigger command to ALPIDE */
void triggerALPIDE(u32 chipID);

/* Sends the opcode to ALPIDE */
void opcodeALPIDE(u32 chipID, u8 opcode);

/* Gives power to the ALPIDE chip */
void turnALPIDE_on(void);

/* Switches the power for the ALPIDE chip off */
void turnALPIDE_off(void);

/* Turns the clock on towards the ALPIDE chip */
void turnALPIDE_CLK_on(void);

/* Turns the clock off for the ALPIDE chip */
void turnALPIDE_CLK_off(void);

/* Initialize the controller registers to 0, and then sets chipID */
void setup_alpide_controller(u32 chipID);

/* Resets all the state inside the Alpide Controller module */
void reset_alpide_controller(void);

#endif /* SRC_UTILS_ALPIDE_CONTROLLER_H */
H.2 alpide_controller.c

/*
 * alpide_controller.c
 * Created on: 17. okt. 2018
 * Author: matiasgg
 */

#include "AlpideController.h"
#include "alpide_controller.h"
#include "xparameters.h"
#include "xil_io.h"

void writeALPIDE(u32 chipID, u32 regAddress, u32 data) {
    // Checking if alpide is busy
    u32 busy = ALPIDECONTROLLER::mReadReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, ALPIDECONTROLLER_S00_AXI_SLV_REG2_OFFSET);
    busy = (busy >> 23) & 0x1; // Only read busy
    while (busy) {
        // Only change registers when Alpide is not busy
        busy = ALPIDECONTROLLER::mReadReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, ALPIDECONTROLLER_S00_AXI_SLV_REG2_OFFSET);
        busy = (busy >> 23) & 0x1;
    }

    // Setting registers
    ALPIDECONTROLLER::mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, ALPIDECONTROLLER_S00_AXI_SLV_REG0_OFFSET, (chipID << 8) | ALPIDE_CMD_WRP); // 0x9C for write and setting chip id
    ALPIDECONTROLLER::mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, ALPIDECONTROLLER_S00_AXI_SLV_REG1_OFFSET, (regAddress << 16) | data); // Setting register address and data

    // Starting a write
    u32 reg3 = ALPIDECONTROLLER::mReadReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET);
    ALPIDECONTROLLER::mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET, reg3 | 1); // Starting a write
}

u32 readALPIDE(u32 chipID, u32 regAddress) {
    // Checking if alpide is busy
    u32 busy = ALPIDECONTROLLER::mReadReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, ALPIDECONTROLLER_S00_AXI_SLV_REG2_OFFSET);
    busy = (busy >> 23);
    while (busy) {
        busy = ALPIDECONTROLLER::mReadReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, ALPIDECONTROLLER_S00_AXI_SLV_REG2_OFFSET);
    }
}
busy = (busy >> 23) & 0x1;
}

// Setting registers
ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
ALPIDECONTROLLER_S00_AXI_SLV_REG0_OFFSET, (chipID << 8) | ALPIDE_CMD_RDOP); // 0x4E for read and setting chip id
ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
ALPIDECONTROLLER_S00_AXI_SLV_REG1_OFFSET, regAddress << 16); // Setting register address and data

// Starting a read
u32 reg3 = ALPIDECONTROLLER_mReadReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET);
ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET, reg3 | 1); // Starting a read
reg3 = ALPIDECONTROLLER_mReadReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET);
while(reg3 & 1){ // Waits until start goes low, that is when start is registered
reg3 = ALPIDECONTROLLER_mReadReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET);
}
// Wait while busy
busy = ALPIDECONTROLLER_mReadReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
ALPIDECONTROLLER_S00_AXI_SLV_REG2_OFFSET);
busy = (busy >> 23);
while(basic){
busy = ALPIDECONTROLLER_mReadReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
ALPIDECONTROLLER_S00_AXI_SLV_REG2_OFFSET);
busy = (busy >> 23) & 0x1;
}
// Return read value
return (0xFFFF & ALPIDECONTROLLER_mReadReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
ALPIDECONTROLLER_S00_AXI_SLV_REG2_OFFSET)); // Returning data high and low

void triggerALPIDE(u32 chipID){

// Checking if alpide is busy
u32 busy = ALPIDECONTROLLER_mReadReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
ALPIDECONTROLLER_S00_AXI_SLV_REG2_OFFSET);
busy = (busy >> 23) & 0x1; // Only read busy
while(basic){ // Only change registers when Alpide is not busy
busy = ALPIDECONTROLLER_mReadReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,

ALPIDECONTROLLER_S00_AXI_SLV_REG2_OFFSET);}  
bus y = (bus y >> 23) & 0x1; 
}  
// Setting opcode registers 
ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, 
ALPIDECONTROLLER_S00_AXI_SLV_REG0_OFFSET, (chipID << 8) | 
ALPIDE_CMD_TRIGGER); // 0x9C for write and setting chip id 
// Starting a write 
u32 reg3 = ALPIDECONTROLLER_mReadReg( 
 XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, 
ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET); 
ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, 
ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET, reg3 | 1); // Starting an write 
} 
void opcodeALPIDE(u32 chipID, u8 opcode){ 
// Checking if alpide is busy 
u32 busy = ALPIDECONTROLLER_mReadReg( XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, 
ALPIDECONTROLLER_S00_AXI_SLV_REG2_OFFSET); 
busy = (bus y >> 23) & 0x1; // Only read busy 
while(bus y){ // Only change registers when Alpide is not busy 
busy = ALPIDECONTROLLER_mReadReg( XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, 
ALPIDECONTROLLER_S00_AXI_SLV_REG2_OFFSET); 
busy = (bus y >> 23) & 0x1; 
}  
// Setting opcode registers 
ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, 
ALPIDECONTROLLER_S00_AXI_SLV_REG0_OFFSET, (chipID << 8) | opcode ); 
// Starting a write 
u32 reg3 = ALPIDECONTROLLER_mReadReg( XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, 
ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET); 
ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, 
ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET, reg3 | 1); // Starting an write 
}  
void turnALPIDE_on(void){  
u32 reg3 = ALPIDECONTROLLER_mReadReg( XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, 
ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET); 
ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR, 
ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET, reg3 | 0x4); // Setting ALPIDE_EN high 
sleep(1); // Make sure Alpide has time to boot, this can properly be a lot shorter.
```c
void turnALPIDE_off(void)
{
    u32 reg3 = ALPIDECONTROLLER_mReadReg(
        XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
        ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET);
    ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
        ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET, reg3 & (~0x4)); // Setting ALPIDE_EN low
}

void turnALPIDE_CLK_on(void)
{
    u32 reg3 = ALPIDECONTROLLER_mReadReg(
        XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
        ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET);
    ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
        ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET, reg3 | 0x8); // Setting MCLK_EN high
    sleep(1); // Make sure clock propagate
}

void turnALPIDE_CLK_off(void)
{
    u32 reg3 = ALPIDECONTROLLER_mReadReg(
        XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
        ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET);
    ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
        ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET, reg3 & (~0x8)); // Setting MCLK_EN low
}

void setup_alpide_controller(u32 chipID)
{
    ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
        ALPIDECONTROLLER_S00_AXI_SLV_REG0_OFFSET, (chipID << 8)&0x7F00); // Setting chip ID
    ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
        ALPIDECONTROLLER_S00_AXI_SLV_REG1_OFFSET, 0x0); // Set to Zero
    ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
        ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET, 0x0); // Set to ZERO
}

void reset_alpide_controller(void)
{
    u32 reg3 = ALPIDECONTROLLER_mReadReg(
        XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
        ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET);
    ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
        ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET, reg3 | 0x2); // Resetting Alpide Controller
    usleep(1);
    ALPIDECONTROLLER_mWriteReg(XPAR_ALPIDECONTROLLER_0_S00_AXI_BASEADDR,
        ALPIDECONTROLLER_S00_AXI_SLV_REG3_OFFSET, reg3 & (~0x2)); // Enabling Alpide Controller
    usleep(1); // Allow alpide controller to reset.
}
```
Appendix I

