8-2020

Data Processing Electronics for an Ultra-Fast Single-Photon Counting Camera

Jackson Hyde

Washington University in St. Louis

Washington University in St. Louis

McKelvey School of Engineering

Department of Electrical and Systems Engineering

Thesis Examination Committee:
Matthew D. Lew
Shantanu Chakrabartty
Ed Richter

Data Processing Electronics for an Ultra-Fast Single-Photon Counting Camera

by

Jackson Richard Hyde

A thesis presented to the McKelvey School of Engineering
of Washington University in partial fulfillment of the
requirements for the degree of

Master of Science

August 2020
Saint Louis, Missouri

Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my advisor, Dr. Matthew D. Lew. His willingness to offer an undergraduate student this research opportunity has shaped my time at Washington University. His creative approach to problem solving has helped me over multiple significant hurdles throughout the two years I've worked on this project. As my academic advisor, he has helped me choose a breadth of classes that made my educational experience here interesting and diverse. Without his patience, ingenuity, and wisdom, I could not have completed this thesis or pursued this degree.

I would also like to thank Dr. James Buckley, whose collaboration with Dr. Lew formed the vision for the imaging system and thus the necessity of this project. He helped me countless times with the fundamentals of hardware design, the complexities of the existing readout electronics, and the broader design questions I faced throughout the project.

Finally, I would like to thank Paul Dowkontt, Richard Bose, Ed Richter, and William D. Richard, whose teaching and advice on hardware design gave me the specific skills and resources I needed to accomplish a design project of this scale.

Jackson Richard Hyde

Washington University in Saint Louis
August 2020

Dedicated to my wife.

ABSTRACT OF THE THESIS

Data Processing Electronics for an Ultra-Fast Single-Photon Counting Camera

by

Jackson Richard Hyde

Master of Science in Electrical Engineering

Washington University in St. Louis, August 2020

Research Advisor: Dr. Matthew D. Lew

Localizing photon arrivals with high spatial (megapixel) and temporal (sub-nanosecond) resolution would be transformative for a number of applications, including single-molecule super-resolution fluorescence microscopy. Here, the Data Processing Field Programmable Gate Array (FPGA) is developed as an ultra-fast computational platform built on an FPGA for a microchannel plate (MCP)-photomultiplier tube (PMT) based single-photon counting camera. Each photon is converted by the MCP-PMT into an electron cloud that generates current pulses across a 50×50 cross-strip anode. The Data Processing FPGA executes a massively parallel center-of-gravity coordinate determination algorithm on the digitized current pulses to determine a 2D position and time of arrival for each charge cloud. The coordinates are relayed to a computer via a Gigabit Ethernet link. The system achieves a local photon throughput of 1.04 MHz. If photons arrive continuously with an average spacing of 1.5 µs across a 10×10 portion of the cross-strip anode, the system accurately localizes photons in both space and time and achieves a spatial precision of 4.1 µm (62 times smaller than the anode pitch) and a temporal precision of 55 ps (at 500 MHz digitization).

Chapter 1

Background

Imaging single molecules reveals fundamentally important information about the evolution of processes inside cells and in chemical reactions. Single-molecule imaging [1, 2, 3] has been used to study, for example, the replication of DNA [4], DNA mismatch repair [5], the generation of high-density lipoproteins that prevent atherosclerosis [6], and the aggregation of amyloid structures which are signature indicators of Alzheimer's [7, 8]. Chemical reactions generating fluorescence have also been the subject of investigations utilizing single-molecule imaging [9, 10]. The underlying biological and chemical processes in all these works are driven by the interactions of individual molecules 1-10 nanometers in size, motivating the design of imaging systems with nanometer spatial resolution. Many applications, detailed in section 1.1, not only require nanometer spatial resolution but also sub-nanosecond temporal resolution – or the ability to timestamp the arrival of photons to the hundreds or even tens of picoseconds.

To address these needs, which may be roughly described as nanoscale spatiotemporal resolution at a high frame rate, a micro-channel plate photomultiplier tube (MCP-PMT) based imaging system is being developed in a joint effort by the Lew and Buckley labs at Washington University in St. Louis. It will feature megapixel spatial resolution, sub-nanosecond temporal resolution, a local photon count rate exceeding 1 MHz, and the ability to capture simultaneous, spatially separated photon arrivals. This thesis describes the development and testing of a massively parallel digital computational platform on an FPGA which interfaces with the MCP-PMT. This platform is a necessity because the massively parallel array of high-speed data streams coming from the detector cannot be efficiently handled by traditional networking solutions like switches or routers. Further, the raw data generated would be difficult to handle in post-processing even if there was a networking solution that could aggregate it into a single storage device. To address these dual concerns, the FPGA design presented here processes the massively parallel data from the MCP-PMT system, computes the position and time-of-arrival of photons with nanoscale spatiotemporal resolution, and transmits results via a Gigabit Ethernet link to a downstream computer.

The remainder of this chapter explains some of the key issues surrounding the work of this thesis. In Section 1.1, I introduce imaging applications which necessitate high-performance systems. In Section 1.2, I describe the novel MCP-PMT camera of which this design is a part. In Section 1.3, I motivate the necessity and communicate the difficulty of this project. Section 1.4 provides an overview of the FPGA design and the remainder of the thesis.

1.1 Imaging Applications

Imaging applications that require high temporal and spatial resolution across a wide field of view pose a challenge to traditional photon counting detectors. Researchers analyzing the dynamics of micro-scale cellular processes look at movies generated by stitching together multiple images generated by the imaging system. In order to get a deep understanding of the single-molecule mechanisms that drive these processes, each frame needs to resolve the position of the individual molecules involved. The field of view of each high-resolution frame must be wide enough for the researchers to see the full extent of the process evolving within the cell. Finally, the required frame rate is dictated by the speed of the interactions being studied: if the cellular dynamics are fast enough, the researchers may face a trade-off between highly resolved, wide field images and the speed at which they can be generated.

Imaging conditions that necessitate high performance systems are not limited to biological applications. For example, the development of imaging systems that can "see around corners" depends in part on resolving the time delay of reflected photon arrivals to gauge distance [11, 12]. By computing the delay in photon arrivals to approximately 100 ps, the photon travel distance can be resolved to 3 cm. This requires the use of sensors with extremely high temporal resolution. However, the usefulness of the image is dependent on the field of view and spatial resolution. These competing concerns drive up the price of the imaging system and motivate the creation of complicated reconstruction algorithms in software to mitigate cost, as in [11] and [12].

Fluorescence-lifetime imaging microscopy (FLIM) is a biological imaging technique which similarly requires precise timing of single photons across a wide field of view. FLIM measures the delay between absorption and emission of photons by fluorescent molecules, or fluorophores, in a biological sample. This delay, or lifetime, ranges from 1 to 10 ns and varies with cellular factors like pH levels, ion concentrations, glucose levels, oxygen concentrations, and more [13]. Thus, visualizing the distribution of lifetimes in a biological sample can convey key information to researchers about cellular dynamics and aid the investigation of disease evolution.

In FLIM, fluorescence lifetime differences in the sub-nanosecond regime correlate to differences in the biological factors listed above. To create a useful image, then, requires spatially resolving fluorophores to the order of one to ten nanometers and temporally resolving their fluorescence to the order of tens or hundreds of picoseconds. This must be done over a wide field of view at a frame rate fast enough to track the evolution of the underlying process. Many approaches to this problem involve scanning either a single excitation spot or, as in [14], a holographically generated array of excitation spots. These systems are limited in their ability to image high speed cellular processes by the time it takes to scan the entire image. Other approaches involve an MCP-PMT system similar to the one being developed, either with a delay-line [15] or cross-strip [16] anode. The general advantage of the MCP-PMT system being developed by the Lew and Buckley Labs over these similar approaches is its higher photon efficiency than [15] and [16] and significantly higher spatial resolution than [15] due to lower dead-time per output event and less readout noise, respectively.

1.2 The MCP-PMT Imaging System

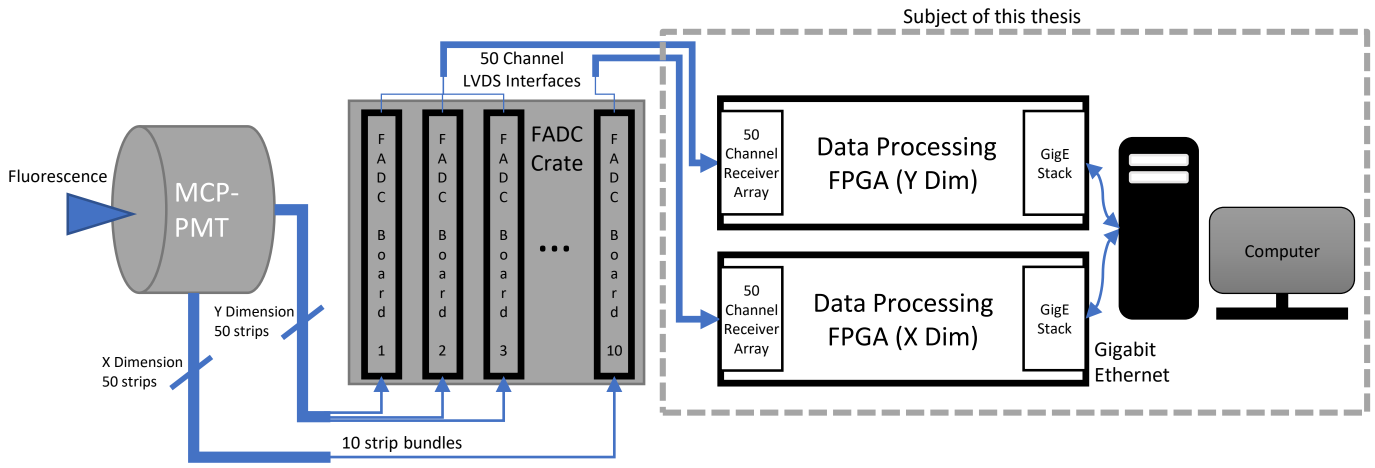

Figure 1.1: The MCP-PMT Imaging System. The wide-field fluorescence from the object plane is focused into the MCP-PMT's photocathode, eventually generating measurable current pulses on the strips (Section 1.2.1). The FADC Crate features 10 FADC Boards that digitize the generated current pulses on strips of the anode. Each of the 10 Channels on a given FADC Board drives an LVDS link. The Data Processing FPGAs interface with each channel's LVDS link and send computed coordinates to the computer via a Gigabit Ethernet link.

The novel imaging system being designed for this project, depicted in Figure 1.1, features a micro-channel plate photo-multiplier tube (MCP-PMT) digitized by a crate of Flash Analog-to-Digital Converter (FADC) boards. The FADC boards send samples via Low-Voltage Differential Signaling (LVDS) interface to the Data Processing FPGA, which is the subject of this thesis. The Data Processing FPGA detects incident photons and computes their position and time of arrival. Figure 1.2 describes the way each module receives and outputs information. This approach is motivated by the shortcomings of other imaging techniques for wide-field FLIM. Red-shifted single photon avalanche detectors (ReSPADs) are too expensive to use beyond arrays of 12x12, which limits them to imaging a small field of view, while charge-coupled device (CCD) and complimentary metal-oxide semiconductor (CMOS) cameras are too slow to measure fluorescence lifetime on their own [17].

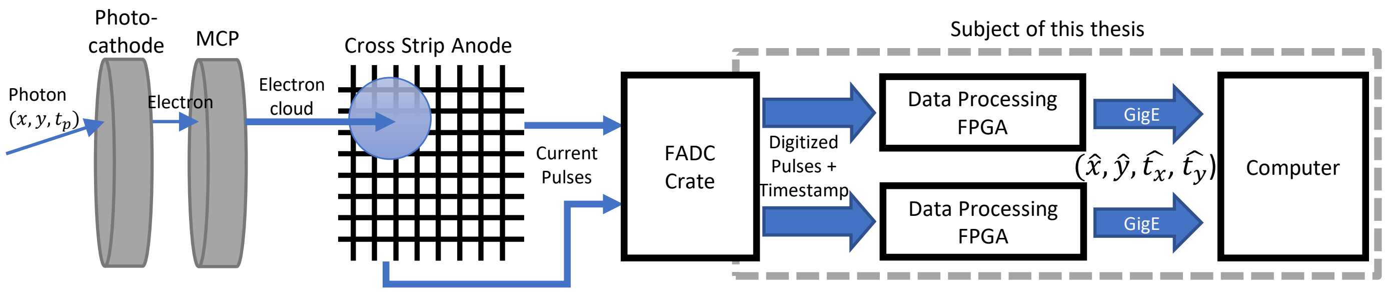

Figure 1.2: Data Flow of the MCP-PMT Imaging System. A single photon emitted from the biological sample collides with the photocathode at position (x, y) and time t_p and is converted into an electron. The electron is accelerated into an MCP, producing an electron cloud that collides with the cross-strip anode. Each impacted strip generates a current pulse (proportional to the collected charge) that is digitized and timestamped by the FADC crate. The FADC crate transmits the samples and timestamp to the Data Processing FPGAs which compute the coordinate $(\hat{x}, \hat{y}, \hat{t}_x, \hat{t}_y)$. $(\hat{x}, \hat{y})$ is an estimate of the cloud's center position, or centroid, on the cross strip anode. $\hat{t}_x$ and $\hat{t}_y$ are each dimension's estimate of the time of digitization of the center of the cloud's induced current pulses. These coordinates are transmitted to the computer via a Gigabit Ethernet link.

1.2.1 The MCP-PMT

Fluorescence emitted by a biological sample is focused onto a photo-cathode that converts each photon into an electron with a quantum efficiency that varies with frequency [18, 19]. The resultant electron is accelerated across a high-voltage potential difference and enters a micro-channel plate [20, 21, 22] featuring a gain of approximately 10^6, multiplying it into an electron cloud. This electron cloud is accelerated across another high-voltage potential and collides with the backplane of the MCP-PMT system, the cross-strip anode. The cross-strip anode is comprised of two layers of fifty copper strips, with one set oriented in the along the x- and one in along the y dimension. It has an active area of approximately 75% and a pitch of approximately 250 µm. Because copper is a conductor, electrons colliding with the strip are captured and, en masse, generate a measurable flow of charge. To accurately measure the arrivals of individual photons, the generated electrons and electron clouds must be undisturbed as they are accelerated towards the backplane. To accomplish this, the entire system described thus far is housed in an extremely low pressure vacuum chamber.

Although the signal generated by a single electron cloud's collision is measurable, it is very small. The magnitude of a single strip's current pulse is a good indicator of the signal strength, so I will compute a rough estimate of the peak current of the pulse on a strip. For the sake of brevity, a uniform spatial distribution of charge on the cross-strip anode is assumed here. Later, as in Chapters 2 and 4, a model of each cloud's charge deposition as a 3-D normal distribution will be presented. Equation 1.1 shows how the gain of the MCP, G, active area ratio of the anode, A, and number of strips involved in the collision, S, relate to the charge deposition on a single strip, C_{strip}. All parameters of Equation 1.1 are unitless.

$$C_{strip} = 1.6 \times 10^{-19}\,\mathrm{C}\frac{G \cdot A}{S} \tag{1.1}$$

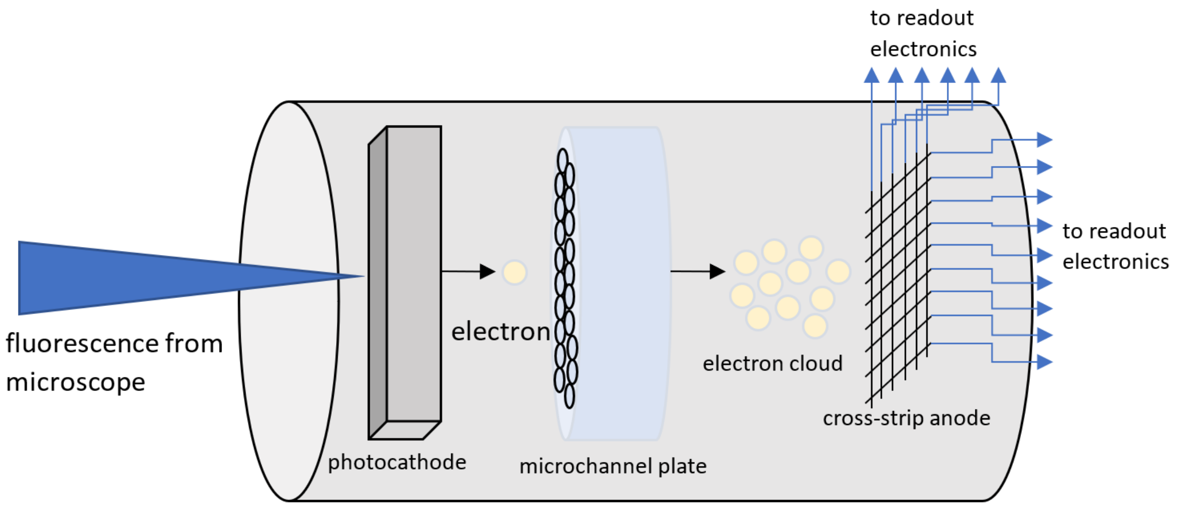

Figure 1.3: The MCP-PMT. A single fluorescence photon from the biological sample is converted to an electron cloud via a microchannel plate. The cloud collides with a portion of the cross-strip anode, creating measurable current pulses on each strip. The anode features 50 strips in each dimension.

Under the assumption of a 10^6 gain from [22], a 75% active area of the cross-strip anode, and 10 strips involved in a collision, the total charge deposited on a single strip is computed in Equation 1.2.

$$C_{strip} = 1.6 \times 10^{-19}\,\mathrm{C}\frac{10^6 \cdot 3/4}{10} = 1.2 \times 10^{-14}\,\mathrm{C} \tag{1.2}$$

Assuming this total charge is deposited over time in a normal distribution with a FWHM of 1 ns [23], the peak value of the distribution represents the maximum instantaneous current, which is the parameter of interest. The standard deviation is related to the FWHM by Equation 1.3 [24]. The peak value is then the scaling factor, 1.2×10^{-14} C, divided by the standard normalizing factor of the normal distribution, $\sqrt{2\pi}\sigma$ as shown in Equation 1.4.

$$\sigma = \frac{1\,\mathrm{ns}}{2\sqrt{2\ln 2}} = 424.66\,\mathrm{ps} \tag{1.3}$$

$$I_{peak} = \frac{C_{strip}}{\sqrt{2\pi}\sigma} = \frac{1.2 \times 10^{-14}\,\mathrm{C}}{\sqrt{2\pi}424.66\,\mathrm{ps}} = 11.27\,\mu\mathrm{A} \tag{1.4}$$

Thus, the peak current on a strip under the assumptions above is 11.27 µA. The inaccuracy in this back-of-the-envelope calculation is in the assumption that charge clouds emitted from the MCP will be uniformly distributed across the portion of the cross-strip anode they collide with. The more complicated model of the following sections assumes the instantaneous charge deposition of each cloud will follow a 2-D normal distribution. Nevertheless, this number is an approximate value that communicates how small the current pulse on each strip will be. The low signal power and highly-parallel nature of the pulses motivates the inclusion of the high-powered amplifier system presented in the following section.

1.2.2 The FADC Readout Electronics

The current pulses generated on each strip by a given cloud are, as shown above, very small. They are also very fast, with a width of approximately 1 nanosecond. A single cloud might collide with 10 strips, producing 10 current pulses. The rate of cloud incidences across the entire anode depends on experimental conditions including the average power of the pulsed laser and the number of fluorophores in the field of view, but even a conservative estimate of 100,000 photons/second puts the total pulse rate on all 100 strips at 1 MHz. As one of the primary goals of this system over other approaches is increased photon efficiency, a high powered digitization system is necessary to capture the wealth of spatiotemporal information deposited on the cross-strip anode.