Hardware-specific driver: Alpide Data

This Appendix contains the code for the Alpide Data driver. The header code is listed first, then the source code.

I.1 alpide_data.h

```c
/*
 * alpide_data.h
 *
 * Created on: 10. jan. 2019
 * Author: matiasgg
 */

#ifndef SRC_UTILS_ALPIDE_DATA_H_
#define SRC_UTILS_ALPIDE_DATA_H_

/* Reads the FIFO value and returns it, then pops the fifo */
u32 popFIFO();

/* Checks if the FIFO is full */
u32 isFIFO_Full();

/* Checks if the fifo is empty */
u32 isFIFO_Empty();

/* Reset the alpide data module, should be done when starting the system */
void reset_alpide_data();

/* Setup the alpide data module*/
void setup_alpide_data();
#endif /* SRC_UTILS_ALPIDE_DATA_H_ */
```
```c
#include "AlpideData.h"
#include "alpide_data.h"
#include "xparameters.h"
#include "xil_io.h"

u32 popFIFO() { // Fetching the data then popping
    u32 busy = (0x2 & ALPIDEDATA_mReadReg(XPAR_ALPIDEDATA_1_S00_AXI_BASEADDR,
    ALPIDEDATA_S00_AXI_SLV_REG3_OFFSET)); // Waiting until popped value changes
    while (busy == 0x2) { // Don’t pop while busy == 1
        busy = (0x2 & ALPIDEDATA_mReadReg(XPAR_ALPIDEDATA_1_S00_AXI_BASEADDR,
        ALPIDEDATA_S00_AXI_SLV_REG3_OFFSET)); // Waiting until popped value changes
    }
    u32 fifo_data = ALPIDEDATA_mReadReg(XPAR_ALPIDEDATA_1_S00_AXI_BASEADDR,
    ALPIDEDATA_S00_AXI_SLV_REG0_OFFSET); // Fetching data
    ALPIDEDATA_mWriteReg(XPAR_ALPIDEDATA_1_S00_AXI_BASEADDR,
    ALPIDEDATA_S00_AXI_SLV_REG2_OFFSET, 0x7); // Setting rd = 1, thereby popping the fifo
    return fifo_data;
}

u32 isFIFO_Full() {
    u32 reg1 = ALPIDEDATA_mReadReg(XPAR_ALPIDEDATA_1_S00_AXI_BASEADDR,
    ALPIDEDATA_S00_AXI_SLV_REG1_OFFSET);
    if((reg1 & 0x1) == 0x1) {
        return 0x1; // Fifo is full
    }
    return 0;
}

u32 isFIFO_Empty() {
    u32 reg1 = ALPIDEDATA_mReadReg(XPAR_ALPIDEDATA_1_S00_AXI_BASEADDR,
    ALPIDEDATA_S00_AXI_SLV_REG1_OFFSET);
    if((reg1 & 0x2) == 0x2) {
        return 0x1; // Fifo is empty
    }
    return 0;
}

void reset_alpide_data() {
    u32 reg2 = ALPIDEDATA_mReadReg(XPAR_ALPIDEDATA_1_S00_AXI_BASEADDR,
    ALPIDEDATA_S00_AXI_SLV_REG2_OFFSET);
    ALPIDEDATA_mWriteReg(XPAR_ALPIDEDATA_1_S00_AXI_BASEADDR,
    ALPIDEDATA_S00_AXI_SLV_REG0_OFFSET, 0x7); // Setting rd = 1, thereby resetting the fifo
}
```

void setup_alpide_data()
{
    XPAR_ALPIDEDATA_S00_AXI_BASEADDR,
    XPAR_ALPIDEDATA_S00_AXI_SLV_REG2_OFFSET, 0x8); // Setting reset signal to one
    usleep(100); // Makes sure the reset is registered
    XPAR_ALPIDEDATA_mWriteReg(XPAR_ALPIDEDATA_S00_AXI_BASEADDR,
    XPAR_ALPIDEDATA_S00_AXI_SLV_REG2_OFFSET, reg2 & 0x7); // Setting reset signal to zero
    usleep(100); // Make sure module is back in reset
}
Appendix J

Alpide Controller Module VHDL

This appendix contains the VHDL code for the Alpide Controller module. The AXI4 interface has been left out, but can be found on this thesis GitHub page [59].

J.1 Controller

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

entity alpide_dctrl_controller is

  port(
    clk : in std_logic;  -- System clock
    reset_n : in std_logic;  -- Reset System, active high
chipid_w : in std_logic_vector(6 downto 0);  -- Chipid
to be written
chipid_r : out std_logic_vector(6 downto 0);  -- Chipid
to be read
opcode : in std_logic_vector(7 downto 0);  -- Opcode
to be written
regaddr_h : in std_logic_vector(7 downto 0);  -- Reg
to be written
address to be written REGADDR[15−8]
regaddr_l : in std_logic_vector(7 downto 0);  -- Reg
address to be written REGADDR[7−0]
data_h_w : in std_logic_vector(7 downto 0);  -- Data
to be written DATA[15−8]

  );

end alpide_dctrl_controller;
```

Copyright 2018 Matias Gjestvang Greaker.
Created : 05-04-2018
Author : Matias G. Greaker <matiasgg@fys.uio.no>
File : alpide_dctrl_controller_2002_carrier.vhd
Standard : VHDL’2002
Purpose : Module to control alpide_dctrl_master for different OPCODES.
```vhdl
architecture controller_arch of alpide_dctrl_controller is

-- Signals for state

type state_type is (IDLE, WAIT_WRITE, HOLD_GAP, MASTER_IDLE, WAIT_READ, READ_CHAR); -- State signals
signal state_reg : state_type := IDLE;
signal busy_i : std_logic; -- Make shure we only load registers when state in IDLE;
signal gap_cycles : integer := 0; -- Controls how long the idle gap between chars should be

-- Signals for/from alpide_dctrl_master

signal master_valid : std_logic;
signal master_busy : std_logic;
signal master_rnw : std_logic;
signal master_reset_n : std_logic;
signal master_char_w : std_logic_vector(7 downto 0);
signal master_char_r : std_logic_vector(7 downto 0);

-- alpide_dctrl_master instantiation

component alpide_dctrl_master is
  port(
    clk : in std_logic; -- ALPIDE clock
    reset_n : in std_logic; -- Synchronous active high
    valid : in std_logic; -- Module enabl, this signal starts a writing or read procedure
    char_w : in std_logic_vector(7 downto 0); -- Char to be written to ALPIDE chip
  end port;
end component;
```

```
data_l_w : in std_logic_vector(7 downto 0); -- Data to be written DATA[7-0]
data_h_r : out std_logic_vector(7 downto 0); -- Data
start : in std_logic; -- Used to start transaction
busy : out std_logic; -- Indicate when state is busy
dctrl_in : out std_logic; -- Signal being sent into ALPIDE chip
dctrl_out : in std_logic; -- Signal being sent from ALPIDE chip
dctrl_en : out std_logic -- Enable signal to switch between dctrl_out and dctrl_in, high means we are writing to dctrl_in, low means we are reading from dctrl_out
);
end entity;
```
J.1. CONTROLLER

```vhdl
    char_r : out std_logic_vector(7 downto 0); -- Char to be read from ALPIDE chip
    rw      : in std_logic;  -- Read or write r/w = 0/1
    busy    : out std_logic; -- Busy signal, says if the module is in the middle of an transaction, therby indicating when we can load in a new char
    dctrl_in : out std_logic; -- Signal being sent into ALPIDE chip
    dctrl_out : in std_logic; -- Signal being sent from ALPIDE chip
    dctrl_en : out std_logic; -- Enable signal to switch between dctrl_out and dctrl_in, high means we are writing to dctrl_in, low means we are reading from dctrl_out;

end component alpide_dctrl_master;

-- Type declaration

type ARRAY_OF_CHAR is array (1 to 5) of std_logic_vector(7 downto 0); -- Using this array type to iterate through the chars written to ALPIDE chip

type ARRAY_OF_CHAR2 is array (1 to 3) of std_logic_vector(7 downto 0);

-- Signal used to write from p_state_controller to p_registers
signal read_reg : ARRAY_OF_CHAR2 := (others => "00000000"); -- Initialize to 0

-- Type declaration

attribute DONT_TOUCH : string;
attribute DONT_TOUCH of dctrl_master_instance : label is "TRUE";

begin

-- Writing read results out entity

    chipid_r <= read_reg(3)(6 downto 0);
    data_l_r <= read_reg(2);
    data_h_r <= read_reg(1);

-- Busy handling

    with state_reg select
      busy <= '0' when IDLE,
             '1' when others;

-- alpide_dctrl_master instance

    dctrl_master_instance : alpide_dctrl_master
      port map(clk => clk,
```

121
```vhdl
reset_n => master_reset_n,
valid  => master_valid,
char_w => master_char_w,
char_r => master_char_r,
rnw    => master_rnw,
busty  => master_busty,
dctrl_in => dctrl_in,
dctrl_out => dctrl_out,
dctrl_en => dctrl_en
);

-- State machine, This state machine runs on rising edge, this is so
alpide_dctrl_master has time to respond and set signal on falling
edge.

p_state_controller : process(clk)
variable w_cnt   : integer; -- Keeping track of how many
char left to write
variable r_cnt   : integer; -- Keeping track of how many
car left to read
variable write_reg : ARRAY_OF_CHAR;
variable gap_cycles_i : integer; -- Having a internal hold
gap
variable hold_idle : integer; -- Variable used to control
how long master should hold the line high (in IDLE), before giving
it to the ALPIDE chip in a read command
begin
if rising_edge(clk) then
    if reset_n = '1' then
        master_reset_n <= '1'; -- Resetting alpide_dctrl_master
        state_reg <= IDLE; -- Resetting alpide_dctrl_controller
    else
        master_reset_n <= '0';
    end if;
    case state_reg is
        when IDLE =>
            master_valid <= '0'; -- Setting default
            master_rnw <= '1'; -- Default is to write
            hold_idle := 0; -- Resetting value
            gap_cycles_i := gap_cycles;
            if start = '1' then
                if (opcode = "10011100") then -- Checking if we
                    have UNICAST or MULTICAST Write
                        w_cnt := 5; -- If we have write command we
                    will write 6 chars
                        r_cnt := 0; -- And read zero
                        write_reg(6) := opcode; -- Setting up
                        write_reg for iteration, this will make it easy to iterate though
                        and write to ALPIDE chip
                    end if;
                end if;
end case;
end if;
```

write_reg(5) := '0' & chipid_w;  -- Chipid is only 7 bit long, so add a 0 bit to MSB.
write_reg(4) := regadr_l;  -- Filling in write_reg in correct order so we can iterate through
write_reg(3) := regadr_h;  --//--
write_reg(2) := data_l_w;  --//--
write_reg(1) := data_h_w;  --//--
-- Write_reg(0) will never be written to chip, I have implemented it this way so it will be easier to understand.

if (opcode = "01001110") then  -- Checking if we have UNICAST read
w_cnt := 3;  -- We will then write 4 chars
r_cnt := 3;  -- And read 3 chars
-- write_reg(4) := opcode;  -- Chipid is only 7 bit long, so add a 0 bit to MSB.
write_reg(2) := regadr_l;
write_reg(1) := regadr_h;
else  -- All other commands only depends on the opcode eg. TRIGGER, BCRST, PULSE, etc., therefore only writing one char.
w_cnt := 1;  -- Writing one char.
r_cnt := 0;  -- And read zero chars
write_reg(1) := opcode;  -- Setting up write_reg for iteration, not so important here since we only write one char
end if;
master_char_w <= opcode;  -- Load inn first char to be written (here the_OPCODE)
master_valid <= '1';  -- Starting write
if master_busy = '1' then  -- Checking if alpide_dctrl_master has start to write
state_reg <= WAIT_WRITE;
master_valid <= '0';  -- Writing has started therefore writing master_valid to 0
-- w_cnt := w_cnt - 1;  -- Writing first char has started therefore w_cnt --
end if;
when WAIT_WRITE =>
master_valid <= '0';  -- Writing control signal valid to 0
-- start <= '0';  -- Writing start back to 0, so we don’t start and infinite loop -- Could not do it here, since driven from multiple instances, this is done in AXI wrapper
if master_busy = '0' then  -- Checking if we have stopped writing, thereby ready for new command
if w_cnt = 0 then  -- Checking if we have written everything
if r_cnt = 0 then  -- Checking if we have something to read
state_reg <= IDLE;  -- Finished writing
else
state_reg <= MASTER_IDLE;  -- We have
something to read going back to idle
else

J.1. CONTROLLER