The FADC system is well equipped for this task. Originally developed in the lab of Jim Buckley by Paul Downkontt for the the VERITAS-4 atmospheric Čerenkov telescope (ACT)

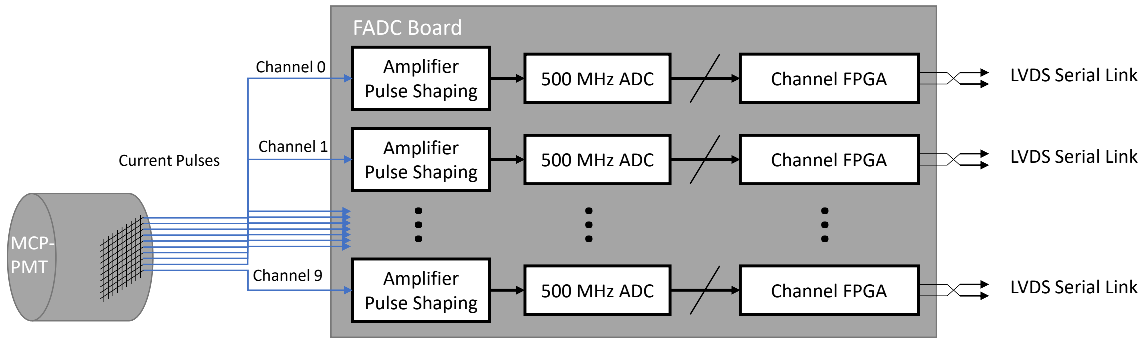

Figure 1.4: The FADC Board. An analog stage amplifies and shapes the current pulses from the MCP-PMT system. 500 MSps, 8-bit ADC's continuously digitize the amplified signal. Each channel's FPGA initiates a message on the LVDS serial link if the digitized signal crosses above a programmable threshold. An FADC Crate is stuffed with 10 identical FADC boards.

array [25], the system implements an array of highly synchronized 500 MSps ADC's. In the VERITAS array, high-speed ($\sim$5 ns) pulses generated by radiation from gamma rays across up to 1000 channels were continuously digitized and stored into buffers that store multiple microseconds of data. An elaborate triggering network and a VMEbus acquisition computer processed and displayed the data from a programmable window. While these features were essential for the original VERITAS application, for the MCP-PMT a reduced feature set is used. In this application, the FADC system will be housed in a single 10-board crate. A single board, described visually in Figure 1.4, features 10 channels, allowing a single FADC crate to interface to each of the 100 channels. The boards have length-controlled traces for synchronous distribution of clocking and control signals, which is essential for the sub-nanosecond precision requirement of the MCP-PMT system.

Every strip is continuously digitized by a 500 MSps, 8-bit ADC. The data is made available to the Channel FPGA four samples at a time with an eight nanosecond clock. The Channel FPGA compares each of the four samples to a programmable threshold. If any of the samples cross the threshold, it packages a programmable window of samples before and after the crossing, along with a two byte timestamp, into a message. The timestamp corresponds to the time of the first sample in the message. The Channel FPGA sends this message out via an LVDS serial link to the Data Processing FPGA.

1.3 Motivation

The MCP-PMT project as a whole is an endeavor to provide a combination of spatiotemporal resolution, photon efficiency, near-megapixel image size, and design flexibility that is not available in any other imaging system available today. The frontiers of single-molecule imaging mentioned in the introduction are being explored with the latest and greatest imaging systems manufactured by highly competitive companies. This novel design will enable researchers to study cellular processes with truly astonishing levels of detail. My contribution is the bridge between the MCP-PMT vacuum chamber/digitization electronics and the computer used by the scientist. It therefore sits in the very challenging space that requires an intimate knowledge of the detector physics, the digitization equipment, massively parallel FPGA design, digital computation, and high-speed networking. This design is the result of a combination of scientific research, engineering work, and programming that was necessary to "cross the bridge" between experiment and experimenter.

The problem statement that was presented to me by Dr. Matthew Lew and Dr. Jim Buckley was essentially a black box that receives digitized current pulses from 100 FADC channels and generates a stream of $(\hat{x}, \hat{y}, \hat{t})$ coordinates – the estimates of the center position and time of each cloud generated inside the MCP-PMT. Everything from the choice of FPGA board and the interface with the FADC crate, down to the algorithm used to generate the coordinates was undecided. Handling the sheer bandwidth generated by the FADC crate – 12.5 Gbps (assuming 125 Mbps links from each channel) was a serious task and required implementing a massively parallel front-end receiver interface. Mapping an algorithm into hardware requires multiple cycles of software development and evaluation followed by hardware implementation and optimization. Developing an algorithm that could be run at high clock speeds on the FPGA fabric posed a significant challenge to my basic understanding of fixed-point digital computation. Finally, figuring out how to get my computed results to a computer as quickly – and with as least overhead – as possible required learning about a variety of networking solutions and handling the bottom two layers of the OSI model.

1.4 Overview

I developed a Data Processing FPGA implemented on the ML605, a high-performance development board featuring the Xilinx® Virtex-6 FPGA. The Virtex-6 FPGA on board the ML605 has ample resources for both routing and storage as well as computation. The Data Processing FPGA depicted in Figure 1.5 is a massively parallel computational platform that exploits the inherent parallelism of FPGAs to process the data generated by the FADC crate into a stream of position and time coordinate pairs, one for each photon. A single Data Processing FPGA is responsible for one dimension of the cross-strip anode, or 50 strips. Two Data Processing FPGAs are therefore used in parallel.

I designed a low-overhead serial protocol implemented over LVDS that allows each FADC Channel FPGA to transmit data at 125 Mbps. On the front end of the Data Processing FPGA is an array of Channel Receiver modules that utilize the hardened SERDES deserialization logic embedded in the Virtex-6 to deserialize the transmitted messages. Each Receiver also implements a math block that computes preliminary coordinates on the received samples.

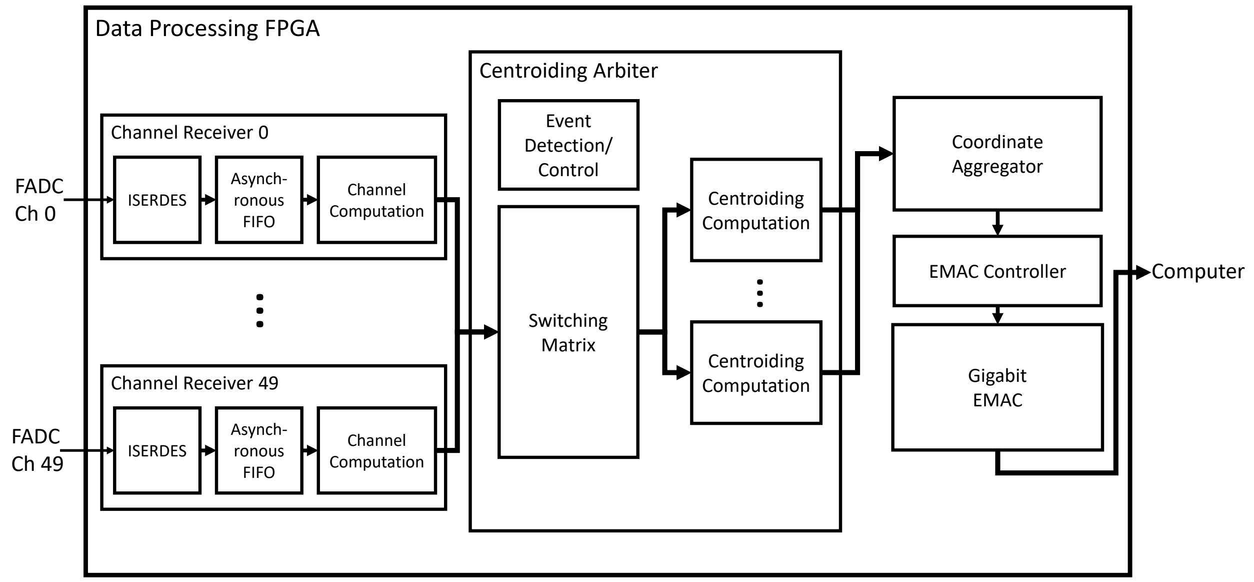

Figure 1.5: Data Processing FPGA Hierarchy. The Data Processing FPGA is responsible for one dimension (50 FADC Channels). An array of 50 Channel Receivers interfaces one-to-one with the 50 FADC Channel FPGAs in a single dimension. Each Channel Receiver (Section 3.3) consists of control logic (not shown) and a hardened Xilinx ISERDES (for deserialization) block feeding samples to the Channel Computation module via an Asynchronous FIFO. The Channel Computation modules (Section 3.3.2 & 3.3.3) drive their results onto a large bus which enters the Centroiding Arbiter (Section 3.5). Inside the Arbiter, Event Detection and Control logic watches the bus and assigns the computation of a single cloud's centroid to one of a number of Centroiding Computation (Section 3.5.2) modules. These modules first read the required Channel Computation results by indexing the massively parallel switching matrix and then use the results to compute the cloud's centroid. A Coordinate Aggregator (Section 3.6) groups the centroids into "packets" and sends them to the Ethernet Medium Access Control (EMAC) Controller for packaging into an Ethernet frame. The Xilinx Gigabit EMAC (Section 3.7) transmits the frame to a downstream computer.

A Centroiding Arbiter groups hits on adjacent strips into "clouds" and dispatches the computation of the final position and time coordinates to an array of math blocks. The stream of coordinates generated by the math blocks is packaged into a Gigabit Ethernet stream by a combination of custom logic, the Xilinx Ethernet MAC IP Core, and the ML605's onboard PHY. A downstream computer will then save and interpolate the coordinate streams generated by each dimension's Data Processing FPGA to create a list of $(\hat{x}, \hat{y}, \hat{t})$ coordinates corresponding to photon arrivals in the MCP-PMT system.

In Chapter 2 of this thesis, I will present the centroiding algorithm. Chapters 3 and 4 present how I implemented this coordinate-estimating computation on an FPGA and how I tested my implementation, respectively. Chapter 5 presents the state of the entire project and an outlook on remaining work on my FPGA design and other components in the system.

Chapter 2

The Centroiding Algorithm

The fundamental operation of each of the two Data Processing FPGA's is to compute a position and time corresponding to the arrival time of the center position of each incident electron cloud, a process hereafter referred to as "centroiding." Section 2.1 presents an overview of the model of an electron cloud. Sections 2.2-2.4 give an overview and component-level view of the Centroiding Algorithm.

2.1 Cloud Model

The theoretical model is built around the assumption that as charge clouds collide with the cross-strip anode, their charge distribution is normally distributed in both space and time. That is, the instantaneous charge deposition on the cross-strip anode at position (x, y) and time t – denoted $C(x, y, t)$ – is a circularly symmetric 2-D Gaussian distribution in position and a 1-D Gaussian distribution in time:

$$C(x,y,t) = \frac{1}{w_t w_{xy}^2} \exp\left(-\frac{(x-\mu_x)^2 + (y-\mu_y)^2}{2w_{xy}^2}\right) \exp\left(-\frac{(t-\mu_t)^2}{2w_t^2}\right) \tag{2.1}$$

Table 2.1 gives the meaning of each parameter of the charge cloud.

Table 2.1: Charge Cloud Parameters.

Parameter	Meaning
A	Scaling constant
(μ_x, μ_y)	Center Position
w_{xy}^2	Spatial Variance (Width)
μ_t	Center Time
w_t^2	Temporal Variance (Width)

Of interest is not the charge deposition, but the instantaneous current on a strip that is sampled by the ADCs. To compute this value accurately, a 10×10 version of the cross-strip anode is modeled. Figure 2.1 gives a schematic of this "test version" of the cross strip anode.

Using the model of the anode in Figure 2.1, it becomes clear that the current induced at time t on a strip is simply the integral of I over the exposed area of the strip. The ith X strip's current is therefore simple to express:

$$I_{X,i}(t) = \int_{x_i}^{x_i+W_x} \int_{y_0}^{y_0+L} C(x,y,t)\, dy\, dx \tag{2.2}$$

Because Y strips are essentially piecemeal integrals in the X dimension, they are more complicated to express. The jth Y strip's current is a sum of 9 exposed regions:

$$I_{Y,j}(t) = \int_{y_j}^{y_j+W_y} \sum_{i=0}^{8} \int_{x_i+W_x}^{x_{i+1}} C(x,y,t)\, dx\, dy \tag{2.3}$$

Because $C(x,y,t)$ is separable in each dimension, the X and Y integrals in both sets of strips can be computed independently. The result in either dimension is therefore a constant times the temporal component of C. This means that all strips share a common pulse envelope, $\exp\left(-(t-\mu_t)^2/2w_t^2\right)$, scaled by a strip-dependent scaling factor. Although the strip's current

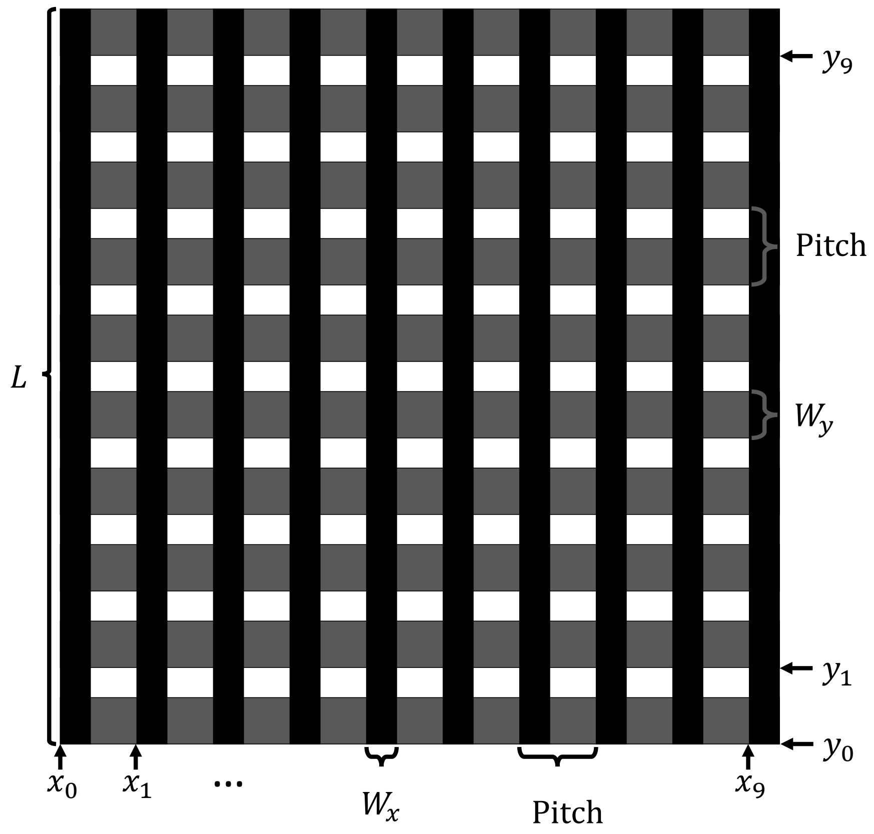

Figure 2.1: Model of Cross-Strip Anode. This schematic of a 10×10 cross-strip anode is taken from the actual design of the cross-strip anode in the Buckley lab but is cropped to $1/5 \times 1/5 = 1/25^{\text{th}}$ of the area of the full-size version. Note that both dimensions have identical pitch; it is merely the strip widths W_x and W_y that differ. W_y is wider to account for the vertical X strips covering the horizontal Y strips. The length of an X strip is given as L. The left edge positions of each X strip are denoted by x_i, while the bottom edge positions of each Y strip are denoted by y_i.

may reasonably be modeled as a smooth Gaussian pulse, the ADC's samples will be noisy estimations of the induced voltage in an analog amplifier. Additionally, the FADC's pulse shaping electronics spread the current pulse over approximately 5-15 nanoseconds to ensure multiple samples are taken. Because the system is not operational, the characteristics of the introduced noise are unknown. For this reason, zero-mean Gaussian noise is applied to the current pulses to produce the measured discrete voltages:

$$V_{X,i,t} = A \cdot I_{X,i}(t) + Z_{i,t} \tag{2.4}$$

$$V_{Y,j,t} = A \cdot I_{Y,i}(t) + Z_{j,t} \tag{2.5}$$

The noise on the ith strip at time t has zero mean and variance σ_z^2:

$$Z_{j,t} \sim \mathcal{N}(0, \sigma_z^2) \tag{2.6}$$

The scaling constant A maps the small fractional value of the currents $I_{X,i}(t)$ and $I_{Y,j}(t)$ into the range $(0, 255)$. It is chosen so that the highest voltages measured in $V_{X,i,t}$ and $V_{Y,j,t}$ (Equations 2.4 & 2.5) are approximately the upper-middle portion of that range: $(128 - 200)$. This is because, in the actual MCP-PMT system, the charge content in a cloud will vary considerably. The MCP gain will be tuned so that the average cloud generates peak voltages in the middle of the ADC's $(0, 255)$ range, avoiding signal saturation. A may therefore be thought of as a gain on a current amplifier in the FADC electronics that transforms the small current pulses (Section 1.2.1) into voltages in the operating range of the ADCs.

2.2 Overview of the Centroiding Algorithm

The Centroiding Algorithm computes an estimate of the cloud's position and time centroid from a single dimension's perspective, for example the X, $(\hat{x}, \hat{t}_x)$. $\hat{x}$ is the algorithm's estimate of the cloud's center position and $\hat{t}_x$ is the estimate of the center of the cloud's arrival time. The algorithm is shown as an abstract component of the larger MCP-PMT system in Figure 2.2.

When a cloud collides with the cross-strip anode, it simultaneously generates current pulses on strips in both the X and Y dimensions. Information about the X position of the cloud's arrival is therefore isolated from information about the Y position, and they can be estimated independently of each other. The time of arrival is, on the other hand, encoded into pulses in both dimensions. As shown in Figure 2.2, this means that both the X and the Y instance of the centroiding algorithm will generate independent estimates of the cloud's time centroid which are interpolated later (on the host computer of Figure 1.2). This interpolation could be used to resolve simultaneous cloud arrivals, however it is not in the scope of this thesis and is discussed as a subject of future work in Chapter 5.

Investigations into the optimal centroiding algorithm for an MCP and cross-strip anode system have found success using an "interpolated convolution" approach [26]. The interpolated convolution method features a spatially-differentiated Gaussian kernel that is determined analytically and then loaded onto the FPGA. The kernel is convolved across the spatial dimension's pulse heights. For each incident cloud, the kernel must be selected to fit the width of the cloud. The resultant discrete convolution's zero-crossing is interpolated and taken to be the cloud's center position. This method proved successful in reducing fixed-position error, where the error in the computed centroid varies with the cloud's position relative to strip

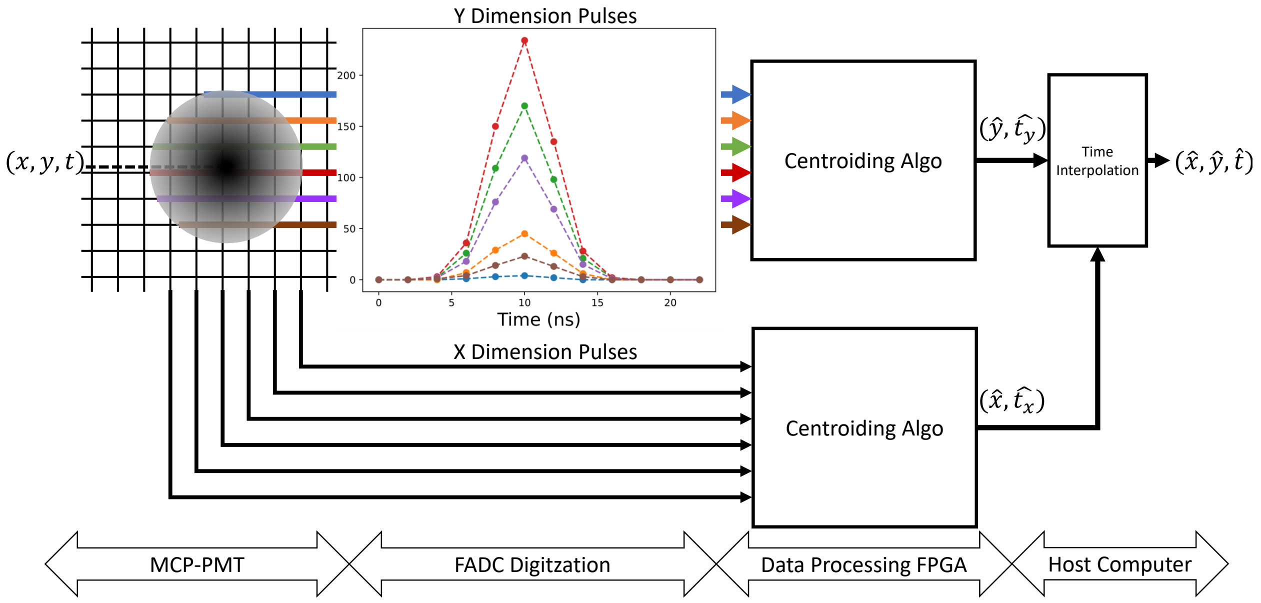

Figure 2.2: System-level View of the Centroiding Algorithm. The algorithm receives as its input the FADC-sampled current pulses generated by a single dimension of the MCP-PMT's cross-strip anode, for example the Y dimension, and computes an estimate of the position centroid and time centroid of the cloud's arrival, $(\hat{y}, \hat{t}_y)$. The process is simultaneously carried out in the X dimension, producing its own estimate of the cloud's arrival, $(\hat{x}, \hat{t}_x)$. A temporal interpolator running on the host computer combines the independently generated coordinates into a final result, $(\hat{x}, \hat{y}, \hat{t})$.

pitch. The method for determining time of arrival is loosely described as an "FIR filter... similar to a digital constant fraction discriminator" which was capable of achieving sub-clock cycle resolution, but no additional implementation details are given [26].