```plaintext
end if;

else
  master_char_w <= write_reg(w_cnt); -- Setting up next char to be written
  if gap_cycles_i = 0 then -- Check if we should quit gap and write new char
    master_valid <= '1'; -- Indicate start of new write
  end if;
  state_reg <= HOLD_GAP; -- Since master_valid is written high before entering HOLD_GAP we will immediately exit HOLD_GAP after entering
  w_cnt := w_cnt - 1; -- Getting ready for next char

  when HOLD_GAP =>
    if master_busy = '1' then -- Writing of new char has started
      master_valid <= '0';
      state_reg <= WAIT_WRITE; -- Go to wait_write while new char is being written
      gap_cycles_i := gap_cycles; -- Setting value back to what it was
    else
      if gap_cycles_i = 0 then
        master_valid <= '1'; -- Setting valid high, this will result in busy going high and so next state will be WAIT_WRITE
      else
        gap_cycles_i := gap_cycles_i - 1; -- One less cycle to wait
      end if;
    end if;

  when MASTER_IDLE =>
    if hold_idle = 3 then -- Making it hold for 5 clk cycles, counting with 2 clk cycles going in and out of states
      master_valid <= '1'; -- Setting valid high, thereby starting bus turn over
      if master_busy = '1' then -- bus turn around has started
        state_reg <= WAIT_READ;
      end if;
    else
      hold_idle := hold_idle + 1;
    end if;
```

124
when WAIT_READ =>
    master_valid <= '0'; -- Pulling it down to prevent
                        further read/write
    if master_busy = '0' then -- Char is ready to be
        if r_cnt = 0 then -- We are finished with
            reading
            state_reg <= IDLE;
        else
            state_reg <= READ_CHAR;
        end if;
    end if;

when READ_CHAR =>
    read_reg(r_cnt) <= master_char_r; -- This will put:
    read_reg(3) = CHIPID, read_reg(2) = DATA[7:0], read_reg(1) = DATA [15:8]
    if master_busy = '1' then
        r_cnt := r_cnt - 1;
        state_reg <= WAIT_READ;
    end if;

when others => state_reg <= IDLE;
end case;
end if;
end process p_state_controller;
end architecture;
J.2 Master

entity alpide_dctrl_master is

    port(
        clk : in std_logic; -- ALPIDE clock
        reset_n : in std_logic; -- Synchronous active high
        valid : in std_logic; -- Module enabl, this signal starts a writing or read procedure
        char_w : in std_logic_vector(7 downto 0); -- Char to be written to ALPIDE chip
        char_r : out std_logic_vector(7 downto 0); -- Char to be read from ALPIDE chip
        rnw : in std_logic; -- Read or write r/w = 0/1
        busy : out std_logic; -- Busy signal, says if the module is in the middle of an transaction, therby indicating when we can load in a new char
        dctrl_in : out std_logic; -- Signal being sent into ALPIDE chip
        dctrl_out : in std_logic; -- Signal being sent from ALPIDE chip
        dctrl_en : out std_logic -- Enable signal to switch between dctrl_out and dctrl_in, high means we are writing to dctrl_in, low means we are reading from dctrl_out
    );

end entity;

architecture dctrl_arch of alpide_dctrl_master is

    -- Signals for state
type state_type is (sIDLE, sSLAVE_IDLE, sWRITE_START, sWRITE_CHAR0,
sWRITE_CHAR1, sWRITE_CHAR2, sWRITE_CHAR3, sWRITE_CHAR4,
sWRITE_CHAR5, sWRITE_CHAR6, sWRITE_CHAR7, sWRITE_STOP, sREAD_CHAR0,
sREAD_CHAR1, sREAD_CHAR2, sREAD_CHAR3, sREAD_CHAR4, sREAD_CHAR5,
sREAD_CHAR6, sREAD_CHAR7, sREAD_STOP, sTURN_AROUND); -- State signals

signal state_reg : state_type;
signal state_next : state_type;

-- Signals controlling CHANGE

signal bmd : std_logic; -- Bus master drive signal, controlling if the bus is driven by chip = '0' or dctrl_master = '1'
signal bmd_cnt : integer range 0 to 55; -- Signal used to count how long the master shall release the bus

-- DCTRL

signal dctrl_i : std_logic; -- Dctrl internal signal

-- Write char internal

signal char_w_i : std_logic_vector(7 downto 0);

-- Internal control signal

signal go_to_read : std_logic; -- Signal used to indicate that ALPIDE has pulled the DCTRL line low, and we can now start reading from the line

begin

-- Busy handling

with state_reg select
  busy <= '0' when sIDLE | sREAD_STOP; -- Busy signal is high while reading/writing or in bus turning phase '1' when others;

-- Bus master drive handling and dctrl handling

dctrl_in <= dctrl_i;
dctrl_en <= bmd;

  -- bmd = bus master driver, controls if alpide_dctrl_master should
have control over the bus or ALPIDE chip.

```vhdl
with state_reg select
  bmd <= '0' when sTURN_AROUND|sSLAVE_IDLE|sREAD_CHAR0|sREAD_CHAR1|sREAD_CHAR2|sREAD_CHAR3|sREAD_CHAR4|sREAD_CHAR5|sREAD_CHAR6|sREAD_CHAR7|sREAD_STOP,
  '1' when others;
```

--- Bmd Count is used to know when to take back the bus. In read mode ALPIDE will use 50 cycles on the line.

```vhdl
p_bmd_count : process(clk)
begin
  if falling_edge(clk) then
    case state_reg is
      when sTURN_AROUND|sSLAVE_IDLE|sREAD_CHAR0|sREAD_CHAR1|sREAD_CHAR2|sREAD_CHAR3|sREAD_CHAR4|sREAD_CHAR5|sREAD_CHAR6|sREAD_CHAR7|sREAD_STOP =>
        bmd_cnt <= bmd_cnt + 1; -- This is used to control when the master can safely read from the dctrl line
      when others =>
        bmd_cnt <= 0;
    end case;
  end if;
end process;
```

--- State register, changing with falling edge, so the line is stable for reading by ALPIDE chip on rising edge

```vhdl
p_state_reg : process(clk)
begin
  if falling_edge(clk) then
    if (reset_n = '1') then
      state_reg <= sIDLE; -- Should ALPIDE chip also be reset from here?
    else
      state_reg <= state_next;
    end if;
  end if;
end process;
```

--- Output registers, used to sample the DCTRL line into register

```vhdl
p_out_reg : process(clk)
begin
  if rising_edge(clk) then
    case state_reg is
      when sSLAVE_IDLE =>
        if dctrl_out = '0' and bmd_cnt < 50 then
          go_to_read <= '1';
        else
          go_to_read <= '0';
        end if;
    end case;
  end if;
end process;
```

when sREAD_CHAR0 =>
    char_r(0) <= dctrl_out;  -- Sample signal, signal
when sREAD_CHAR1 =>
    char_r(1) <= dctrl_out;  -- Sample signal, signal
when sREAD_CHAR2 =>
    char_r(2) <= dctrl_out;  -- Sample signal, signal
when sREAD_CHAR3 =>
    char_r(3) <= dctrl_out;  -- Sample signal, signal
when sREAD_CHAR4 =>
    char_r(4) <= dctrl_out;  -- Sample signal, signal
when sREAD_CHAR5 =>
    char_r(5) <= dctrl_out;  -- Sample signal, signal
when sREAD_CHAR6 =>
    char_r(6) <= dctrl_out;  -- Sample signal, signal
when sREAD_CHAR7 =>
    char_r(7) <= dctrl_out;  -- Sample signal, signal
when others =>
    go_to_read <= '0';
end case;
end if;
end process;

-- Next-state logic

p_state : process(go_to_read, state_reg, char_w, valid, rnw, bmd_cnt, clk, dctrl_out, dctrl_i)
begin
    case state_reg is
        when sIDLE =>
            dctrl_i <= '1';  -- Driving line high in IDLE
            if valid = '1' then
                if rnw = '1' then  -- If we have write statement
                    state_next <= sWRITE_START;
                elsif rnw = '0' then  -- IF we have read statement
                    state_next <= sTURN_AROUND;  -- Starting bus
                end if;
            end if;
        else
            state_next <= sIDLE;
        end if;
        when sSLAVE_IDLE =>  -- Waiting for start signal
            dctrl_i <= '-';  -- To avoid latch, is don't care
            -- Uncommented as not needed
            -- if bmd_cnt < 50 then  -- Only read from dctrl while
            -- ALPIDE is driving, unless we can be reading Z values
            -- if rising_edge(clk) then
            --    if dctrl = '0' then
            --        state_next <= sREAD_CHAR0;  -- Jumps
            end if;
        end when;
    end case;
end process;
J.2. MASTER

straight to read char0 since start bit happens in sSLAVE_IDLE state
---
---
---
---
---
---
---
---

if bmd_cnt = 53 then — Take back bus when we are sure
dctrl is Z
    state_next <= sIDLE;
    elsif go_to_read = '1' then — This signal is being set
    by p_out_reg process
    state_next <= sREAD_CHAR0;
    else
    state_next <= sSLAVE_IDLE;
end if;

when sWRITE_START =>
dctrl_i <= '0'; — Drive line low, indicate start
state_next <= sWRITE_CHAR0;

when sWRITE_CHAR0 =>
dctrl_i <= char_w(0);
state_next <= sWRITE_CHAR1;
when sWRITE_CHAR1 =>
dctrl_i <= char_w(1);
state_next <= sWRITE_CHAR2;
when sWRITE_CHAR2 =>
dctrl_i <= char_w(2);
state_next <= sWRITE_CHAR3;
when sWRITE_CHAR3 =>
dctrl_i <= char_w(3);
state_next <= sWRITE_CHAR4;
when sWRITE_CHAR4 =>
dctrl_i <= char_w(4);
state_next <= sWRITE_CHAR5;
when sWRITE_CHAR5 =>
dctrl_i <= char_w(5);
state_next <= sWRITE_CHAR6;
when sWRITE_CHAR6 =>
dctrl_i <= char_w(6);
state_next <= sWRITE_CHAR7;
when sWRITE_CHAR7 =>
dctrl_i <= char_w(7);
state_next <= sWRITE_STOP;
when sWRITE_STOP =>
dctrl_i <= '1';
state_next <= sIDLE;
when sREAD_CHAR0 =>

```vhdl
when sREAD_CHAR1 =>
  dctrl_i <= '-'; -- To avoid latch, is don't care
  state_next <= sREAD_CHAR2;

when sREAD_CHAR2 =>
  dctrl_i <= '-'; -- To avoid latch, is don't care
  state_next <= sREAD_CHAR3;

when sREAD_CHAR3 =>
  dctrl_i <= '-'; -- To avoid latch, is don't care
  state_next <= sREAD_CHAR4;

when sREAD_CHAR4 =>
  dctrl_i <= '-'; -- To avoid latch, is don't care
  state_next <= sREAD_CHAR5;

when sREAD_CHAR5 =>
  dctrl_i <= '-'; -- To avoid latch, is don't care
  state_next <= sREAD_CHAR6;

when sREAD_CHAR6 =>
  dctrl_i <= '-'; -- To avoid latch, is don't care
  state_next <= sREAD_CHAR7;

when sREAD_CHAR7 =>
  dctrl_i <= '-'; -- To avoid latch, is don't care
  state_next <= sREAD_STOP;

when sREAD_STOP =>
  dctrl_i <= '-'; -- To avoid latch, is don't care
  state_next <= sSLAVE_IDLE;
  if rising_edge(clk) then -- Checking if ALPIDE writes
    one to end writing, if not assert
    assert (dctrl_out = '1')
    report "Reading from ALPIDE failed. Stop bit
    not '1'." severity ERROR;
  end if;

when sTURN_AROUND =>
  dctrl_i <= '-'; -- To avoid latch, is don't care
  if bmd_cnt = 5 then -- This means ALPIDE chip has taken
    over the line, and we can leave bus turn around (See. p 35 alpide
    manual)
    state_next <= sSLAVE_IDLE; -- Goes to slave idle
  else
    state_next <= sTURN_AROUND;
  end if;

when others =>
  dctrl_i <= '-'; -- To avoid latch, is don't care
  state_next <= sIDLE;

end case;
```
end process;

end architecture;
Appendix K

Alpide Data Module VHDL

This appendix contains the VHDL code for the Alpide Data module. The AXI4 interface is left out as it is more or less generic. The AXI4 interface code can be found in this thesis GitHub page [59].

K.1 Top

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

entity top_data is
  generic(
    address_width : integer := 16
  );

  port(
    clk : in std_logic; -- Input clock
    reset : in std_logic; -- Reset read and write pointer to 0
    data_alpide : in std_logic_vector(7 downto 0); -- Data from Alpide
    rd : in std_logic;
    en_write : in std_logic; -- Enable write to fifo
    en_read : in std_logic; -- Enable read from fifo
    fifo_data_out : out std_logic_vector(31 downto 0);
    full : out std_logic;
    empty : out std_logic;
    half_full : out std_logic;
  );
```

```
K.1. TOP

architecture top_arch of top_data is
  signal data_alpide_sync_i : std_logic_vector(7 downto 0);
  signal new_data_i : std_logic;
  signal data_con_i : std_logic_vector(7 downto 0);
  signal idle_i : std_logic;
  signal wr_i : std_logic;
  signal write_fifo_i : std_logic;
  signal data_fifo_i : std_logic_vector(31 downto 0);
  signal rd_reg : std_logic; -- Used to set rd only high for 1 clock cycle
  signal rd_reg2 : std_logic; -- Used to set rd only high for 1 clock cycle
  signal popped_i : std_logic; -- Signal used to signal out if the FIFO has been popped
  signal addra_i : std_logic_vector(address_width – 1 downto 0);
  signal addrb_i : std_logic_vector(address_width – 1 downto 0);
  signal wea_i : std_logic;
  signal full_i : std_logic;
  signal empty_i : std_logic;
  signal half_full_i : std_logic;
  signal ena_i : std_logic;
  signal enb_i : std_logic;
  signal busy_i : std_logic; -- Indicate when we can pop a new value

component alpide_data_synch
  port( 
    clk : in std_logic;
    data_alpide : in std_logic_vector(7 downto 0);
    data_alpide_synch : out std_logic_vector(7 downto 0);
    new_data : out std_logic
  );
end component;

-- Alpide data filter

component alpide_data_filter
  port( 
    clk : in std_logic;
    data_alpide_synch : in std_logic_vector(7 downto 0);
  );
new_data : in std_logic;
reset : in std_logic;
data_con : out std_logic_vector(7 downto 0); — Data that should be written towards concatenate
idle : out std_logic; — Indicate to concatenate that we are in idle
wr : out std_logic — High when new data is at the line);
end component;

--- Alpide data concatenate

component alpide_data_concatenate
port(
  clk : in std_logic; — Clock of memory, should at least be 40*4MHz
data_con : in std_logic_vector(7 downto 0); — Parallel data bus from Alpide
rd : in std_logic;
reset : in std_logic;
idle : in std_logic; — Idle signal from filter, telling us when to write to fifo regardless of byte count
data_fifo : out std_logic_vector(31 downto 0);
wr : out std_logic);
end component;

--- BRAM

component simple_dual_one_clock
generic(
  address_width : integer
);
port(
  clk : in std_logic;
  ena : in std_logic;
enb : in std_logic;
wea : in std_logic;
addra : in std_logic_vector(address_width - 1 downto 0);
addrb : in std_logic_vector(address_width -1 downto 0);
dia : in std_logic_vector(31 downto 0);
dob : out std_logic_vector(31 downto 0)
);
end component;

--- FIFO controller

component fifo_controller is
```vhdl
K.1. TOP