As mentioned in [26], a weighted-average center of gravity approach [27] is simpler to implement on an FPGA and can be used for determining both the position and time centroid of the cloud. The simplicity *does not* come at a measurable cost to accuracy or precision as shown in Chapter 4. Computing preliminary coordinates on each received pulse in parallel as they arrive minimizes the complexity of the "brain" of the system which computes the final position and time centroids. This allows faster system clock speeds that ultimately reduce local deadtime and increase system throughput (Chapter 3). Further, as the vacuum chamber assembly is still under development and has not been characterized, choosing a more generic approach avoids wasted front-end development and allows for optimizing the algorithm once the system is operational. Finally, several crucial computational blocks are reused multiple times throughout the algorithm, meaning extended hardware development time could be focused on creating a set of plug-and-play arithmetic components that interface seamlessly.

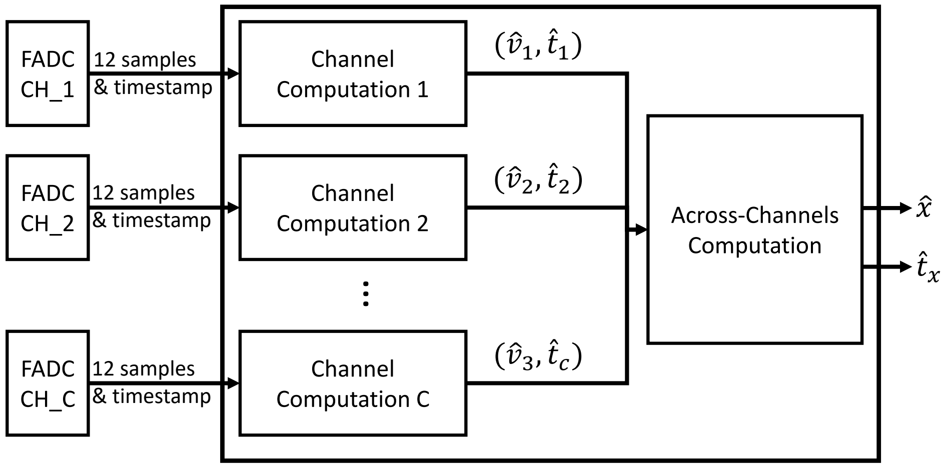

Figure 2.3: Structure of the Centroiding Algorithm. The centroiding algorithm is divided into two stages. The first is an array of C identical single channel computations, where C is the number of strips in the X-dimension of the cross-strip anode that received a current pulse from the incident cloud. Each channel computation generates preliminary coordinates $\hat{v}$ (average pulse height) and $\hat{t}$ (time centroid of pulse). The second stage is an across-channels computation that uses each of the preliminary coordinates to generate an estimate of the position and time centroids of the cloud, $(\hat{x}, \hat{t}_x)$.

The aforementioned split between the preliminary coordinates calculated independently for each channel and the final estimate is described visually in Figure 2.3. Section 2.3 describes how the single channel computations produce an estimate of each pulse's average magnitude and time centroid. Section 2.4 explains how Figure 2.3's "Across-Channels Computation" uses the channel computation results to create a final estimate of the cloud's position and time centroids.

2.3 Single Channel Computation

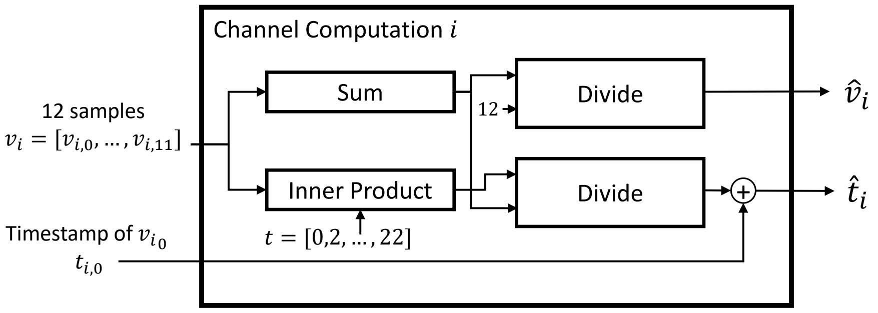

Figure 2.4: The Channel Computation. The ith instance of the channel computation algorithm in the X dimension generates an estimate of the pulse's time centroid, $\hat{t}_i$, and an estimate of the received pulse's average height, $\hat{v}_i$. The average pulse height is computed by a simple sum and division. Due to the 500 MSps ADC's, the twelve input samples correspond to twelve sample times ranging from 0 to 22 ns. A center-of-gravity centroiding operation is performed on the sample times using the samples as weights. This entails dividing the inner product of the coordinate vector and the weight vector by the sum of the elements of the weight vector. The result is an offset from the system time of the first sample, $t_{i,0}$.

The single channel computation takes as input the vector $v_i = [v_{i,0}, v_{i,1}, \ldots, v_{i,11}]$ which holds the 12 samples received from the ith channel. It also uses $t_{i,0}$, the timestamp of $v_{i,0}$ in nanoseconds. It computes preliminary coordinates as shown in Figure 2.4 and described below.

$\hat{v}_i$, the average height of pulse i, is simply the sum of the samples in the vector v_i divided by 12.

$$\hat{v}_i = \frac{\sum_{j=0}^{11} v_{i,j}}{12} \tag{2.7}$$

$\hat{t}_i$ estimates the center time of the pulse. The computation used is a center-of-gravity operation (a weighted average) which takes the sample magnitudes as weights on the corresponding sample times. The sample times range from 0 to 22 ns due to the 500 MSps rate of the ADC's.

The result is added to the system-time of the first sample, $t_{i,0}$.

$$\hat{t}_i = t_{i,0} + \frac{\sum_{j=0}^{11} 2jv_{i,j}}{\sum_{j=0}^{11} v_{i,j}} \tag{2.8}$$

2.4 Across-Channels Computation

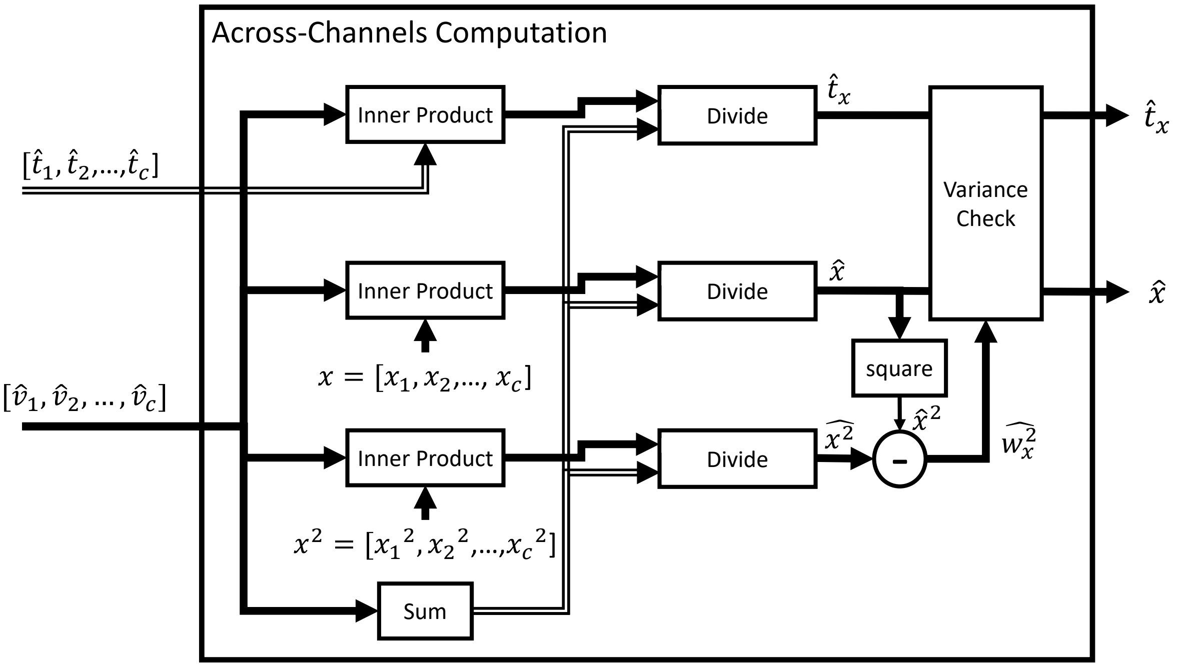
Across-Channels Computation
$[\hat{t}_1, \hat{t}_2, \dots, \hat{t}_c]$
$[\hat{v}_1, \hat{v}_2, \dots, \hat{v}_c]$
Inner Product
Inner Product
Inner Product
Sum
$x = [x_1, x_2, \dots, x_c]$
$x^2 = [x_1{}^2, x_2{}^2, \dots, x_c{}^2]$
Divide
Divide
Divide
$\hat{t}_x$
$\hat{x}$
$\widehat{x^2}$
square
$\hat{x}^2$
-
$\widehat{w_x^2}$
Variance Check
$\hat{t}_x$
$\hat{x}$

Figure 2.5: The Across-Channels Computation. The across-channels computation computes the centroid $(\hat{x}, \hat{t}_x)$ of the cloud. It receives the c preliminary coordinate pairs $(\hat{v}_i, \hat{t}_i)$ in vector form. Three centroids consisting of an inner product and a division take the vector of $\hat{v}_i$'s as weights. The coordinate vectors are the pulse time centroids $\hat{t}_i$, positions of the strips x_i, and squared strip positions x_i^2. These coordinates generate estimates $\hat{t}_x$, $\hat{x}$, and $\widehat{x^2}$ respectively. The cloud's estimated spatial variance, $\widehat{w_x^2} = \widehat{x^2} - (\hat{x})^2$, is compared to the standard range for the variance of a single-cloud collision in Variance Check. If the variance is inside of this range, the cloud's X dimension centroid $(\hat{x}, \hat{t}_x)$ is returneed. If not, the results are dropped.

The across-channels computation uses the preliminary coordinates $(\hat{v}_i, \hat{t}_i)$ generated by the single-channel computations to estimate the cloud's position centroid, $\hat{x}$, and time centroid, $\hat{t}_x$, described in Section 2.4.1 and Section 2.4.2 respectively. Computation of the cloud's spatial variance is used for detecting overlapping clouds and is detailed in Section 2.4.3. Figure 2.5 presents this part of the algorithm in block-diagram form.

2.4.1 Position Centroid Computation

Under the assumption that the cloud's charge deposition on the cross-strip anode follows a 2-D normal distribution, strips closer to the center of the cloud receive larger current pulses. The average pulse heights $\hat{v}_i$ are therefore a robust indicator of a strip's proximity to the center of the cloud. They are taken as weights on the corresponding strip's position x_i in the center of gravity computation that generates $\hat{x}$.

$$\hat{x} = \frac{\sum_{i=0}^{c} \hat{v}_i x_i}{\sum_{i=0}^{c} \hat{v}_i} \tag{2.9}$$

2.4.2 Time Centroid Computation

Each $\hat{t}_i$ estimates the time centroid of the pulse received by that channel. Under the twin assumptions of synchronized digitization on all channels and signal path uniformity on all channels, the time centroid of the pulse may be interpreted as the time centroid of the cloud. However, as depicted in Figure 2.2, a channel whose strip was on the edge of the cloud will receive a current pulse that is both lower in magnitude, decreasing SNR in a noisy system, and shorter in duration, reducing the number of useful samples for the time centroid algorithm to use. These have the combined effect of reducing the SNR and precision of measurements from these channels. Thus, the across-channels computation performs another center of gravity operation on $\hat{t}_i$ using the average pulse heights $\hat{v}_i$ as weights (Equation 2.10). Especially in the presence of noise, this method generates an impressive theoretical improvement in accuracy over a simple unweighted average of the time centroids as shown in Chapter 4.

$$\hat{t}_x = \frac{\sum_{i=0}^{c} \hat{v}_i \hat{t}_i}{\sum_{i=0}^{c} \hat{v}_i} \tag{2.10}$$

2.4.3 Variance Computation

A practical problem motivates the variance computation here: the MCP-PMT system will occasionally capture near-simultaneous photon arrivals. Should these photons be in close spatial proximity to each other, they will generate electron clouds that overlap in time and space. The across-channels computation relies on a cloud detection stage (described in Section 3.5.1) to group hits on multiple adjacent strips into “clouds” and present their preliminary coordinates as a set. Multiple clouds that are separated by at least a strip in both dimensions on the cross-strip anode pose no problem to this scheme, but simultaneous clouds that overlap in *either dimension* present as a single “cloud” in the dimension(s) of overlap. The cloud detection logic will then present all the preliminary coordinates from both clouds to the across-channels computation as a single set. Thus, a single spatial and temporal centroid will be generated for their overlap pattern. This has the effect of blending spatially proximate, simultaneous emissions from separate photons into a single (x, y, t) coordinate – clearly erroneous behavior that should be avoided.

As shown in Chapter 4, *temporally separated* but *spatially similar* cloud arrivals can cause the FADC electronics to transmit pulses from different clouds at the same time – tricking the cloud detection logic into thinking they belong to the same cloud – or alternatively drop one or two of the edge pulses from a single cloud. These situations, like the simultaneous arrival problem just described, result in the FADC crate transmitting a set of pulses that the Centroiding Arbiter's cloud detection logic cannot distinguish from a standard cloud. This leads to erroneous $(\hat{x}, \hat{t}_x)$ results.

To solve this problem, the across-channels computation drops clouds which are wider or narrower than the standard range for a single-cloud collision. This is implemented practically by estimating the spatial variance $\widehat{w_x^2}$ of the cloud, where w_x is the cloud's spatial standard deviation, or width, in the X dimension. The centroids $(\hat{x}, \hat{t}_x)$ are only returned if $\widehat{w_x^2}$ falls within a standard, parameterized range. The spatial variance is an ideal characteristic for this task as opposed to the standard deviation because of its simple implementation in hardware (it doesn't involve a square root – Section 3.5.2) and because it is more sensitive to small perturbations in cloud width, allowing the fine tuning of standard range parameters. The thresholds which define an acceptable value for the spatial variance will be determined experimentally once the vacuum chamber assembly has been completed. The standard "shorthand" formula for computing variance is used:

$$\widehat{w_x^2} = \widehat{x^2} - \hat{x}^2 = \frac{\sum_{i=0}^{c} \hat{v}_i x_i^2}{\sum_{i=0}^{c} \hat{v}_i} - \left(\frac{\sum_{i=0}^{c} \hat{v}_i x_i}{\sum_{i=0}^{c} \hat{v}_i} \right)^2 \tag{2.11}$$

2.5 Characterizing Estimator Accuracy and Precision

The Centroiding Algorithm ultimately generates, for each incident electron cloud, three estimates: $\hat{x}$, $\hat{t}_x$, and $\widehat{w_x^2}$. To quantify their performance, it is first useful to define the relevant properties of an estimator [28]. Each property is given for $\hat{x}$ and is applicable to the other two estimates as well. For an input cloud composed of S pulses $V_{x,i,t}$, $i \in [0, S-1]$, C is the matrix created by the S pulses. If the cloud has center position μ_x, the error of $\hat{x}$ is:

$$e(\hat{x})_C = \hat{x}(C) - \mu_x \tag{2.12}$$

Over N noisy observations of clouds with the same center position μ_x, where each cloud's pulse matrix is C_k, $k \in [0, N-1]$, the bias B of the estimator $\hat{x}$ is defined as:

$$B(\hat{x}) = \frac{1}{N}\sum_{k=0}^{N-1}(\hat{x}(C_k) - \mu_x) = \frac{1}{N}\sum_{k=0}^{N-1} e(\hat{x})_{C_k} \tag{2.13}$$

The bias B is clearly the simple average of the errors $e(\hat{x})_{C_k}$. It is therefore a measure of the accuracy of the estimator – how close the estimates are, on average, to the true parameter. It is therefore useful for determining systematic error in the system, as shown in Section 4.3. It does not account for the *spread* of the estimates around their average value, a characteristic here called precision.

The sample variance s^2 of the estimator $\hat{x}$ over the same set of observations is

$$\begin{aligned}
s^2(\hat{x}) &= \frac{1}{N-1}\sum_k \left(\hat{x}(C_k) - \frac{1}{N}\sum_l \hat{x}(C_l)\right)^2 && (2.14)\\
&= \frac{1}{N-1}\sum_k \left(\hat{x}(C_k)^2 - \frac{2}{N}\hat{x}(C_k)\sum_l \hat{x}(C_l) + \left(\frac{1}{N}\sum_l \hat{x}(C_l)\right)^2\right) && (2.15)\\
&= \frac{1}{N-1}\sum_k \hat{x}(C_k)^2 - \frac{2}{N(N-1)}\left(\sum_k \hat{x}(C_k)\right)^2 + \frac{N}{N-1}\left(\frac{1}{N}\sum_l \hat{x}(C_l)\right)^2 && (2.16)\\
&= \frac{1}{N-1}\left(\sum_k \hat{x}(C_k)^2 - \frac{1}{N}\left(\sum_k \hat{x}(C_k)\right)^2\right) && (2.17)
\end{aligned}$$

Here we have arrived at the familiar shorthand for the variance computation. It is clear that this value does not depend on the cloud's position μ_x, but is a measure of precision – the degree to which the estimator's results agree with each other.

The standard deviation s is the square root of the unbiased sample variance s^2:

$$s = \sqrt{s^2} = \sqrt{\frac{1}{N-1}\left(\sum_k \hat{x}(C_k)^2 - \frac{1}{N}\left(\sum_l \hat{x}(C_l)\right)^2\right)} \tag{2.18}$$

The advantage of computing the standard deviation is that it "tells the same story" as the variance – meaning it conveys the precision of the estimator – but in the same physical unit as the estimator. The downside is that it is a biased estimate of the population standard deviation, although the use of the *unbiased* sample variance (dividing by $N-1$ above) lessens the degree to which s is biased. It is therefore more appropriate to call it the *corrected* sample standard deviation.

Chapter 3

Implementation

The Data Processing FPGA is a massively parallel computational platform implemented inside the Xilinx Virtex-6 FPGA featured on the Xilinx ML605 evaluation board. This design implements the centroiding algorithm described in Chapter 2 to measure the incident cloud's centroid in one dimension, $(\hat{x}, \hat{t}_x)$. A view of the data-path and module hierarchy of the Data Processing FPGA is given in Figure 1.5. A massively parallel front end interface (Sections 3.2 & 3.3) to the 50 relevant channels of the FADC crate computes preliminary coordinates on the digitized current pulses. A Centroiding Arbiter (Sections 3.4 & 3.5) performs the dual tasks of recognizing cloud arrivals and computing their centroids. An Ethernet Medium Access Control (EMAC) provided by Xilinx and some associated control logic (Sections 3.6&3.7) packages the centroids into Ethernet frames and sends them to a computer via a Gigabit Ethernet stream. Figure 3.1 names the interfaces used between each stage of the design and also depicts clock-domain crossings.