begin

    ena_i <= en_write;
    enb_i <= en_read;
    full <= full_i;
    empty <= empty_i;
    half_full <= half_full_i;
    wea_i <= write_fifo_i and (not full_i); -- Only write to fifo when fifo is not full, if fifo is full bytes will be lost
    popped <= popped_i;
    busy <= busy_i;

    -- Only high for one clock cycle
    read_pop : process(clk)
        begin
            if falling_edge(clk) then
                rd_reg <= rd;
                rd_reg2 <= rd_reg;
            end if;
        end process;
    rd_i <= rd_reg and (not rd_reg2); -- rd_i goes high for on clock cycle on falling_edge. Thereby only popping one fifo_data at a time

    -- The busy signal is high for two falling edges to ensure that a rd = 0 gets red
    busy_ctrl : process(clk)
        variable hold_busy : std_logic;
        begin
            if rising_edge(clk) then
                if rd_i = '1' then
                    busy_i <= '1';
                    hold_busy := '1';
                elsif hold_busy = '1' then
                    busy_i <= '1';
                    hold_busy := '0';
                else
                    busy_i <= '0';
                end if;
            end if;
```
end process;

-- Popped signal goes high when rd_i is '1', this means we have popped the FIFO
popped_pros : process(clk)
begin
  if reset = '1' then
    popped_i <= '0';
  elsif rising_edge(clk) then
    if rd_i = '1' then -- Switch popped when we are sure a value has been popped
      popped_i <= not popped_i; -- Popped will switch between 0 and 1 every time we have popped a value from the FIFO
    end if;
  end if;
end process;

synch : alpine_data_sync
port map(
  clk => clk,
  data_alpine => data_alpine,
  data_alpine_sync => data_alpine_sync_i,
  new_data => new_data_i);

filter : alpine_data_filter
port map(
  clk => clk,
  data_alpine_sync => data_alpine_sync_i,
  new_data => new_data_i,
  reset => reset,
  data_con => data_con_i,
  idle => idle_i,
  wr => wr_i);

conc : alpine_data_concatenate
port map(
  clk => clk,
  data_con => data_con_i,
  rd => wr_i,
  reset => reset,
  idle => idle_i,
  data_fifo => data_fifo_i,
  wr => write_fifo_i);

bram : simple_dual_one_clock
generic map( address_width => address_width)
port map(
  clk => clk,
  ena => ena_i,
  enb => enb_i,
  wea => wea_i,
  addra => addra_i,
  addrb => addrb_i,
  dia => data_fifo_i,
  dob => fifo_data_out)
K.1. TOP

```vhdl
)

fifo_ctrl : fifo_controller
  generic map(address_width => address_width)
  port map(
    clk => clk,
    reset => reset,
    wr => write_fifo_i,
    rd => rd_i,
    full => full_i,
    empty => empty_i,
    half_full => half_full_i,
    w_addr => addra_i,
    r_addr => addrb_i
  );

end top_arch;
```
K.2 Synchronizer

---Copyright 2018 Matias Gjestvang Greaker.
---Created : 13–12–2018
---Author : Matias G. Greaker <matiasgg@fys.uio.no>
---File : alpide_data_synch.vhd
---Standard : VHDL'2002
---Purpose : Synchronizes the data

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

entity alpide_data_synch is
  port(
    clk : in std_logic;  -- Clock of memory, should at least be 40*4MHz
    data_alpide : in std_logic_vector(7 downto 0);  -- Signals from alpide
  );

  signal reg1 : std_logic_vector(7 downto 0);  -- Signals for synchronizer
  signal reg2 : std_logic_vector(7 downto 0);
  signal data_alpide_synch : out std_logic_vector(7 downto 0);  -- Synched alpide data

  signal data_alpide : in std_logic_vector(7 downto 0);  -- Parallel data bus from Alpide
  signal new_data : out std_logic  -- High when new data is at the line
  );

end entity;

architecture synch_arch of alpide_data_synch is
  -- Signals data filtering
  signal new_data_i : std_logic;  -- Is high one clock cycle when new data is on the data_alpide line
  signal wea_i : std_logic;  -- Internal signal
  signal stable_prev : std_logic;  -- Previous stable value

  signal two_equal : std_logic;
  signal stable_i : std_logic;  -- Signal to ensure we don't read transient signals

end architecture;
K.2. SYNCHRONIZER

begin

data_alpide_synch_i <= reg2; — Synched alpide data
data_alpide_synch <= data_alpide_synch_i;
new_data <= new_data_i;

-- Two register synchronizer

two_reg_synchronizer: process(clk)
begin
  if rising_edge(clk) then
    reg1 <= data_alpide;
    reg2 <= reg1;
  end if;
end process;

-- Monitoring if the data signal has been stable for two clock cycles. To avoid transient data reading

two_clock_monitor: process(data_alpide_synch_i, reg1)
begin
  if data_alpide_synch_i = reg1 then — If data_alpide_synch is equal to reg1 we have had a stable signal for two clock cycles. We assume that no transient period lasts this long.
    stable_i <= '1';
  else
    stable_i <= '0';
  end if;
end process;

-- Storing prev value of data stable, this is to edge detect data stable

stable_reg: process(clk)
begin
  if rising_edge(clk) then
    stable_prev <= stable_i;
  end if;
end process;

-- This process will handle the special case where two equal consecutive bytes are tranfered
-- This will not work for many equal bytes, as the counter in two_consec will eventually skew,
-- But no more than two equal bytes can be expected.

new_data_logic: process(two_equal, stable_prev, stable_i)
begin
  if two_equal = '0' then
    new_data_i <= stable_i and (not stable_prev);
  else
    new_data_i <= '1';
  end if;
end process;
This sets the signal `two_equal` to '1' if we should write even though the previous data matches the current data:

```vhdl
two_consec : process(clk)
    variable stable_cnt : integer range 0 to 8 := 0;
begin
    if rising_edge(clk) then
        if stable_i = '1' then
            stable_cnt := stable_cnt + 1;
            if stable_cnt > 4 then -- We are then sure that we have two equal bytes (Changed this to 4, to low value can cause double read, but to high will cause never to read two equal)
                two_equal <= '1'; -- This will do a write even though we have two equal consecutive bytes
                stable_cnt := 0;
            else
                two_equal <= '0';
            end if;
        else
            two_equal <= '0';
            stable_cnt := 0;
        end if;
    end if;
end process;
end synch_arch;
```


K.3 Filter

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

entity alpide_data_filter is

    port(
        clk : in std_logic; -- Clock of memory, should at least be 40*4MHz
        data_alpide_sych : in std_logic_vector(7 downto 0); -- Signals from alpide
        new_data : in std_logic;
        reset : in std_logic;
        data_con : out std_logic_vector(7 downto 0); -- Parallel data bus from Alpide
        idle : out std_logic; -- Indicate to concatenate that we are in idle
        wr : out std_logic; -- High when new data is at the line and should be written to FIFO
    );

end entity;

architecture filter_arch of alpide_data_filter is

    signal wr_i : std_logic;
    signal idle_i : std_logic;

    type state_type is (READ_IF_NOT_IDLE, READ_N_BYTES); -- State signals
    signal state_reg : state_type := READ_IF_NOT_IDLE; -- State reg will represent the bit just sent from ALPIDE

    signal data_alpide_prev : std_logic_vector(7 downto 0); -- Previous value of alpide data written, used to e.g. not write multiple CHIP
K.3. FILTER

Trailers

begin

    data_con <= data_alpide_sync;
    wr <= wr_i;
    idle <= idle_i;

    -- Process to control when wr_i should go high

state_p : process(clk)
begin

    variable num_bytes : integer range 0 to 3 := 0;

    if falling_edge(clk) then
        if reset = '1' then
            state_reg <= READ_IF_NOT_IDLE;
            data_alpide_prev <= '11111111'; -- Setting this to IDLE in the start, to make sure we read first byte
        else
            wr_i <= '0';
        end if;
    end if;

    case state_reg is
    when READ_IF_NOT_IDLE =>
        if new_data = '1' then
            if data_alpide_synch /= "11111111" and data_alpide_prev /= data_alpide_synch then -- We are now receiving first NEW byte
                idle_i <= '0';
                wr_i <= '1';
                data_alpide_prev <= data_alpide_synch;
                -- Checking if this is the start of a longer than 1 byte data word, in that case read more bytes
                -- regardless of them being 0xFF.
                if data_alpide_synch(7 downto 4) = "1010" then -- We here have the start of a CHIP HEADER, read one more byte regardless of value
                    num_bytes := 1; -- Read one more byte
                    state_reg <= READ_N_BYTES;
                elsif data_alpide_synch(7 downto 4) = "1110" then -- We here have the start of a CHIP EMPTY FRAME
                    num_bytes := 1;
                    state_reg <= READ_N_BYTES;
                elsif data_alpide_synch(7 downto 6) = "01" then -- We have start of data short
                    num_bytes := 1;
                    state_reg <= READ_N_BYTES;
                elsif data_alpide_synch(7 downto 6) = "00" then -- We have start of data long
                    num_bytes := 2;
                    state_reg <= READ_N_BYTES;
                elsif data_alpide_synch(7 downto 4) = "1011" then -- We have chip trailer
                    num_bytes := 0;
                    state_reg <= READ_IF_NOT_IDLE;
                else
                    num_bytes := 0;
                    state_reg <= READ_IF_NOT_IDLE;
                end if;

    end case;
end process;

else
  idle_i <= '1';
end if;

when READ_NBYTES =>
  idle_i <= '0';
  if new_data = '1' then
    wr_i <= '1';
    data_alpine_prev <= "1111111";
    --To make sure we write the data even though this data is equal to the next (That can happen in some cases)
    num_bytes := num_bytes - 1;
    if num_bytes = 0 then  -- Leave state when n
bytes has been written
      state_reg <= READ_IF_NOT_IDLE;
    end if;
  end if;
end when others => state_reg <= READ_IF_NOT_IDLE;
end case;
end if;
end process;
end filter_arch;
K.4 Concatenate

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

entity alpide_data_concatenate is
    port(
        clk : in std_logic;  -- Clock of memory, should at least be 40*4MHz
        data_con : in std_logic_vector(7 downto 0);  -- Parallel data bus from Alpide read are ready to be read
        reset : in std_logic;  -- Reset
        idle : in std_logic;  -- Idle signal from filter, telling us when to write to count
        data_fifo : out std_logic_vector(31 downto 0);  -- 32-bits data towards FIFO
to write
    );
end entity;

architecture con_arch of alpide_data_concatenate is

    -- Signals for FIFO

    signal data_fifo_i : std_logic_vector(31 downto 0);
    signal write_to_fifo : std_logic;
    signal wr_i : std_logic;

    -- Signals for state

    type state_type is (b0, b1, b2, b3);  -- State signals
    signal state_reg : state_type;  -- State reg will represent the bit just sent from ALPIDE

begin
    data_fifo <= data_fifo_i;
    wr <= wr_i;
```

Purpose: Concatenate the data bytes to 32-bit.
process(clk)
begin
  if falling_edge(clk) then
    if write_to_fifo = '1' then
      wr_i <= '1';
    else
      wr_i <= '0';
    end if;
  end if;
end process;

state_p : process(clK)
begin
  if rising_edge(clk) then
    if reset = '1' then
      write_to_fifo <= '0';
      state_reg <= b0;
    else
      write_to_fifo <= '0';
    end if;
    case state_reg is
      -- Saving the byte as little endien
      when b0 =>
        if rd = '1' then
          data_fifo_i(7 downto 0) <= data_con;
          state_reg <= b1;
        end if;
      when b1 =>
        if idle = '1' then
          data_fifo_i(31 downto 8) <= (others => '1')
        end if;
      when b2 =>
        if rd = '1' then
          data_fifo_i(31 downto 16) <= (others =>
          '1'); -- IDLE pad if we dont have any more bytes to read
          write_to_fifo <= '1';
          state_reg <= b0;
        elsif rd = '1' then
          data_fifo_i(23 downto 16) <= data_con;
          state_reg <= b3;
        end if;
      when b3 =>
        if rd = '1' then
          data_fifo_i(31 downto 24) <= (others =>
          '1'); -- IDLE pad if we dont have any more bytes to read
          write_to_fifo <= '1';
          state_reg <= b0;
        elsif rd = '1' then
          data_fifo_i(31 downto 24) <= data_con;
          state_reg <= b0;
          write_to_fifo <= '1';
        end if;
      when others =>
        if rd = '1' then
          data_fifo_i(31 downto 24) <= (others =>
          '1'); -- IDLE pad if we dont have any more bytes to read
          write_to_fifo <= '1';
          state_reg <= b0;
        elsif rd = '1' then
          data_fifo_i(31 downto 24) <= data_con;
          state_reg <= b0;
          write_to_fifo <= '1';
        end if;
    end case;
  end if;
end process;
end if;
when others => state_reg <= b0;
end case;
end if;
end if;
end process;
end con_arch;
K.5 FIFO