After an introductory section describing the boards and physical components used (Section 3.1.1) and the fixed-point notation (Section 3.1.2), the remainder of this chapter walks through each interface and component, starting from the custom serial interface connected to each FADC Channel FPGA and terminating at the Gigabit Ethernet connection on the ML605.

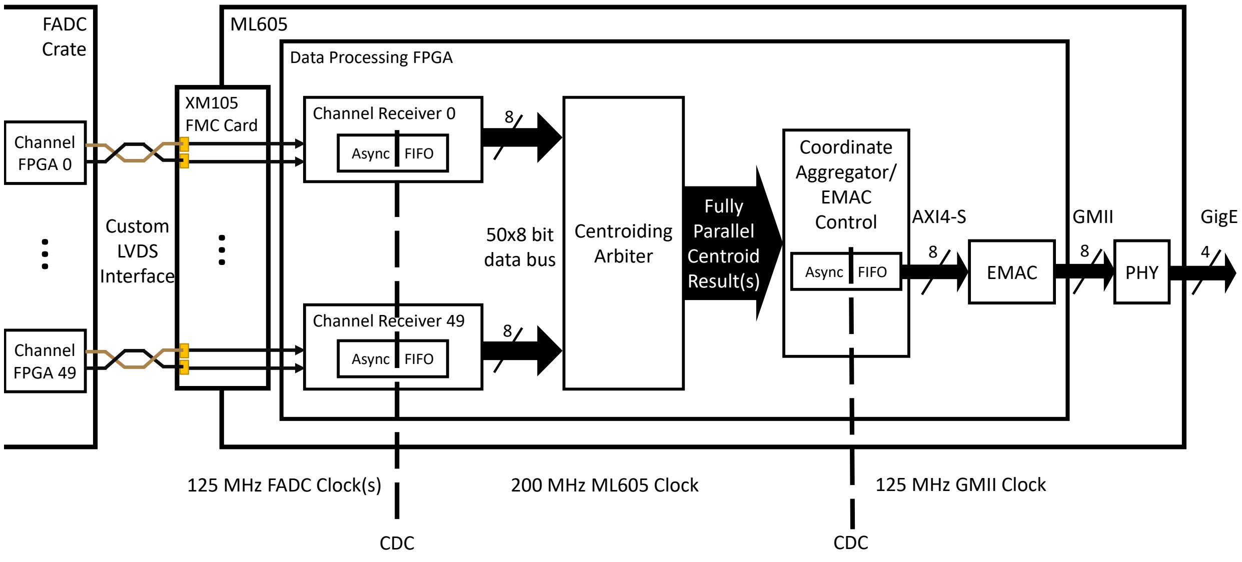

Figure 3.1: Data Processing FPGA Interfaces. An LVDS interface (Section 3.2.1) carrying a custom protocol (Section 3.2.2) is used to transmit data from the FADC Channel FPGAs to the pins of a Xilinx XM105 FPGA Mezzanine Card (FMC) [29]. The XM105 is mounted on an ML605 Evaluation Board [30] which provides a parallel interface to the Data Processing FPGA. Inside each Channel Receiver (Section 3.3), an Asynchronous FIFO provides a clock domain crossing (CDC) from the 125 MHz domain provided by the FADC clock(s) to the 200 MHz ML605 system clock domain. In this domain, the Centroiding Arbiter (Section 3.5) accesses the preliminary computations of the Channel Receivers via a 50×8-bit data bus (Section 3.4). The resultant centroids $(\hat{x}, \hat{t}_x)$ are presented to the Coordinate Aggregator (Section 3.6) via a fully parallel interface. The Coordinate Aggregator contains an asynchronous FIFO that crosses out of the 200 MHz ML605 domain and into the 125 MHz GMII (Gigabit Media Independent Interface) domain. As part of the Ethernet interface (Section 3.7), the EMAC Controller packages the data into Ethernet frames and sends them via an 8-bit AXI4-S [31] interface to the EMAC where they are finally encapsulated into the Ethernet protocol. The PHY [32] receives the Ethernet stream via GMII and produces the physically accessible Gigabit Ethernet link.

3.1 Introduction

3.1.1 Boards and Components

The work of this thesis is built around the Xilinx ML605 Evaluation Board [30]. It features a wide array of external interfaces and programming options as well as a sizable FPGA – the Virtex-6 XC6VLX240T-1FFG1156 [33] featuring 301,440 flip-flops, 768 DSP blocks for arithmetic operations, and 14,976 Kb of RAM. These ample programming resources make it ideal in the early stages of this project when design feasibility was still an open question. An additional benefit was the Buckley Lab's possession of two ML605 cards courtesy of Xilinx for student research purposes.

The ML605's FPGA Mezzanine Card (FMC) slot is essential for supporting the massive 50 channel interface to the FADC crate. The Xilinx XM105 "debug" card [29] is intended for use as a break-out for monitoring and testing signals inside the FPGA. However, it provides easy access to 152 general-purpose IO pins of the Virtex-6 when seated in the High Pin-Count (HPC) slot of the ML605. As discussed in Section 3.2, data transmitted from the 50 FADC Channel FPGAs is encoded as a differential signal on unshielded-twisted-pair (UTP) cables. Each Channel FPGA's data pair connects to two pins of the Data Processing FPGA, requiring 100 pins for the data signals alone. Additional pairs transmit a clock signal for the data – each FADC Board (10 channels) has a single clock, yielding 5 clocks total. The combined IO requirement of the data and clock signals is thus 110 pins. The XM105 card can accommodate all these signals with pins to spare for debugging purposes.

The Gigabit Ethernet link is accessed via the ML605's RJ-45 connector. A Marvell Alaska 88E1111 PHY [32] encodes the data from the EMAC's GMII (Gigabit Media Independent Interface) into the physical specification necessary to implement Gigabit Ethernet.

The FPGA is programmed via two methods, depending on the use case. A JTAG (Joint Test Action Group) connection directly to the FPGA's configuration logic enables fast programming for debugging and testing purposes. However, the downloaded bitstream is stored in volatile memory that is lost when the FPGA configuration sequence is ended. If power is lost, the device will boot into an undefined state that has no functionality. For long term deployment, the ML605 offers multiple non-volatile flash memories including a 128 Mb Platform Flash XL and a 32 MB Linear BPI Flash memory. A set of configuration switches control which of the three boot options the FPGA uses.

3.1.2 Fixed Point Arithmetic

The fixed-point notation used in the rest of the thesis is introduced here. An unsigned fixed point number $U(a.b)$ has a integer bits and b fractional bits, where a and b are integers. The range is thus $0 \leq U(a.b) \leq 2^a - 2^{-b}$ [34]. A signed 2's complement fixed point number $A(a.b)$ has 1 sign bit, a integer bits, and b fractional bits. It has a range $-2^a \leq A(a.b) \leq 2^a - 2^{-b}$. A multiplication of two unsigned fixed point numbers $U(a_1.b_1)$ and $(a_2.b_2)$ yields a result representable in $U((a_1 + a_2).(b_1 + b_2))$ bits. The multiplication of two signed fixed point numbers similarly adds the number of bits to the left and right of the radix point: $A(a_1.b_1) \cdot A(a_2.b_2) = A((a_1 + a_2 + 1).(b_1 + b_2))$. The additional 1 integer bit is a semantical result of the fact that a signed fixed point number $A(x.y)$ as defined here has $x + 1$ bits to the left of the radix point [34].

3.2 FADC Communication

A low-overhead, low-resource, massively parallel LVDS interface serves as the Data Processing FPGA's connection to the upstream imaging system and electronics, described in Section 3.2.1. Section 3.2.2 presents a lightweight serial protocol with a minimal but robust set of features that enables high throughput and flexibility for future feature additions.

3.2.1 LVDS Interface

The LVDS interface (Figure 3.2) is built around 55 twisted pairs carrying LVDS signals. Each of the 5 FADC boards generates 10 FADC Channel FPGA data signals and a 125 MHz clock signal. Both the data and the clock signals are driven by standards-compliant drivers onto $100\,\Omega$ UTP ribbon cable. Each differential signal connects to a pair of pins on the XM105 card. Ordinarily, a termination resistor would have to be shunted across each pair of pins. This is avoided by utilizing the Virtex-6's $100\,\Omega$ internal termination. The differential signals are made accessible to the internal FPGA fabric via the Virtex-6's built-in differential input buffers.

Each data signal is synchronous to its board's clock signal, but clocks between boards are mesochronous, i.e., they have the same frequency but unknown phases. Additional work described in Chapter 5 may be able to accomplish synchronization between all five boards, but for now the presence of 5 mesochronous clocks implies 5 separate clock domains inside the Virtex-6. The fabric of the Virtex-6 is subdivided into a set of "clock regions," areas of logic that, generally speaking, are associated with a single IO bank and can only be clocked by clock pins inside that IO bank. The user has the choice of connecting their input clocks

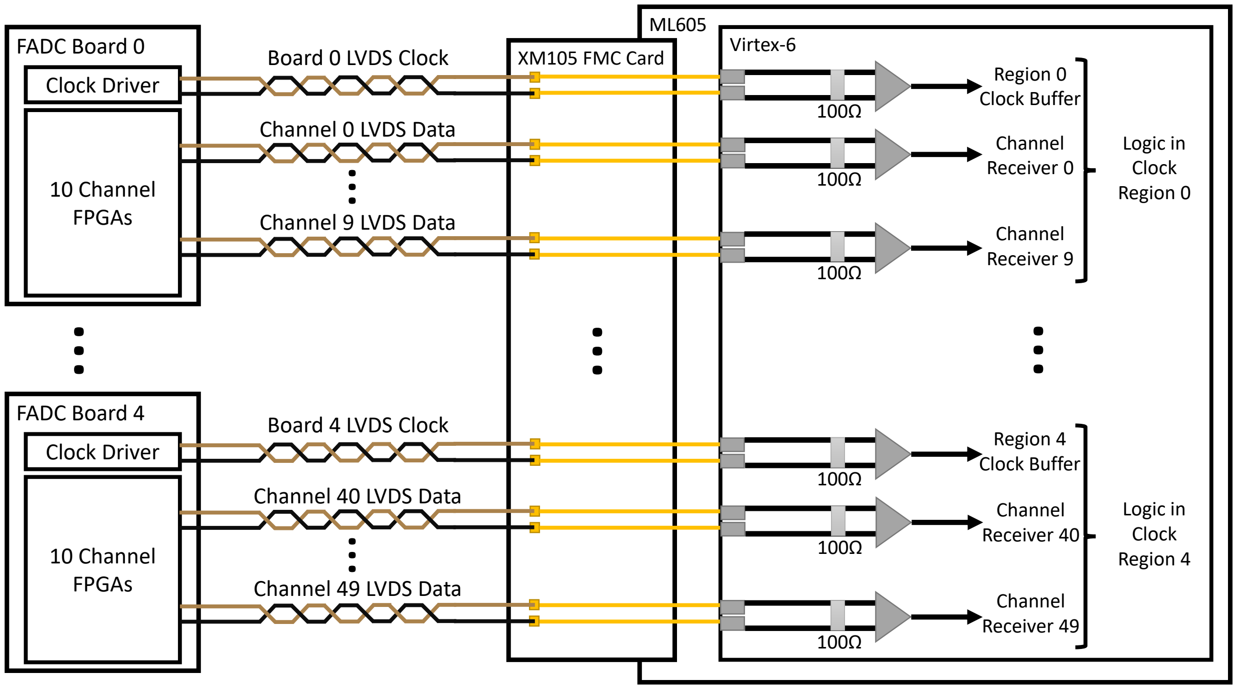

Figure 3.2: LVDS Interface. 10 LVDS data signals (corresponding to 10 strips of the cross-strip anode) and an LVDS clock are driven by each FADC board. The signals travel across 100 Ω unshielded twisted pair (UTP) cables that connect to the pins of the XM105 card. The signals reach the pins of the Virtex 6 and are terminated across internal 100 Ω resistors. Differential input buffers (grey triangles) present the clock signals to regional clock buffers and the data signals to the Channel Receiver modules. Each clock signal must, in general, reside inside the clock region of the corresponding data signals (for a detailed explanation see Section 3.2.1).

to global clock pins that can drive every clock region, single-region clocking pins (SRCC's) that drive a single region, or multi-region clocking pins (MRCC's) that can drive a single region and the two "adjacent" regions. The ML605 only exposes MRCC or SRCC pins to the FMC card, so global clocking is not an option. The clock signal and 10 data signals from each board must therefore be connected to either the same IO bank (if using an SRCC) or adjacent IO banks (if using the MRCC). Using a combination of MRCC and SRCC pins, the XM105 provides sufficient pin exposure to clock the 10 data signals from an FADC board with their synchronous clock. Figure 3.2 presents an idealized version of this arrangement where each FADC board's signals are assigned a unique region.

3.2.2 Serial Protocol

The FADC Channel FPGA sends messages containing 12 8-bit samples and a 16-bit timestamp to the Channel Receiver inside the Data Processing FPGA according to the serial protocol described in Figure 3.3. The protocol uses 8-bit words sent most significant bit (MSB) first on a 125 MHz clock. The idle state of the transmission is a framing word, `8Ch`. This word was chosen because it has the characteristic that when shifted (with bit recovery), it never equals itself. This means that the Channel Receiver can observe the deserialized version of the received data and know exactly which bit is the most significant and which is the least.

Once the Receiver has aligned itself to the deserialized data, it waits for the Channel FPGA to send the header word `AAh`. `AAh` (`10101010b`) has a Hamming distance of 3 from the framing word `8Ch` (`10001100b`) (it differs in three bit positions). A study using similar technology operating at a lower clock rate found LVDS links had a bit error ratio (BER) of less than 10^{-12} [35]. Even under the drastic exaggeration of a BER of 10^{-6}, the probability p_3 that a framing word suffers 3 induced errors is:

$$p_3 = \binom{8}{3}(10^{-6})^3(1-10^{-6})^5 \approx 10^{-18} \tag{3.1}$$

In other words, given the link speed's operation at 125 Mbps (15.625 MBps), it would take $t_3 = 10^{18}$ bytes/$(15.625 \times 10^6$ bytes/sec$) \approx 2000$ years of framing word transmission on average for one of them to suffer three errors. No consideration was made for *which three bits* will be in error. This effectively guarantees that noise will not "trick" the channel receiver into deserializing a nonexistent message.

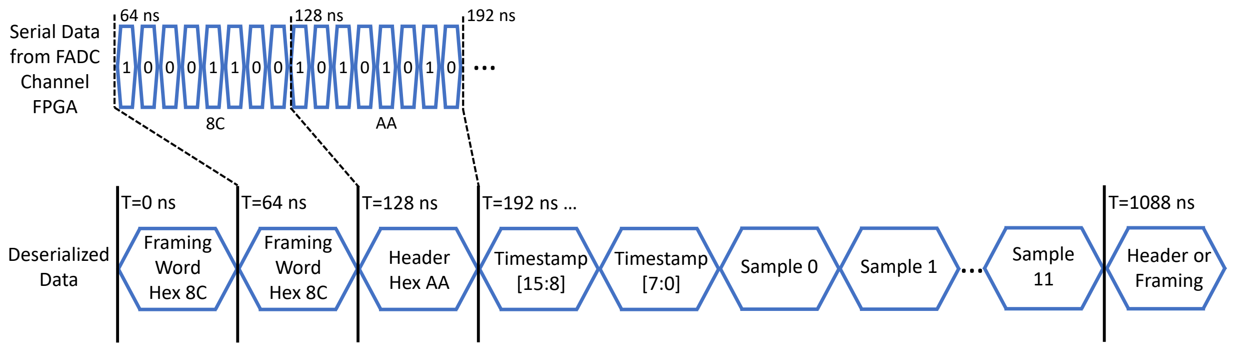

Figure 3.3: Serial Protocol. The FADC Channel FPGA communicates with the Data Processing FPGA by sending 8-bit words most significant bit (MSB) first. When not active, the Channel FPGA must repeatedly send the framing word `8Ch`. To initiate a message, the header word `AAh` is sent followed by a 16-bit timestamp corresponding to the time of the first sample, $t_{i,0}$. The twelve samples are sent with the earliest sample first. At the end of a message, the Channel FPGA can either end transmission by sending the framing word or initiate a new message by sending the header word.

The local dead time of the MCP-PMT system is exactly the amount of time it takes to send a message from the Channel FPGA to the Data Processing FPGA. That is, all of the steps in the "pipeline" that follow this section take less time than the LVDS message transmission, which is 15 bytes/$(15.625 \times 10^6$ bytes/sec$) = 960\,$ns long. Thus it is important to trim as many nanoseconds from the message transmission as possible. Two steps have been taken with this goal: first, the design has been modified from what was initially a two-byte header to the single word `AAh` that is currently in use. Second, the capability to "stream" was added, i.e. continue transmission by sending a header and additional message after the last sample. This avoids the 64 ns delay incurred from sending the framing word in between messages. The requirement to send a header instead of allowing "true streaming" – a timestamp immediately after the last sample – maintains the confidence that a message was initiated intentionally.

3.3 Channel Receiver

The ith Channel Receiver (Figure 3.4) is responsible for processing the differential serial data stream generated by the ith FADC Channel FPGA into the two preliminary coordinates of Section 2.3: the average pulse height $\hat{v}_i$ and the pulse time centroid $\hat{t}_i$. A Xilinx ISERDES block (Section 3.3.1) deserializes the serial data into parallel words which are written into an asynchronous FIFO by the control logic. On the read side of the FIFO, a Channel Computation (Section 3.3.2 and Section 3.3.3) computes the preliminary coordinates and presents them to the Centroiding Arbiter.

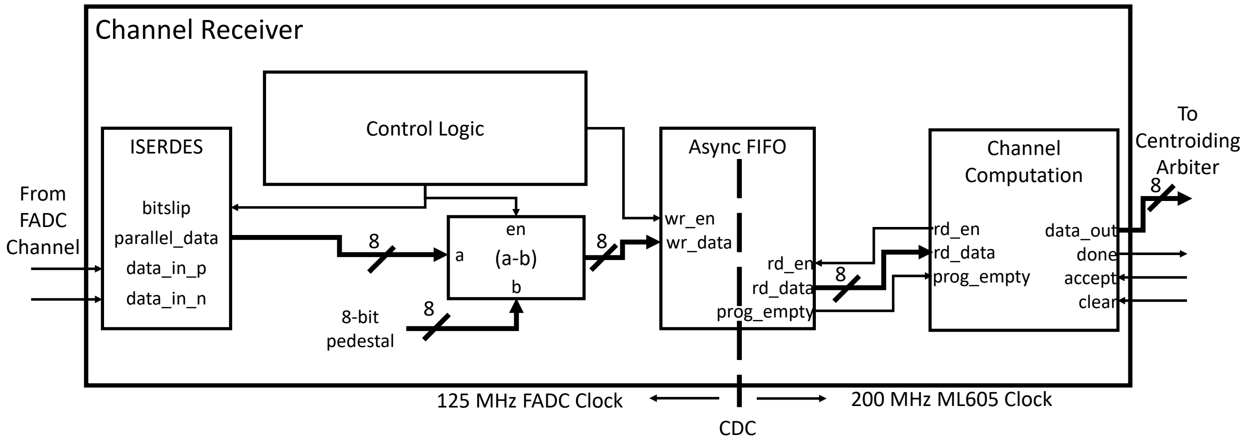

Figure 3.4: Channel Receiver. The serial data obeying the protocol from Section 3.2.2 enters the ISERDES differential `data_in_p` and `data_in_n` ports. Inside, it is shifted into a register with bit ordering set by the control logic using the `bitslip` signal. It emerges as an 8-bit deserialized version, `parallel_data`. If `parallel_data` is a sample, the control logic subtracts the 8-bit pedestal value associated with the FADC electronics so that a 0 volt signal corresponds to a digital value of 0. If `parallel_data` is one of the two timestamp bytes, it does not subtract the pedestal. It writes the 2-byte timestamp and 12 8-bit samples into an asynchronous FIFO. On the other side, in the 200 MHz ML605 clock domain, the Channel Computation (Section 3.3.2 and Section 3.3.3) performs the single channel computation on the received data.