BRAM

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

-- Simple Dual-Port Block RAM with One Clock
-- Correct Modelization with a Shared Variable
-- Recommended template, from Xilinx UG901 p.116

entity simple_dual_one_clock is
generic(
   address_width : integer := 16 -- This will set the memory
depth to 2^address_width - 1
);

port(
   clk : in std_logic;
   ena : in std_logic;
   enb : in std_logic;
   wea : in std_logic;
   addr : in std_logic_vector(address_width - 1 downto 0);
   addrb : in std_logic_vector(address_width - 1 downto 0);
   dia : in std_logic_vector(31 downto 0);
   dob : out std_logic_vector(31 downto 0)
);

end simple_dual_one_clock;

architecture syn of simple_dual_one_clock is
type ram_type is array (2**address_width - 1 downto 0) of
std_logic_vector(31 downto 0);
shared variable RAM : ram_type;
begin
   process(clk)
   begin
      if rising_edge(clk) then
         if ena = '1' then
            if wea = '1' then
               RAM(to_integer(unsigned(addr))) := dia;
            end if;
         end if;
      end if;
   end process;
end syn;

end entity simple_dual_one_clock;
```

---

K.5 FIFO
### FIFO controller

---

---Copyright 2018 Matias Gjestvang Greaker.

---Created : 02–01–2019
---Author : Matias G. Greaker <matiasgg@fys.uio.no>
---File : fifo_controller.vhd
---Standard : VHDL’2002
---Purpose : FIFO controller for the BRAM. This code is based upon example from P. CHU p. 284

---

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

entity fifo_controller is
  generic(
    address_width : integer := 16
  );

port(
  clk : in std_logic; -- Input clock
  reset : in std_logic; -- Reset read and write pointer to 0
  wr : in std_logic; -- Initiate write
  rd : in std_logic; -- Initiate read from fifo
  full : out std_logic; -- FIFO is full
  empty : out std_logic; -- FIFO is empty
  half_full : out std_logic; -- FIFO is half full
  w_addr : out std_logic_vector(address_width –1 downto 0); -- Address towards write, address A in BRAM
  r_addr : out std_logic_vector(address_width –1 downto 0); -- Address toward read, address B in BRAM
);
end fifo_controller;

architecture controller_arch of fifo_controller is
  -- Making the register one bit wider, this way we can keep track of if the register is full or not
  signal w_ptr_reg, w_ptr_next : unsigned(address_width downto 0);
  signal r_ptr_reg, r_ptr_next : unsigned(address_width downto 0);
end process;

process(clk)
begin
  if rising_edge(clk) then
    if enb = ’1’ then
      dob <= RAM(to_integer(unsigned(addrb)));
    end if;
  end if;
end process;

end syn;

FIFO controller
```vhdl
    signal full_flag , empty_flag : std_logic;
    signal half_full_flag : std_logic;
    constant fifo_half_size : integer := (2**address_width)/2;

    begin  
    
      -- Pointer registers
      ptr_reg : process(clk, reset)
      begin
        if(reset = '1') then
          w_ptr_reg <= (others => '0');
          r_ptr_reg <= (others => '0');
        elsif rising_edge(clk) then
          w_ptr_reg <= w_ptr_next;
          r_ptr_reg <= r_ptr_next;
        end if;
      end process;

      -- Write pointer next state logic
      w_ptr_next <= w_ptr_reg + 1 when wr = '1' and full_flag = '0' else w_ptr_reg;
      full_flag <= '1' when r_ptr_reg(address_width) /= w_ptr_reg(address_width) and
                      r_ptr_reg(address_width - 1 downto 0) = w_ptr_reg(address_width - 1 downto 0)
                   else '0';

      -- Half full flag
      half_full_logic : process(r_ptr_reg, w_ptr_reg)
      variable valu : integer;
      begin
        if r_ptr_reg(address_width) = w_ptr_reg(address_width) then
          valu := to_integer(r_ptr_reg(address_width - 1 downto 0)) -
                  to_integer(r_ptr_reg(address_width - 1 downto 0));
          if valu > fifo_half_size then
            half_full_flag <= '1';
          else
            half_full_flag <= '0';
          end if;
        else
          valu := to_integer(w_ptr_reg(address_width - 1 downto 0)) -
                  to_integer(w_ptr_reg(address_width - 1 downto 0));
          if valu < fifo_half_size then
            half_full_flag <= '1';
          else
            half_full_flag <= '0';
          end if;
        end if;
      end process;
      half_full <= half_full_flag; -- Writing if FIFO is half full

      -- Write port output
      w_addr <= std_logic_vector(w_ptr_reg(address_width - 1 downto 0));
      full <= full_flag;

      -- Read pointer next state logic (Reading out four bytes at a time, and only writing one byte at a time)
```

K.5. FIFO

```vhdl
r_ptr.next <= r_ptr_reg + 1 when rd = '1' and empty_flag = '0' else r_ptr_reg;
empty_flag <= '1' when r_ptr_reg = w_ptr_reg else '0';

-- Read port output
r_addr <= std_logic_vector(r_ptr_reg(address_width - 1 downto 0));
empty <= empty_flag;

end controller_arch;
```
Appendix L

Measurements from ALPIDE
Figure L.1: Hitmaps for distances 5 mm to 12 mm. Each hitmap was taken with 100,000 trigger signals. The colorbar show the hit recurrence.
Figure L.2: Hitmaps for distances 13 mm to 20 mm. Each hitmap was taken with 100,000 trigger signals. The colorbar show the hit recurrence.
Figure L.3: Cluster size distributions for 100,000 triggers at distances 5 mm to 12 mm. The mean value, $\mu$, and the standard deviation, $\sigma$, is shown in each plot.
Figure L.4: Cluster size distributions for 100,000 triggers at distances 13 mm to 20 mm. The mean value, $\mu$, and the standard deviation, $\sigma$, is shown in each plot.
Table L.1: Faulty pixels found to register hits regardless of radiation environment.

<table>
<thead>
<tr>
<th>Column</th>
<th>Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>859</td>
<td>103</td>
</tr>
<tr>
<td>81</td>
<td>70</td>
</tr>
<tr>
<td>738</td>
<td>137</td>
</tr>
<tr>
<td>558</td>
<td>281</td>
</tr>
<tr>
<td>943</td>
<td>363</td>
</tr>
<tr>
<td>749</td>
<td>247</td>
</tr>
<tr>
<td>955</td>
<td>122</td>
</tr>
<tr>
<td>978</td>
<td>404</td>
</tr>
<tr>
<td>1014</td>
<td>462</td>
</tr>
<tr>
<td>760</td>
<td>367</td>
</tr>
<tr>
<td>128</td>
<td>384</td>
</tr>
<tr>
<td>767</td>
<td>445</td>
</tr>
</tbody>
</table>
References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