3.3.1 Deserialization Logic

The Xilinx ISERDES block is the point-of-contact between the LVDS interface and the FPGA fabric. It is a "hardened" block that resides very close to the physical pin of the FPGA [36]. Differential input buffers (IBUFDS) at each pair of pins convert the differential data signals to single ended versions which can be used by the internal registers of the ISERDES block. The clock signal, also differential at the pins, drives a differential clock input buffer (IBUFGDS) that generates a single-ended version. An IO clock buffer (BUFIO) is necessary for this single ended clock to reach the "IO column," the region of the device where the ISERDES resides [37]. The ISERDES uses the BUFIO-driven 125 MHz clock to save the input data on each positive edge.

The fabric-facing side of the ISERDES block features an 8-bit `parallel_data` output and a `bitslip` input. Use of these signals requires the designer to present the ISERDES with an additional divided clock. This additional clock must have a frequency equal to the input clock's frequency divided by the width of each word – in this case 125 MHz/8 = 15.625 MHz. To generate this clock, a regional clock buffer (BUFR) is connected to the original IBUFGDS-buffered single-ended 125 MHz clock and parameterized to divide the input frequency by 8 [37]. The BUFR-driven 15.625 MHz clock is used by the control logic and ISERDES of the 10 Channel Receivers whose data is synchronous to the original clock as described in Section 3.2.1.

The operation of the control logic is shown in Figure 3.5. The 8-bit `parallel_data` is updated by the ISERDES on each positive edge of the divided clock. During the framing portion of the serial protocol (Section 3.2.2) the control logic observes this output and, if it doesn't match the framing word `8Ch`, asserts the `bitslip` signal for one divided clock cycle. The

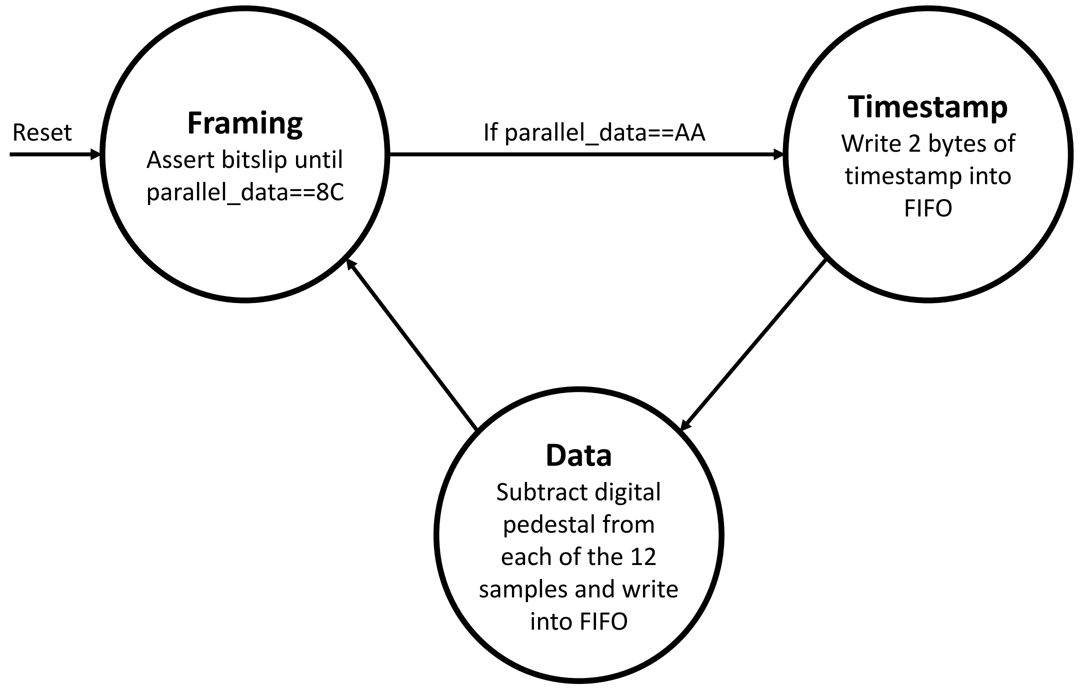
Reset
Framing
Assert bitslip until parallel_data==8C
If parallel_data==AA
Timestamp
Write 2 bytes of timestamp into FIFO
Data
Subtract digital pedestal from each of the 12 samples and write into FIFO

Figure 3.5: Channel Receiver Control Logic. The Channel Receiver (Section 3.3.1, Figure 3.4) features control logic that is responsible for moving the deserialized message from the ISERDES into an asynchronous FIFO. In the Framing state, the `bitslip` signal is asserted until the ISERDES `parallel_data` matches the framing word, `8Ch`. If `parallel_data` matches the header `AAh`, the control logic transitions to Timestamp. The two bytes of the timestamp are written sequentially into the FIFO. The logic then transitions to Data. For 12 cycles, the control logic subtracts the digital pedestal value associated with a 0 volt signal (`11h`) from `parallel_data` and writes the result into the FIFO. The control logic then reenters Framing until a new header arrives.

ISERDES then shifts the order of the bits of `parallel_data` by one to the left and updates the `parallel_data` output with this new order. Once the bits are properly aligned, the control logic waits for the header word `AAh` to appear on `parallel_data`.

When the header word appears, the control logic enters two states designed to efficiently move the message's contents into the asynchronous FIFO. The first two bytes of the message are the timestamp of the first sample, $t_{i,0}$, and are written (with one register stage to improve timing) directly into the FIFO. The next 12 bytes are the samples of the current pulse and have to be modified slightly before entering the FIFO. In the FADC electronics, a 0 volt signal is mapped to a digital “pedestal” value – `11h`, or decimal 17 – to allow representation of slightly negative signals. Retaining this pedestal value introduces systematic error in the centroiding computation because it essentially gives a nonzero weight to a 0 volt sample. The control logic thus subtracts the digital pedestal from each of the 12 samples before writing them into the FIFO.

3.3.2 Channel Computation: Sum and Inner Product

The Channel Computation module implements the single-channel computation described in Figure 2.4. A state machine (Figure 3.6) controls the interface with the asynchronous FIFO and the computation of the results $\hat{v}_i$ and $\hat{t}_i$. This section describes the portion of the logic responsible for computing $\text{sum}(v)$ and $v \cdot t$. The following section describes the divisions.

The FIFO’s `prog_empty` signal is high whenever the number of words in the FIFO is less than a parameterized threshold. By setting the threshold just below the number of words in a single FADC message (14), the flag will deassert (i.e. become 0) after a complete message has been written in. This action kicks the Channel Computation logic out of the Idle state and into Timestamp in which the two bytes of the $t_{i,0}$ timestamp are read out of the FIFO sequentially and saved into a flip-flop. The logic then transitions to Samples, which implements the computation of $\text{sum}(v)$ and $v \cdot t$.

Each sample leaving the asynchronous FIFO is stored in a pipeline register to improve timing, as the path from FIFO to computational logic was sometimes close to 5 ns. The sample leaves the pipeline stage and enters the FPGA implementation of the sum and inner product blocks, shown in Figure 3.7. The sum is simply an adder that adds the latest sample to a register

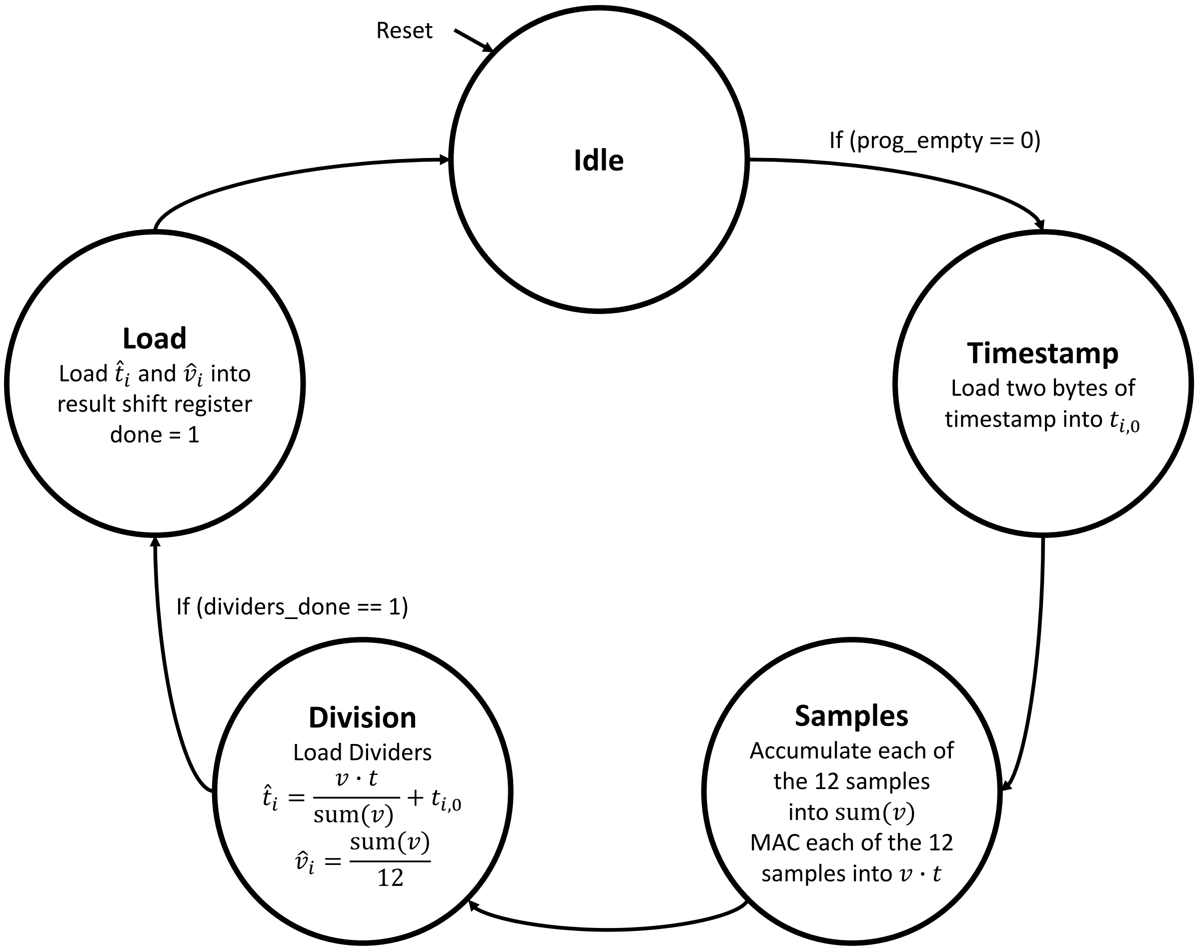
Reset
Idle
If (prog_empty == 0)
Timestamp
Load two bytes of timestamp into $t_{i,0}$
Samples
Accumulate each of the 12 samples into sum(v)
MAC each of the 12 samples into $v \cdot t$
Division
Load Dividers
$\hat{t}_i = \frac{v \cdot t}{\text{sum}(v)} + t_{i,0}$
$\hat{v}_i = \frac{\text{sum}(v)}{12}$
If (dividers_done == 1)
Load
Load $\hat{t}_i$ and $\hat{v}_i$ into result shift register
done = 1

Figure 3.6: Channel Computation Logic. When the number of words in the Asynchronous FIFO reaches a complete message, `prog_empty` deasserts. This causes the logic to transition into Timestamp, where the two bytes of the timestamp of the first sample ($t_{i,0}$) are loaded into a flip-flop. The logic then enters Samples, where the 12 samples are read out of the FIFO. They enter an accumulator (yielding sum(v)) and a multiply-accumulator (MAC) with the time offsets of each sample (yielding $v \cdot t$). See Figure 3.7 for implementation. The logic then initiates the two divisions of Figure 2.4 which, with the addition of $t_{i,0}$ to $v \cdot t/\text{sum}(v)$, generate the results $\hat{v}_i$ and $\hat{t}_i$. See Section 3.3.3 for implementation. In Load, the results are loaded into a shift register and the `done` signal is asserted.

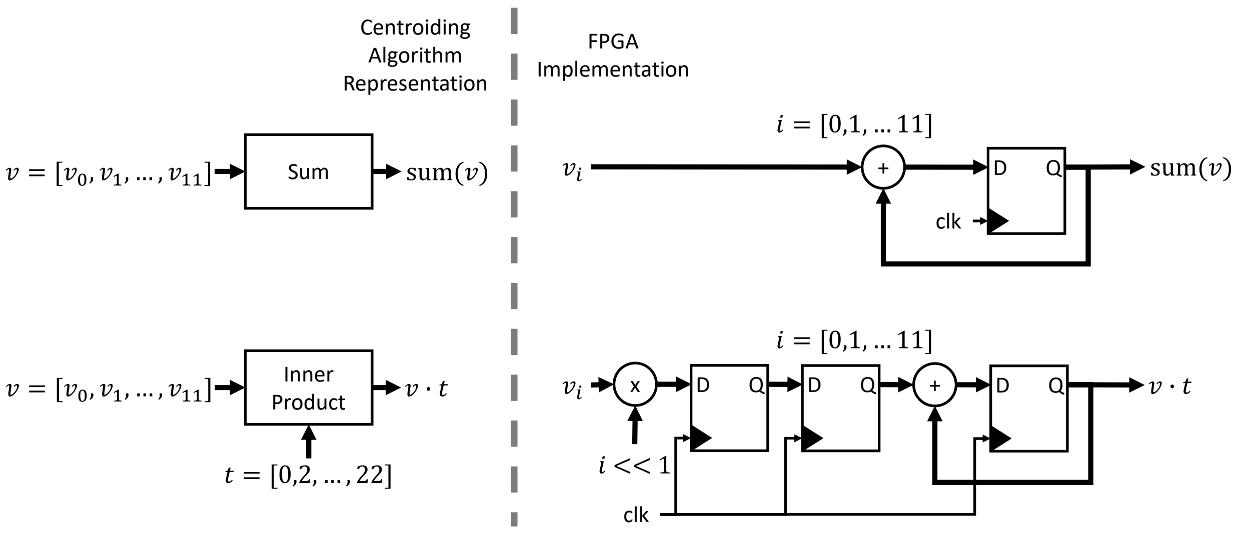

Figure 3.7: Sum and Inner Product Implementation. A sum over 12 elements, for example $v = [v_0, v_1, ..., v_{11}]$ is implemented as an accumulator which adds its present value (initially zero) to the current element for 12 `clk` cycles. An inner product of those same 12 elements with a weight vector $t = [0, 2, ..., 22]$ (represented by left shifting i by 1, represented by the operator `<<`) is implemented as a 3-stage multiply and accumulator (MAC). A multiplier produces the product of the current weight and the current element which is saved in a register on `clk`. At the next `clk` edge, the product enters a second pipeline stage to improve timing and a newly computed product enters the first pipeline stage. On the third cycle, the original product will be "accumulated" into the signal $v \cdot t$ and the second product will be stored in the second pipeline stage. The MAC process thus takes 14 cycles because of the 2-cycle latency between input and accumulation.

holding the sum of the previous samples. This repeated process of add-store-add is referred to below as an accumulator. The inner product is similar to an accumulator, except there is a 2-cycle multiplier stage before the adder – creating a multiply-accumulator, or MAC. Each 8-bit sample is multiplied by a 5-bit time index in the range $[0, 2, ..., 22]$ with the result saved in a register.

In the Xilinx tools, as in every major FPGA vendor's toolchain, the multiplier stage can be inferred into DSP blocks simply by writing $*$ in Verilog. The tool detects the 2 pipeline stages after the multiplication and leverages the implied extra 2 clocks (10 ns) to optimally place and route the DSP blocks. Because the operands of this multiplication are so small (8-bit sample and 5-bit time index) and the Virtex-6 fabric is very robust at 200 MHz, the tool sometimes chose to bypass the DSP block and use a LUT-based implementation.

Because the sum(v) and $v \cdot t$ results are generated by repeated additions/multiplications, the integer bit length of the output register has to be manually assigned. The maximum value of sum(v) is $\sum_{i=0}^{11} 255 = 3060$ which takes 12 integer bits to represent, so the sum(v) result register is 12 bits. The maximum value of $v \cdot t$ is $\sum_{i=0}^{11} 2i \cdot 255 = 33660$ which takes 16 integer bits to represent, so the $v \cdot t$ result register is 16 bits.

3.3.3 Channel Computation: Division

The Divider module (Figure 3.8) implements iterative non-restoring integer division [38] and features fully parameterized word width for both the divisor and dividend, as well as the option to internally add fractional bits to the dividend before computation. This enables the use of the same Verilog module with different parameters to compute every quotient necessary for the design. As shown in Figure 2.4, the Channel Computation module must compute two divisions: the sample sum sum(v) by 12 and the inner product $v \cdot t$ by sum(v). These divisions generate, respectively, $\hat{v}_i$ (the average pulse height) and the offset to $t_{i,0}$ used for $\hat{t}_i$ (the pulse time centroid).

To more precisely determine each desired quantity, the Divider module introduces fractional bits to the numerator, creating a fixed point value. The implementation of a fixed-point division is a non-trivial operation in any digital system, but especially so in an FPGA [38].

Unlike the multiplication operator $*$, simply writing a "/" in Verilog will generate a vendor-specific, massive, and (at the 200 MHz ML605 clock speed) unroutable implementation of a division algorithm in combinational logic.

While the overview in [38] considers the feasibility of a pipelined implementation that could compute a quotient each clock cycle, the 960 ns LVDS message length means that the channel computation can take up to $960/5 = 192$ clocks, of which the sum and pipelined inner product use 13. This leaves 179 clocks for the division, far more than necessary for the iterative approach presented below. An iterative solution consumes less resources and is thus preferable over a pipelined or fully-parallel approach – the division module needs to be instantiated twice per channel for each of the 50 channels. Thus any simplification of the Divider module (and the Channel Receiver in general) will have a multiplied improvement on device utilization and timing performance.

The non-restoring division algorithm was chosen because of the simplicity of implementation in a hardware description language like Verilog. Each cycle contains a single comparison which decides the next quotient digit $Q[i]$ and the arithmetic operation to perform on the partial remainder and divisor (either addition or subtraction). This is an improvement on the number of comparisons and per-cycle results that have to be computed in the standard long division, also called restoring division – so named because it "restores" the value of the partial remainder instead of allowing it to go negative. This simplification approximately halves the amount of logic necessary [38], important because of the parallel instantiation of 100 Dividers.

The primary practical drawback of this approach compared to restoring division is that the digits of the generated quotient Q are either -1 or $+1$. Because $Q[i] \in \{-1, 1\}$, a conversion process is necessary to perform the map $Q[i] \in \{-1, 1\} \rightarrow \text{Quotient}[i] \in \{0, 1\}$,

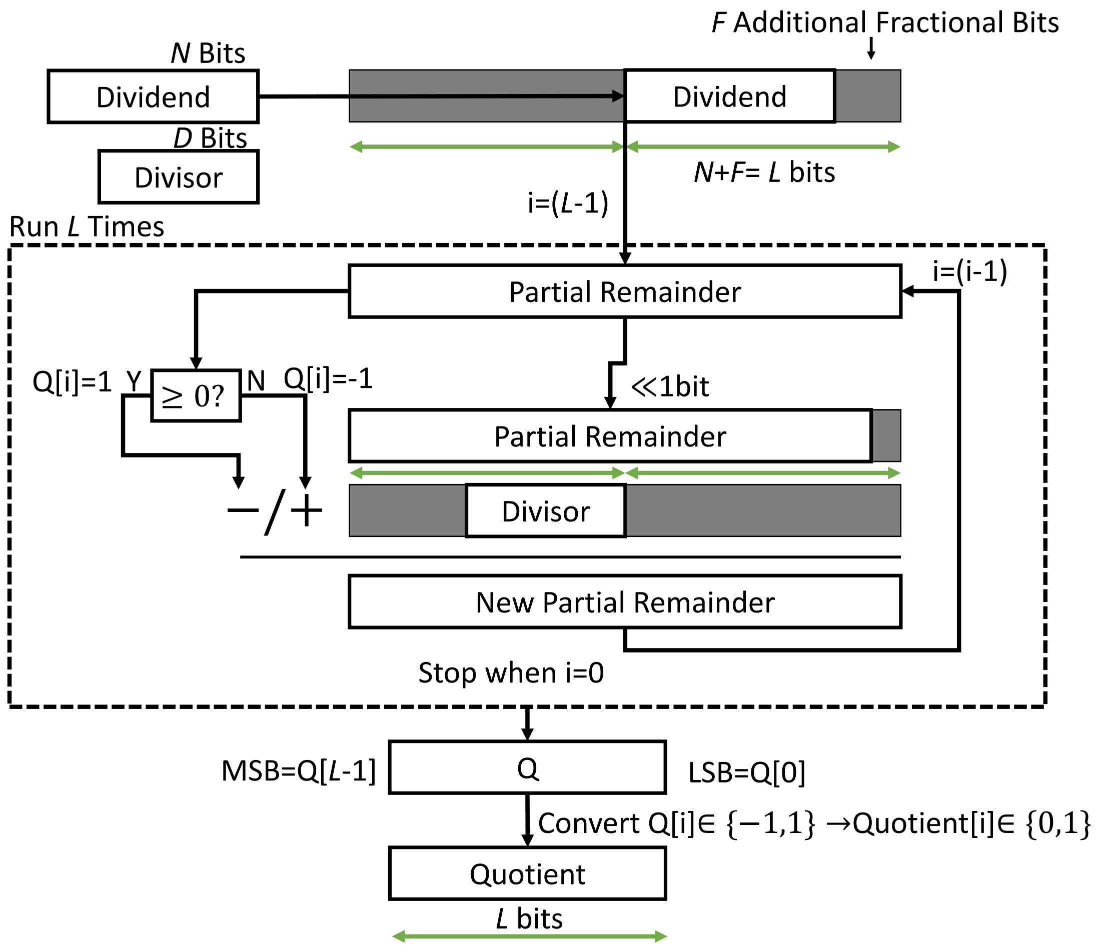

Figure 3.8: Division Implementation. The Divider module computes Dividend/Divisor where Dividend is N bits and Divisor is D bits. A parameter inserts F additional fractional bits (can be 0). The internal register size is $2L = 2(N + F)$ bits. The result register Q is L bits. On the first cycle, Dividend is shifted by F bits and loaded into the partial remainder register and Divisor is shifted by L bits and loaded into a register. The cycle index i is initialized to $(L - 1)$. Each cycle, the comparison checks the sign of the partial remainder. If nonnegative, the ith bit of Q is set to 1 and the chosen operation is subtraction. Otherwise, the ith bit of Q is set to -1 (represented by 0) and the chosen operation is addition. The chosen operation is then performed on the partial remainder shifted left 1 bit (represented by the operator <<) and the shifted Divisor. i is decremented each cycle and the process ends when $i = 0$. A final conversion maps $Q[i] \in \{-1, 1\} \rightarrow \text{Quotient}[i] \in \{0, 1\}$.

where Quotient is Q in 2's complement representation. Because the -1's in Q are represented by a binary 0, Quotient can be expressed as

$$\text{Quotient} = Q - \bar{Q} \tag{3.2}$$

where $\bar{Q}$ indicates the bitwise inversion, or one's complement. As in [38], the definition of 2's complement inversion $-A = \bar{A} + 1$ allows for a simplification, where << is the left shift operator:

$$\text{Quotient} = Q + \bar{\bar{Q}} + 1 = 2Q + 1 = (Q\texttt{<<}1) + 1 \tag{3.3}$$

Because $Q[0] \in \{-1, 1\}$, Q is always odd. After converting to standard 2's complement representation, the final remainder should be used to fill in the even results, a process not mentioned in [38]. In the restoring division, which has nonnegative remainders, the remainder can be discarded after the last iteration because true integer division will always round *down* the remainder. In nonrestoring division, the remainder can be negative throughout the computation and may be negative after the last iteration, implying that the true result is something *less* than Q. The final quotient must therefore be conditionally "rounded down" by subtracting 1 from the 2's complement Quotient in the event of a negative remainder. This can be easily incorporated into the map above by noting that if the remainder is negative, its sign bit R_s is 1 (in 2's complement representation):

$$\text{Quotient} = (Q\texttt{<<}1) + \bar{R}_s \tag{3.4}$$

where the addition of $\bar{R}_s$ means 1 will only be added if R_s is a 0, implying the final remainder was positive.

Because the Divider module performs integer division on fixed point inputs with understood fractional bits, the output quotient will have understood fractional bits, the number of which must be determined. The division of two fixed point numbers X ($U(a_1.b_1)$) by Y ($U(a_2.b_2)$) will produce a fixed point quotient Q ($U(a_3.b_3)$). We wish to find a_3, the number of integer bits, and b_3, the number of fractional bits.

In this implementation of non-restoring division, Q has the same number of bits as X (once the parameterized shift of Figure 3.8 is included), so one of the two independent equations necessary is $a_3 + b_3 = a_1 + b_1$. The other relation necessary to determine a_3 and b_3 is derived from expressing X/Y in terms of the raw integer magnitudes of their bits, $|X|$ and $|Y|$:

$$\frac{X}{Y} = \frac{|X| \cdot 2^{-b_1}}{|Y| \cdot 2^{-b_2}} = \frac{|X|}{|Y|} \cdot 2^{b_2 - b_1} = \frac{|X|}{|Y|} \cdot 2^{-(b_1 - b_2)} \tag{3.5}$$

The result thus has $b_3 = (b_1 - b_2)$ fractional bits – an intuitive consequence when division is viewed as a kind of inverse operation to multiplication's $b_3 = (b_1 + b_2)$ fractional bits. By plugging b_3's new expression into the equivalence $a_3 + b_3 = a_1 + b_1$, we find $a_3 = a_1 + b_2$. The fact that the quotient has b_2 more integer bits than the dividend is also intuitive: if the divisor is less than 1, the division essentially becomes a multiplication. The product's upper bound is proportional to how close the divisor can get to zero and thus how many fractional bits the divisor has.

As mentioned above, the arguments of the two Divider modules in each Channel Computation block are all integers: $\hat{v}_i$ is the result of dividing a $U(12.0)$ sum(v) by the 4-bit integer 12, and the quotient used for $\hat{t}_i$ is the result of the $U(16.0)$ inner product $v \cdot t$ divided by $U(12.0)$ sum(v). While this implies the quotients would have 12 and 16 integer bits respectively, knowing the physical meaning of the results allows for restrictions beyond those implied by the math. The average pulse height has the range of a sample: $\hat{v}_i \in [0, (255 - \text{pedestal})]$, so it can be represented by 8 integer bits. The pulse time centroid is a value in nanoseconds, $\hat{t}_i \in [0, 22]$, so it only needs 5 integer bits.

Each quotient will also, by the relation $b_3 = (b_1 - b_2) = 0 - 0$, have zero fractional bits if the dividends aren't shifted left. Both Dividers are thus parameterized to add 8 digits of fractional precision (F in Figure 3.8). Shifting each dividend left by $F = 8$ bits allow a temporal binning Δt of:

$$\Delta t = \frac{1000\,\text{ps}}{2^8\text{bins}} = 3.906\,\text{ps/bin} \tag{3.6}$$

The precision improvement in $\hat{v}_i$ is also important because this value will be used as a weight in each of the three computations in the Centroiding Arbiter. A less-precise weight will give channels either more or less of a say in the final result than they actually deserve. Adding 8 fractional bits to each dividend does have the drawback of increasing the number of cycles in the division algorithm and thus adding latency. However, it only increases the number of clocks required to 20 and 24 – well below the 179 clock dead-time limit!

To compute the $\hat{t}_i$ pulse time centroid, the $U(5.8)$ quotient $(v \cdot t)/\text{sum}(v)$ relative pulse time centroid must be added to the first sample's $U(16.0)$ system timestamp, $t_{i,0}$, as shown in Figure 2.4. Although relatively innocuous, the format of this simple addition has major implications. If $t_{i,0}$ is in the last 22 nanoseconds of the representable range $[0, 65535]$, the addition could overflow and require 17 integer bits. Allowing a carry bit causes major problems for the across-channels computation (described in Section 3.5.2), so the carry bit is dropped and an overflow is simply wrapped back around to the bottom side of the range $[0, 65535]$.

When the two Divider modules have finished their computation, they raise a `divider_done` signal. Because the $\hat{v}_i$ quotient takes four fewer clocks than the $(v \cdot t)/\mathrm{sum}(v)$ quotient, the `divider_done` signals from each Divider are AND-ed together to create `dividers_done`. As shown in Figure 3.6, when `dividers_done` goes high, both quotients are ready and the logic enters the Load state where the two results $\hat{t}_i$ ($U(16.8)$) and $\hat{v}_i$ ($U(8.8)$) are loaded into a 40-bit shift register. The logic indicates completion by raising the `done` signal on the interface with the Centroiding Arbiter, described in Section 3.4. The logic then enters the Idle state and waits for a new message signaled by the deassertion of the FIFO's `prog_empty`.

3.4 Channel-Arbiter Interface

The interface between the Channel Receivers and the Centroiding Arbiter is a 50-lane data bus. Each lane connects a single Channel Receiver with the Centroiding Arbiter, so there is therefore no bus sharing. Rather, data transmission on each lane is facilitated via an 8-bit data signal (`data`) and three control signals: `done`, `clear`, and `enable`. This arrangement is given in Figure 3.9.

A channel's assertion of `done` indicates to the Centroiding Arbiter that it has completed the computation of the 40-bit preliminary result $(\hat{t}_i, \hat{v}_i)$. After the Centroiding Arbiter decides whether or not the channel's pulse was part of a cloud (Section 3.5.1), it asserts `clear` to cause the channel to deassert `done` and avoid double counting. If the pulse was part of a cloud, the `enable` signal is used to shift the 40-bit result across the 8-bit `data` signal. The enable signal must therefore be asserted for 4 clocks to shift out the remaining 32 bits in the shift register.

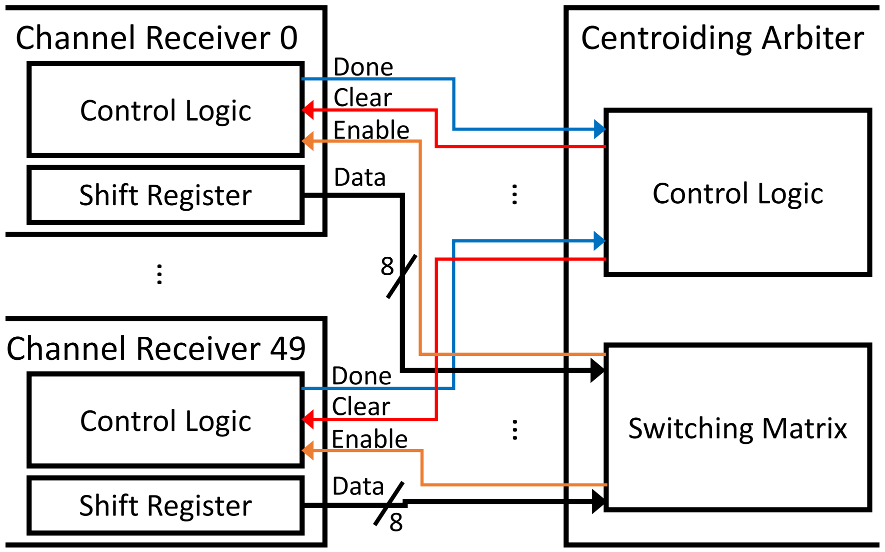

Figure 3.9: Channel-Arbiter Interface. Each of the 50 Channel Receivers communicates with the Centroiding Arbiter via its own dedicated lane. Each lane contains an 8-bit `data` signal and three control signals: `done`, `clear`, and `enable`. The Channel Receivers assert `done` when they finish computing the preliminary coordinates as described in Section 3.3.3. The control logic in the Centroiding Arbiter groups channels with high `done` signals into clouds for processing. It then asserts `clear` on those channels which causes their `done` signals to deassert, preventing double counting. When the Centroiding Computation block (not shown) is ready to process a channel's `data`, it uses the switching matrix to assert the channel's `enable` and shift the result one byte at a time out of the channel's shift register.

The decision to use an 8-bit datapath was motivated by design feasibility concerns. First, it is desirable to pick a bit width that is an integer factor of the 40-bit result length. This allows the result to be shifted out with no wasted bandwidth in an integer number of clocks. However, if the entire result was presented in parallel, or even if a 20-bit path was used, the tools have an enormous difficulty placing and routing the design in a way that meets timing. This is primarily because each of the (potentially) multiple Centroiding Computation blocks of Figure 3.10 must be able to read from *each* of the 50 `data` signals.

To route each channel to each Centroiding Computation block, a very congested set of multiplexers and decoders is instantiated that determine which channel the Centroiding Computations are currently interfacing with – the switching matrix of Figure 3.9. Widening each channel's `data` bus by 1 bit therefore increases the number of bits that the switching matrix must successfully route to *each* Centroiding Computation block by 50. The 8-bit width was therefore chosen as a conservative and tool-friendly alternative to a fully-parallel option. Its bandwidth, at 1.6 Gbps per lane, is still very fast and enables the Centroiding Computation blocks to quickly read the 5-byte $(\hat{t}_i, \hat{v}_i)$ result from each of the relevant channels.

3.5 Centroiding Arbiter

The Centroiding Arbiter is responsible for identifying cloud arrival events (arbitration) and computing their centroid $(\hat{x}, \hat{t}_x)$ (centroiding). This requires a well-designed mix of three distinct varieties of digital design: decision-tree style control logic for cloud recognition (Section 3.5.1), a computational block to implement the across-channels computation (Section 3.5.2), and a structural, routing intensive solution to pass data to the computational blocks (Section 3.5.3). It is therefore the largest and most complicated component inside the Data Processing FPGA. Figure 3.10 depicts the interconnections between the submodules.

The fundamental parameter of the Centroiding Arbiter is the number of Centroiding Computation blocks to instantiate. In Figure 3.10 and throughout the rest of this chapter, the number is assumed to be 2 for both visual and verbal ease of explanation. However, instantiating more Centroiding Computation blocks in parallel would enable the processing of multiple spatially and temporally separated-but-proximate clouds, increasing photon efficiency. If the number is too low, multiple clouds separated by at least a single strip and around 100 ns each would potentially be dropped due to the inability of a single Centroiding Computation block to keep up with the rate of input clouds. However, instantiating just a single additional Centroiding Computation block has massive implications on design feasibility due to the increased routing and resource consumption. An optimal number will therefore be determined later, when the MCP-PMT system has been completed and characterized.

3.5.1 Arbiter Control Logic

The Arbiter control logic detects a cloud's arrival and assigns the computation of its centroid to one of the Centroiding Computation blocks. Figure 3.11 depicts the underlying state machine. As described in Section 3.4, the assertion of any of the bits in `channel_done` indicates that a Channel Receiver has finished processing a pulse and has the preliminary coordinates $(\hat{t}_i, \hat{v}_i)$ ready. In Verilog, the reduction OR operator on a vector `signal`, denoted `|signal`, returns a 1 if any of the bits in `signal` are 1 and a 0 otherwise, and is therefore used to test `channel_done`. If `|channel_done` goes high, the logic knows at least one Channel Receiver has finished computing a pulse's preliminary coordinates.

Fundamentally, a cloud generates pulses on a contiguous set of strips, causing multiple consecutive bits in `channel_done` to assert at approximately the same time. As shown in Figure 2.2, pulses that receive less of the charge cloud may not begin to rise until significantly later, delaying their transmission and causing a staggered completion of $(\hat{t}_i, \hat{v}_i)$ coordinates for the pulses within a single cloud. Therefore, once `|channel_done` asserts, the logic waits a parameterizable number of clocks (`ACCUM` in Figure 3.11) to "accumulate" finished results into `channel_done`. It then saves `channel_done`.

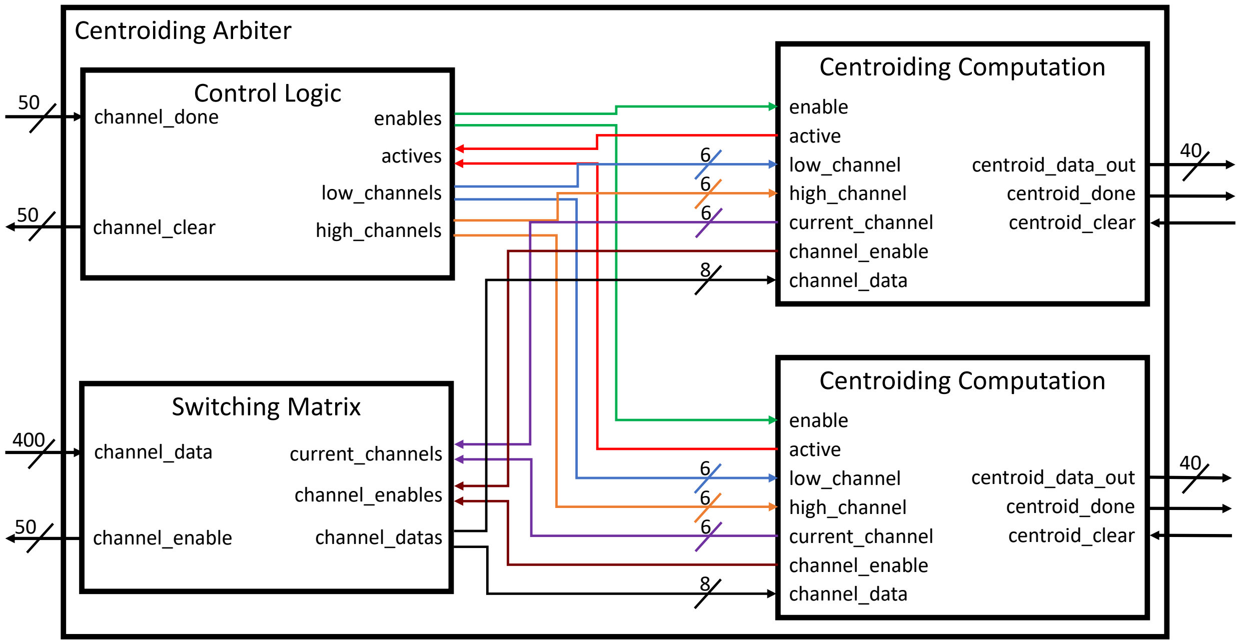

Figure 3.10: Centroiding Arbiter. The interconnections between submodules of the Centroiding Arbiter (Section 3.5) are given. The 50-bit `channel_done` and `channel_clear` vectors provide a means of 2-way communication between the cloud-identifying control logic and the 50 Channel Receivers. When the control logic has identified a cloud, it assigns its centroid computation to the first Centroiding Computation block which is not `active` by asserting the corresponding `enable` and loading the cloud edge channels (determined by Algorithm 1) onto the corresponding `low_channel` and `high_channel` signals. The Centroiding Computation block reads the preliminary results from each channel iteratively by loading `current_channel` and asserting its `channel_enable` signal. The switching matrix aggregates the desired reading behavior from all of the Centroiding Computation blocks (2 depicted) into the 50-bit `channel_enable` vector. Any channel which is enabled streams its 8-bit data onto the 400-bit `channel_data` vector. The switching matrix again uses `current_channel` to connect each Centroiding Computation block with its desired stream. When the Centroiding Computation block has finished a result, it asserts `centroid_done` and presents the 40-bit result on the `centroid_data_out` signal.

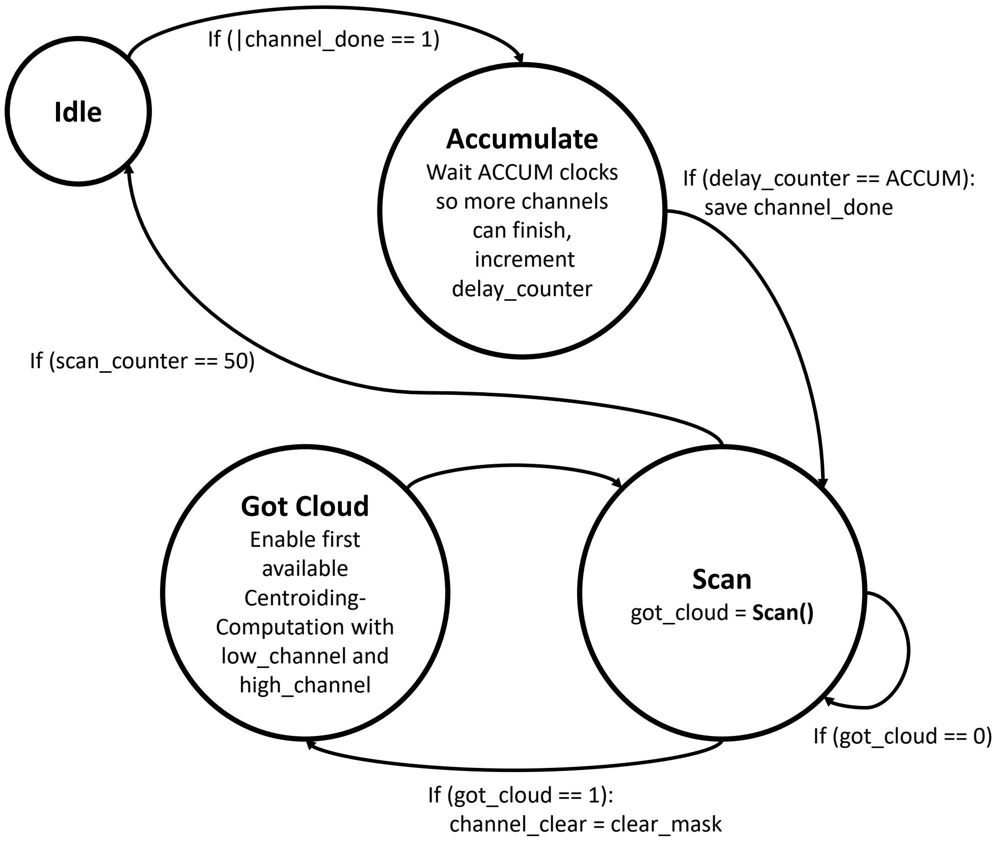
Idle
If (|channel_done == 1)
Accumulate
Wait ACCUM clocks so more channels can finish, increment delay_counter
If (delay_counter == ACCUM): save channel_done
If (scan_counter == 50)
Got Cloud
Enable first available Centroiding-Computation with low_channel and high_channel
Scan
got_cloud = Scan()
If (got_cloud == 0)
If (got_cloud == 1): channel_clear = clear_mask

Figure 3.11: Centroiding Arbiter Logic. Explained extensively in Section 3.5.1, the Arbiter control logic consists of 4 states. In Idle, the logic waits for any Channel Receiver to finish processing a pulse, indicated by the reduction OR of `channel_done`: |`channel_done`. In Accumulate, the logic pauses for `ACCUM` clocks to gather all the temporally staggered pulses of a cloud. In Scan, the logic iterates across the saved `channel_done` vector looking for clouds. It executes the **Scan** algorithm, Algorithm 1, every cycle to check the next bit of `channel_done` and "build" clouds by saving a `low_channel` and `high_channel` for each cloud. When the end of a cloud is reached, the logic transitions to the Got Cloud state to assign the centroid computation to the first available Centroiding Computation block. It then returns to Scan and repeats the process until the last channel in `channel_done` is reached, at which point it returns to Idle and waits for new clouds to arrive.

Algorithm 1 Scan

```
1:  got_cloud ← 0
2:  channel_done ← channel_done >> 1
3:  scan_counter ← scan_counter + 1
4:  if channel_done[0] = 1 then                          ▷ current channel is done
5:      hit ← 1
6:      clear_mask[scan_counter] ← 1                     ▷ clear the done flag
7:      if last_hit = 0 then                             ▷ low edge of new cloud
8:          low_channel ← scan_counter
9:      end if
10: else
11:     hit ← 0
12:     if last_hit = 1 then                             ▷ just finished a cloud
13:         if scan_counter − low_channel ≥ MIN then     ▷ It was acceptable size
14:             got_cloud ← 1
15:             high_channel ← scan_counter − 1
16:         end if
17:     end if
18: end if
19: last_hit ← hit
20: return got_cloud
```

The Arbiter control logic then determines the edges (`low_channel` and `high_channel` in Figure 3.10) of the (possibly multiple) cloud(s) that have been saved in `channel_done`. To do this, it scans across the saved `channel_done` vector one bit at a time. This is accomplished by right shifting by one bit each clock, `channel_done>>1`, and analyzing the LSB, `channel_done[0]`. The process performed each time a new bit of `channel_done` is analyzed is described in **Scan**, Algorithm 1. If the current LSB of `channel_done` is a 1, the `hit` signal of **Scan** is set to 1. To determine if this is a cloud's lower edge, the previous value of `hit`, `last_hit`, is checked. If `last_hit` is a 0, this channel corresponds to the cloud's lower edge. The value of `scan_counter` (which increments each time `channel_done` is shifted) is then saved into `low_channel`.

After the lower edge of a cloud, and as the interior channels (if any) are scanned, `hit` stays high. When the logic reaches the upper edge of a cloud (indicated by `hit` $= 0$ and `last_hit` $= 1$ in **Scan**) the cloud has been fully traversed. However, it is not appropriate to accept the cloud unconditionally because, as shown in Chapter 4, erroneous transmission of only a small subset of a cloud's channels can occur. The width of the cloud, (`scan_counter` $-$ `low_channel`), is therefore tested against a parameterized hard minimum, `MIN`. If the width of the cloud is greater than or equal to `MIN`, a valid cloud has been scanned. The `high_channel` of the cloud is assigned `scan_counter` $- 1$, and the logic transitions to the Got Cloud state.

As the logic transitions to the Got Cloud state, the `clear_mask` of **Scan** is used. As the logic scans across a cloud, it loads `clear_mask[scan_counter]` with a 1 for each channel in the cloud. At the end of the cloud, `channel_clear` is assigned the `clear_mask`, which causes the `clear` signals of each channel involved in the cloud to assert. This causes each channel's `done` signal to deassert, as described in Section 3.4. When the logic eventually returns to Idle, the `done` signals of all of the "old" results will therefore be deasserted and any `done` signals that remain high correspond to new results – meaning they were finished while the logic was scanning the "old" results.

The Got Cloud state is responsible for assigning the centroid computation of the cloud defined by `low_channel` and `high_channel` to the first available Centroiding Computation block. It does this by iterating across `actives` – the vector composed of the `active` signal from each Centroiding Computation block – using a counter, `computation_counter`. If `actives[computation_counter]` $= 0$, the control logic asserts the corresponding enable `enables[computation_counter]`. It also loads the 6-bit `low_channel` and `high_channel` values into the appropriate element of the arrays `low_channels` and `high_channels`, respectively. The appropriate element is the, like in the `enables` vector, given by `computation_counter`.

These values define the range over which the Centroiding Computation block should perform its computation.

The logic then returns to the Scan state and repeats the process until it has scanned each of the 50 channels. It transitions back to Idle and waits for the reduction OR of `channel_done` to assert once more.

3.5.2 Centroiding Computation

The Centroiding Computation block implements the across-channels computation of Section 2.4 to compute the cloud's centroid $(\hat{x}, \hat{t}_x)$ and transmit it only if the cloud's estimated spatial variance $\widehat{w_x^2}$ falls within a standard range. The logic controlling this process is described in Figure 3.12. The overall functional flow of this block is remarkably similar to the Channel Computation block (Sections 3.3.2 & 3.3.3) – iteratively load the input data set, performing an accumulation and MAC as they arrive, then compute the desired results with a set of Divider blocks. The additional complexities of this block are: the more complicated process required to load each channel's preliminary results, the requirement to handle time-wrapping, the existence of three parallel MAC and Divider blocks, and the additional state to compute the spatial variance $\widehat{w_x^2}$ and compare it to the acceptable range. These complexities are explained below.

Load Result

The switching matrix's difficult task of routing each channel's data to the input of each Centroiding Computation block (Section 3.5.3) results in a delay between when the Centroiding

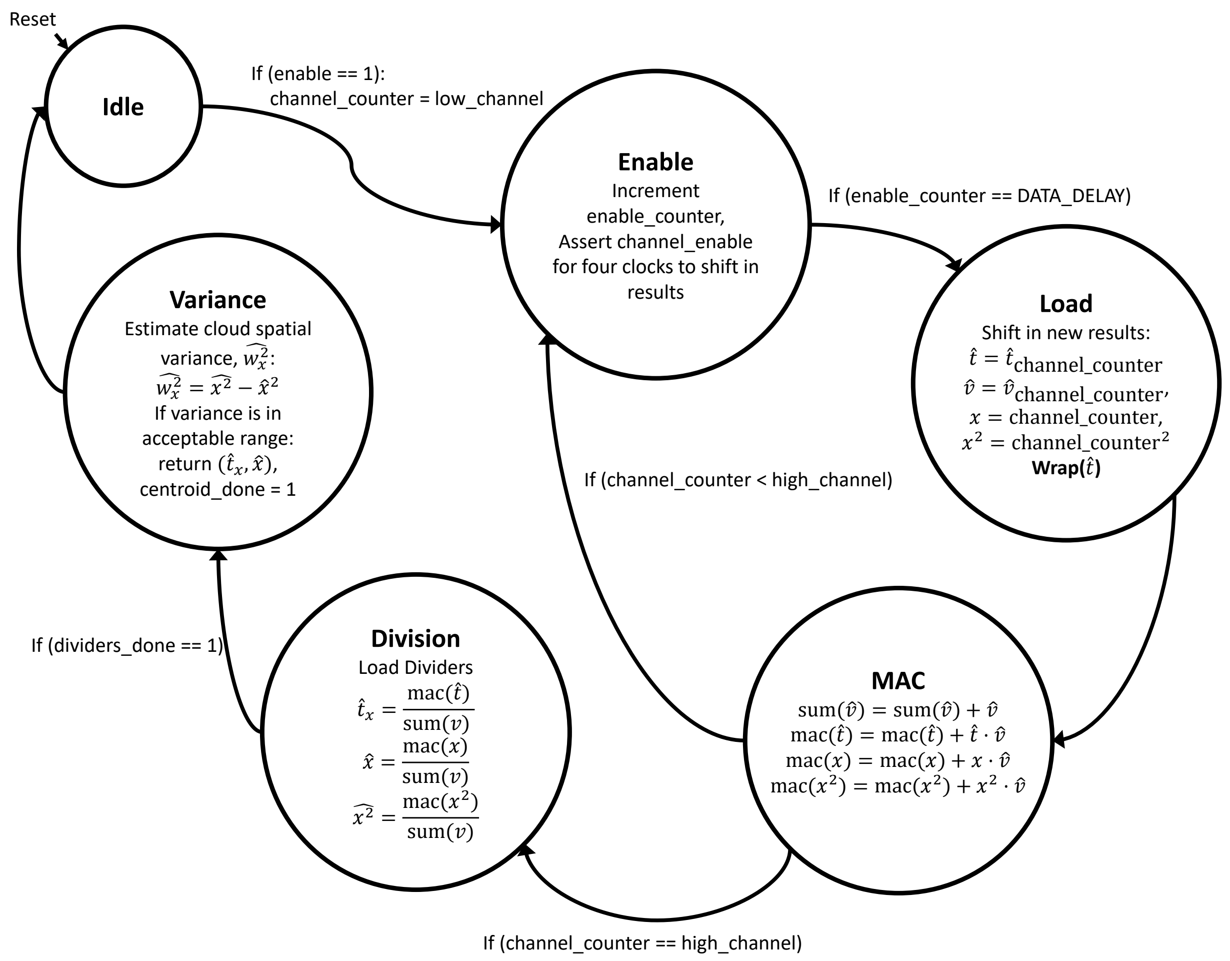
Reset
Idle
If (enable == 1):
channel_counter = low_channel
Enable
Increment enable_counter,
Assert channel_enable for four clocks to shift in results
If (enable_counter == DATA_DELAY)
Load
Shift in new results:
$\hat{t} = \hat{t}_{\text{channel\_counter}}$
$\hat{v} = \hat{v}_{\text{channel\_counter}}$,
x = channel_counter,
x^2 = channel_counter^2
Wrap($\hat{t}$)
If (channel_counter < high_channel)
MAC
$\text{sum}(\hat{v}) = \text{sum}(\hat{v}) + \hat{v}$
$\text{mac}(\hat{t}) = \text{mac}(\hat{t}) + \hat{t} \cdot \hat{v}$
$\text{mac}(x) = \text{mac}(x) + x \cdot \hat{v}$
$\text{mac}(x^2) = \text{mac}(x^2) + x^2 \cdot \hat{v}$
If (channel_counter == high_channel)
Division
Load Dividers
$\hat{t}_x = \frac{\text{mac}(\hat{t})}{\text{sum}(v)}$
$\hat{x} = \frac{\text{mac}(x)}{\text{sum}(v)}$
$\widehat{x^2} = \frac{\text{mac}(x^2)}{\text{sum}(v)}$
If (dividers_done == 1)
Variance
Estimate cloud spatial variance, $\widehat{w_x^2}$:
$\widehat{w_x^2} = \widehat{x^2} - \hat{x}^2$
If variance is in acceptable range:
return $(\hat{t}_x, \hat{x})$,
centroid_done = 1

Figure 3.12: Centroiding Computation Logic. The Centroiding Computation logic is responsible for computing a cloud's centroid $(\hat{x}, \hat{t}_x)$ and transmitting it only if the cloud's spatial variance $\widehat{w_x^2}$ falls within an acceptable range. The process is initiated by the Centroiding Arbiter's assertion of `enable` and loading of the cloud edge channels into `low_channel` and `high_channel`. In the Enable state, the logic asserts `channel_enable` which kicks off a read process through the switching matrix to fetch the $(\hat{t}_i, \hat{v}_i)$ result from the channel specified by `channel_counter`. In the Load state, the logic shifts in the 5-byte result and loads $\hat{t}, \hat{v}, x,$ and x^2. It corrects $\hat{t}$ to account for time-wrapping by executing **Unwrap**, Algorithm 2. The four mathematical operations in the MAC state are then performed. The Enable-Load-MAC cycle is repeated until the final channel specified by `high_channel` has been completed. The results $\hat{t}_x, \hat{x}, \widehat{x^2}$ are then computed by dividing the corresponding MAC result by the sum of the weights, sum(v). The cloud's centroid $(\hat{t}_x, \hat{x})$ is returned if the estimated spatial variance $\widehat{w_x^2} = \widehat{x^2} - \hat{x}^2$ is inside the standard range for a single cloud.

Computation block asks for a channel's result and when it arrives. When the Centroiding Arbiter control logic asserts `enable` and loads the cloud edge channels `low_channel` and `high_channel`, the Centroiding Computation immediately asserts `channel_enable` and loads `low_channel` into `current_channel` (signals shown in Figure 3.10). The asserted `channel_enable` propagates through the switching matrix and to the channel specified by `current_channel`, causing a shift in the shift register and a new byte of the result $(\hat{t}_{\text{current\_channel}}, \hat{v}_{\text{current\_channel}})$ propagating back through the switching matrix and into the Centroiding Arbiter's `channel_data` port.

However, the time delay associated with this long path and complicated switching far exceed a single 5 ns clock cycle. The process must be broken up into stages by adding pipeline registers for the `channel_enable` on the way out and `channel_data` on the return path. The number of registers required varies with the design width – for the 10 X channel, 10 Y channel test design (Chapter 4), the design passed timing with a 7-cycle delay from the assertion of `channel_enable` to the arrival of the next byte of `channel_data`. The parameter `DATA_DELAY` in Figure 3.12 is thus set to 7 cycles for the test setup. For the first 4 of those cycles, `channel_enable` needs to be asserted to shift out the 4 remaining bytes of the shift register. The total delay from `channel_enable` is first asserted to when the *last* byte of the channel's result arrives on `channel_data` is thus $7 + 4 = 11$ cycles.

Unwrap Times

It is at this point that $(\hat{t}_{\text{current\_channel}}, \hat{v}_{\text{current\_channel}})$ will have been shifted in and the logic can perform the math in the MAC state; however an additional step is required to account for the troublesome wrapping of the pulse time centroid $\hat{t}_{\text{current\_channel}}$. To understand the conditions that motivate this process, consider Table 3.1 which shows a set of 16-bit FADC timestamps $t_{i,0}$, relative pulse time centroids $(v_i \cdot t)/\text{sum}(v_i)$, and pulse time centroid estimates $\hat{t}_i = t_{i,0} + (v_i \cdot t)/\text{sum}(v_i)$ for a hypothetical 3-channel cloud. Recall that $\hat{t}_i$ is in $U(16.8)$ format and can therefore only represent $\hat{t}_i \in [0.0, 65535.996]$.

Table 3.1: Time-Wrapping Pulses.

Channel Number i	$t_{i,0}$	$v_i \cdot t/\text{sum}(v_i)$	$\hat{t}_i$
0 (`low_channel`)	65526	9.5	65535.5
1	65524	12.0	0.0
2 (`high_channel`)	65528	7	65535

The three pulses in Table 3.1 all had center times that were proximate to the last representable time. Because the Centroiding Computation essentially uses a weighted average of the $\hat{t}_i$ results to compute $\hat{t}_x$, this situation – if left uncorrected – will result in a tremendously inaccurate result: (weights set to 1 for simplicity):

$$\hat{t}_x = \frac{65535.5 + 0.0 + 65535}{3} = 43690.17 \tag{3.7}$$

Intuitively, the correct result can be obtained by replacing 0.0 with 65536.0 in the average:

$$\hat{t}_x = \frac{65535.5 + 65536.0 + 65535}{3} = 65535.5 \tag{3.8}$$

While the correction above is obvious when viewing the entire set of $\hat{t}_i$ results, the requirement to *iteratively* MAC these values – with no knowledge of the next channel's $\hat{t}_i$ result – motivates the process described by **Unwrap**, Algorithm 2. The algorithm is motivated by the fact that problematic time-wrapping cases occur when $\hat{t}_i$ values exist in both the first and fourth "quadrants" of the allowed range – $[0, 16384)$ and $[49152, 65536)$. Checking whether or not the $\hat{t}_i$ value is in either quadrant is simple in hardware – low quadrant times have 0's for the top two bits, and high quadrant times have 1's. Because no two pulses generated by the same MCP-PMT system could actually have center times in the first and fourth quadrant – implying they had a difference in center times exceeding 32 768 ns – the presence of $\hat{t}_i$ values in both the first and the fourth quadrants indicates that a wrapping event like that of Table 3.1 has occurred. The algorithm therefore maintains a notion of the previous quadrants and iterates over the $\hat{t}_i$ results of each of the c channels ($i \in [0, c-1]$), making corrections as necessary:

Algorithm 2 Unwrap

```
 1: got_low_quadrant ← 0
 2: got_high_quadrant ← 0
 3: for $i \in [0, c-1]$ do
 4:     if $\hat{t}_i < 16384$ then                               ▷ $\hat{t}_i$ is in low quadrant
 5:         if got_high_quadrant = 0 then
 6:             got_low_quadrant ← 1
 7:         else                                            ▷ previously had high quadrants
 8:             $\hat{t}_i \leftarrow \hat{t}_i + 65536$                          ▷ correct by wrapping up
 9:         end if
10:     else if $\hat{t}_i \geq 49152$ then                           ▷ $\hat{t}_i$ is in high quadrant
11:         if got_low_quadrant = 0 then
12:             got_high_quadrant ← 1
13:         else                                             ▷ previously had low quadrants
14:             $\hat{t}_i \leftarrow \hat{t}_i - 65536$                        ▷ correct by wrapping down
15:         end if
16:     end if
17: end for
```

The implementation of Algorithm 2 in hardware is simple and only requires the use of two single bit flags `got_low_quadrant` and `got_high_quadrant`. The subtraction or addition of 65536 causes $\hat{t}_i$ to lie outside of $[0, 65536)$. This motivates the transition to a signed representation for the result of the addition or subtraction. If only a single sign bit is added, the result can represent $[-65536, 65536)$. This means that the addition of 65536 would overflow. For this reason, an additional magnitude bit is also added, resulting in a signed fixed point number in $A(17.8)$ format. Recall that a number $A(17.8)$ has an implied sign bit and is therefore 26 bits, capable of representing $[-131072, 131072)$.

Although a number in $A(17.8)$ format is capable of representing times both larger than and smaller than the original range of $[0, 65536)$, for a given cloud it is only necessary to extend the range in one direction. This is because if a low quadrant $\hat{t}_i$ is scanned first, any subsequent high quadrant time centroids will be mapped to negative values. On the other hand, if a high quadrant $\hat{t}_i$ is scanned first, any subsequent low quadrant time centroids will be mapped to values greater than 65536. This implies that, with some clever manipulation, only a single additional bit could be used. This would result in a very marginal reduction in operand width (and thus less difficult routing) in the MAC and Division states. However, it ultimately comes at the cost of added complexity and less comprehensibility in the decision-tree logic of **Unwrap**. Therefore, the result of the addition or subtraction (or simply the $\hat{t}_i$ value itself, if no modifications are made) is saved in a signed fixed point $A(17.8)$ register `time_estimate_extended`.

MAC

With the time value corrected, the control logic of Figure 3.12 enters the MAC state. Here, as in the MAC state of the Channel Computation module (Section 3.3.2), an accumulator and (in this case) several MAC blocks are updated with the newest channel's results. The accumulator $\mathrm{sum}(\hat{v})$ maintains a running sum of the weights $\hat{v}_i$. The weight is used in the three MAC's on the "coordinates" which were saved in the Load state: `time_estimate_extended` (signified by $\hat{t}$ for brevity in Figure 3.12), x, and x^2. The MAC's are computed identically to the block diagram of Figure 3.7 – a 2-stage multiplier and a single-stage accumulator. Unlike in the Channel Computation, the operands of the three MAC's are wide and the Xilinx tools therefore used a complicated, multi-stage DSP-based implementation for the multipliers.

The widths of the computed values sum($\hat{v}$), mac($\hat{t}$), mac(x), and mac(x^2) are derived here. Each width depends on the input widths: the corrected time value `time_estimate_extended` is $A(17.8)$. The channel number x requires, due to 50 channels, 6 bits to represent. The squared channel number x^2 requires 12 bits. The weights $\hat{v}_i$ are $U(8.8)$. Table 3.2 shows the result format, computed using Section 3.1.2 and the fact that a cloud could, in theory, have up to 50 channels. The corresponding upper-bound of 50 repeated additions in the accumulator and MAC blocks leads to the addition of 6 integer bits to represent each result.

Table 3.2: Sum and MAC Result Format.

Function	Output Format
sum($\hat{v}$)	$U(14.8)$
mac($\hat{t}$)	$A(30.16)$
mac(x)	$U(20.8)$
mac(x^2)	$U(26.8)$

Division

When the final channel – specified by `high_channel` – has been processed in the MAC state, the logic transitions to the Division state. The three coordinate estimates ($\hat{t}_x, \hat{x}, \widehat{x^2}$) are computed by dividing the underlying MAC of each coordinate by the sum of the weights, exactly as in the Channel Computation block (Section 3.3.3). The Divider block (Figure 3.8 and Section 3.3.3) used for each division is parameterized with the widths of the dividend N, divisor D, and the number of fractional bits F to be inserted into the dividend. In the case of the $\hat{t}_x$ division, no additional fractional bits are needed because the quotient will have $16-8=8$ fractional bits. For $\hat{x}$ and $\widehat{x^2}$, however, the quotient would have $8-8=0$ fractional bits without the addition of extra fractional bits. For continuity with the temporal result, F is set to 8 for these two Dividers, giving each spatial coordinate 8 bits of fractional precision.

The increased operand widths (47 bits for mac($\hat{t}$)) in the Divider modules and the congested routing surrounding the Centroiding Computation block(s) meant that the Xilinx tools occasionally struggled to place and route the Dividers. The $\hat{t}_x$ Divider was especially difficult due to the presence of a $47 \cdot 2 = 94$ bit addition/subtraction that has to be performed each cycle (see Figure 3.8). However, it was discovered that the tool was *conditionally performing* the addition or subtraction based on the sign of the partial remainder – meaning the partial remainder's sign bit had to tell an "add/subtract" block which operation to perform. By forcing the tool to physically place both the addition and subtraction blocks and use the sign bit to *select the appropriate result* of either the addition or the subtraction, timing success was achieved on every subsequent build.

Variance

When all of the dividers have raised their `divider_done` signal, the logic enters the Variance state. As described in Section 2.4.3, the variance is computed by $\widehat{w_x^2} = \widehat{x^2} - \hat{x}^2$. This simply requires a 2-stage multiplier of $\hat{x} \cdot \hat{x}$ and a subtraction. The logic therefore is structurally identical to a MAC block, except it uses a subtraction instead of an addition. The fixed point format of each operand must be considered to perform the subtraction correctly. $\widehat{x^2}$ is derived from a weighted average of the squared channel number and thus has a range of $\widehat{x^2} \in [0^2, 49^2] = [0, 2401]$, representable in 12 integer bits. With the addition of the 8 fractional bits in the Divider, the bottom 20 bits of the quotient mac(x^2)/sum($\hat{v}$) are needed for $\widehat{x^2}$. Following a similar line of reasoning, $\hat{x}$ is simply a weighted average of the channel number and thus requires 6 integer bits to represent. With the addition of the 8 fractional bits in the Divider, the bottom 14 bits of the quotient $\text{mac}(x)/\text{sum}(\hat{v})$ are needed for $\hat{x}$.

The variance is therefore a subtraction of two fixed point numbers $\widehat{x^2} = U(12.8)$ and $\hat{x}^2 = U(6.8) \cdot U(6.8) = U(12.16)$. Because the fractional part of $\hat{x}^2$ has an extra 8 bits, it must be right-shifted by 8 to align it with $\widehat{x^2}$. The variance $\widehat{w_x^2}$ is thus a $U(12.8)$ number, and is compared to a set of thresholds, `low_variance` and `high_variance`, that will be set experimentally once the MCP-PMT has been completed and characterized. If `low_variance` $\leq \widehat{w_x^2} \leq$ `high_variance`, the cloud is considered acceptable.

Output Format

The variance successfully passing the threshold check causes the logic to assert `centroid_done` and load the result $(\hat{t}_x, \hat{x})$ into `centroid_data_out`. As the top two bits of $\hat{t}_x$ (the sign bit and additional magnitude bit added earlier) are only for mathematical bookkeeping, they are dropped and a $U(16.8)$ number is recovered. Due to the very helpful properties of 2's complement representation, if the quotient $\hat{t}_x$ was slightly negative, it becomes a large $U(16.8)$ number. If it was slightly above 65536, it becomes a small $U(16.8)$ number. This is, finally, the desired wrapping behavior.

Although the position centroid $\hat{x}$ only necessitates 6 integer bits, it is zero-extended to 8 integer bits. This yields a final result $(\hat{t}_x, \hat{x})$ that is 40 bits long, or 5 bytes. This is ideal because the Ethernet Interface is 1 byte wide, resulting in an even 5 data words per centroid result. It is also helpful because it maintains the byte placement of the result such that the bottom two bytes are the position centroid and the top three are the time centroid, greatly simplifying the task of parsing the generated Ethernet stream on a computer. The combined width of $\hat{t}_x$ and $\hat{x}$ is therefore $U(16.8) + U(8.8) = 40$ bits. With the result loaded and `centroid_done` asserted, the logic transitions back into Idle and waits for the Centroiding Arbiter control logic to enable a new round of computations.

3.5.3 Switching Matrix

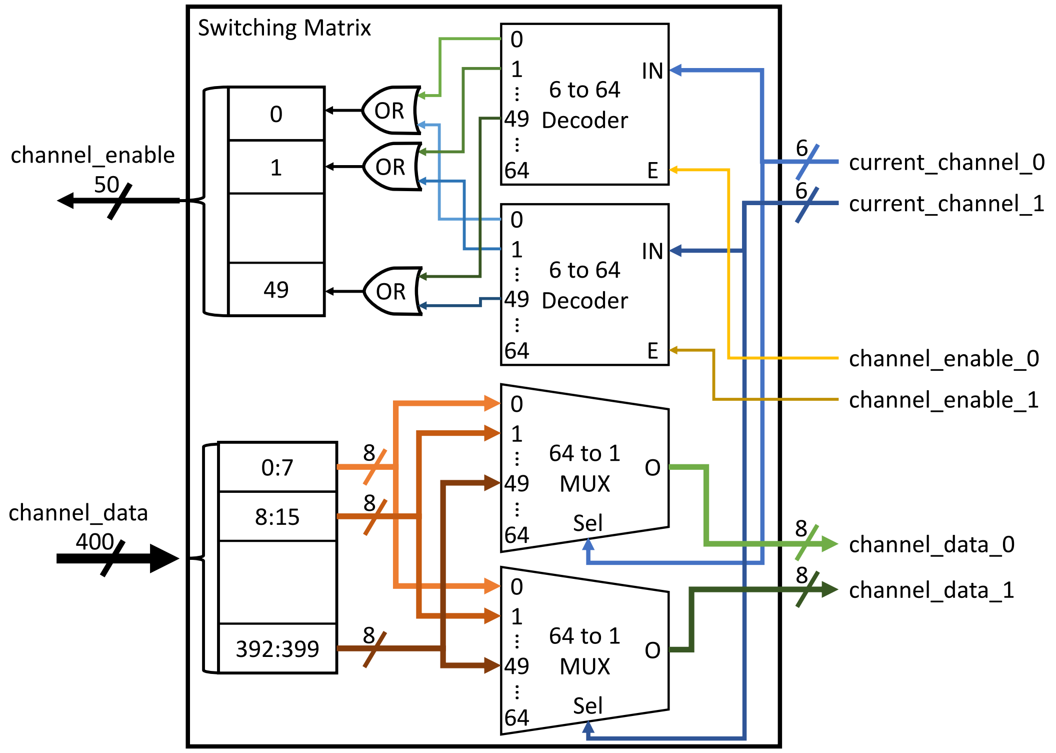

Figure 3.13: Switching Matrix. The switching matrix routes the read requests of each Centroiding Computation block into a single 50-bit `channel_enable` vector and simultaneously routes the desired channel's data to the appropriate Centroiding Computation block. An array of 6-to-64 decoders map the Centroiding Computation's 6-bit `current_channel` index into a one-hot vector. For example, if `current_channel_0` is the integer 17 and `channel_enable_0` is a 1, then the 17th bit of the top decoder will be a 1 and the rest will be zero. The corresponding bits of each decoder are OR'ed together, resulting in a 50-bit `channel_enable` vector that has 1's at each of the channels that the Centroiding Computation blocks are reading from. As the 8-bit data from each channel streams back on the 400-bit `channel_data` vector, an array of 64-to-1 multiplexers (MUX's) select the 8-bit data specified by each `current_channel` index.

The switching matrix is responsible for the dual tasks of transforming the independent read requests of each of the Centroiding Computation blocks into a single 50-bit `channel_enable` vector and routing the desired channel's data back to the appropriate Centroiding Computation block. The process is depicted visually in Figure 3.13. The switching matrix decodes the 6-bit `current_channel` index of each Centroiding Computation into a one-hot (meaning only one bit is a 1) vector with a 1 in the bit specified by `current_channel`. The decoded indices are OR'ed together into a single, 50-bit `channel_enable` vector. This enables multiple channels to be read from at the same time by different Centroiding Computation blocks.

On the return path, the switching matrix essentially performs the inverse operation on the 400-bit `channel_data` vector. It uses `current_channel` from each Centroiding Computation to multiplex a single channel's data onto the `channel_data_i` outputs. The Centroiding Computation block must therefore maintain the same `current_channel` from the beginning of the enable process to the arrival of the last byte of data. To ease the strain on the tools of routing this massively parallel matrix of signals, the outputs of the decoders, the OR gates, and the MUX's of Figure 3.13 each have at least one register stage, contributing to the 7-cycle delay mentioned in Section 3.5.2.

3.6 Coordinate Aggregator

The Coordinate Aggregator deserializes the centroid results from the Centroiding Computation blocks and writes them into an asynchronous FIFO (Figure 3.14). The state machine which controls the process is given in Figure 3.15. In Idle, it watches the vector of `centroid_done` signals and, if any of the bits go high, saves the current state of the vector – identical to the process described in Section 3.5.1. The logic first writes a "length" byte into the FIFO in the

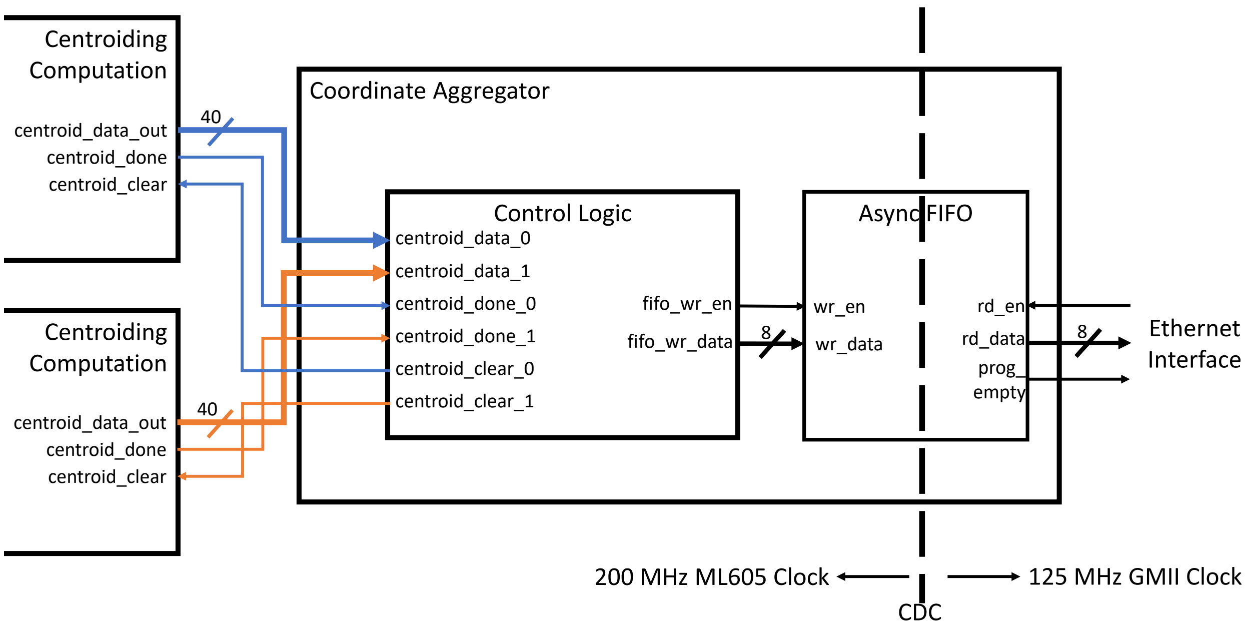

Figure 3.14: Coordinate Aggregator. The Coordinate Aggregator interfaces with each of the Centroiding Computation blocks to multiplex their results onto a single 8-bit `fifo_wr_data` signal. When a Centroiding Computation block finishes processing a cloud's data, it asserts `centroid_done`. The fully-parallel 40-bit `centroid_data_out` signal transfers the $(\hat{t}_x, \hat{x})$ centroid result in a single cycle. When the control logic (Figure 3.15) has saved the result from a Centroiding Computation block, it asserts `centroid_clear`, causing `centroid_done` to deassert and preventing reprocessing the same result. The logic then shifts the message out to the asynchronous FIFO. The read side of the FIFO is in the 125 MHz GMII clock domain and is handled by the Ethernet Interface (Section 3.7).

Header state, specifying the number of Centroid results that are about to be written. This functions, practically, as a sub-header within the Ethernet packet that makes parsing the packet's contents on the computer side simple and robust.

The logic then scans, reads, and shifts out the 40-bit $(\hat{t}_x, \hat{x})$ result from each finished Centroiding Computation block in Load. It also asserts `centroid_clear` on each processed Centroiding Computation block, acknowledging the result and causing the corresponding

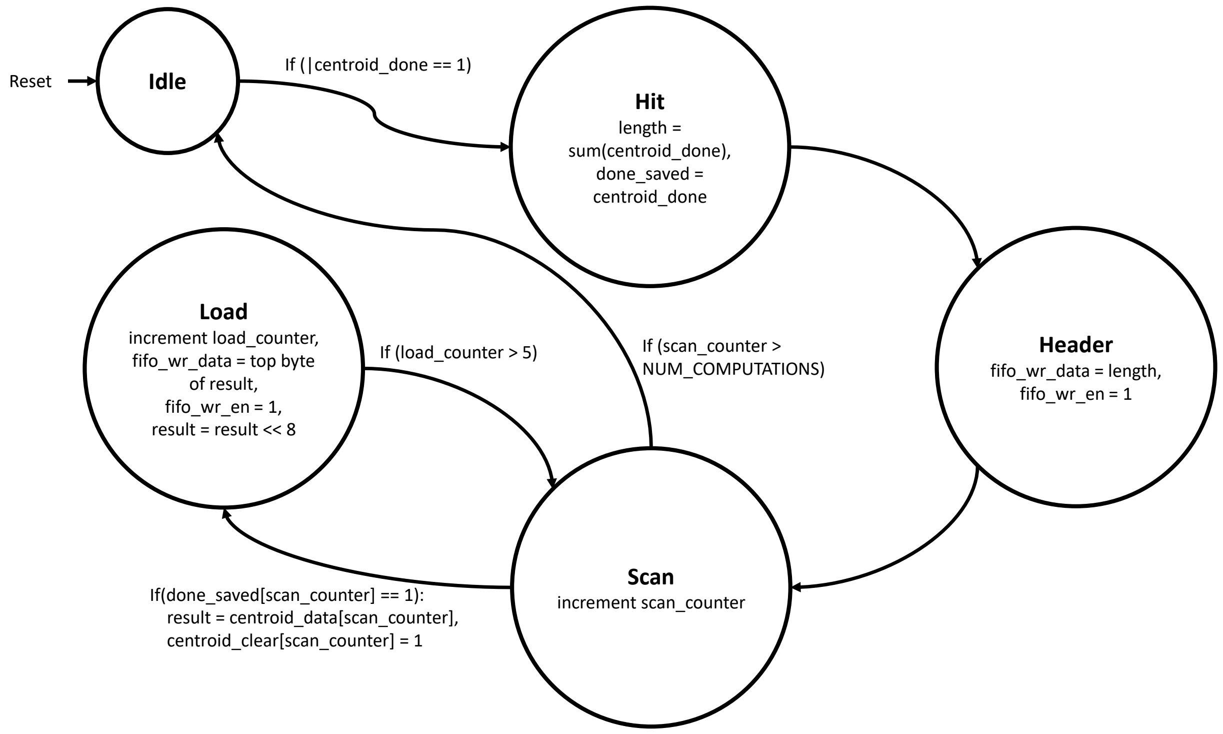
Reset
Idle
If (|centroid_done == 1)
Hit
length = sum(centroid_done), done_saved = centroid_done
Header
fifo_wr_data = length, fifo_wr_en = 1
Load
increment load_counter, fifo_wr_data = top byte of result, fifo_wr_en = 1, result = result << 8
If (load_counter > 5)
If (scan_counter > NUM_COMPUTATIONS)
Scan
increment scan_counter
If(done_saved[scan_counter] == 1): result = centroid_data[scan_counter], centroid_clear[scan_counter] = 1

Figure 3.15: Coordinate Aggregator Logic. The Coordinate Aggregator streams the centroid results ($\hat{t}_x, \hat{x}$) of each processed cloud into an asynchronous FIFO that is read by the Ethernet Interface. It watches the vector of `centroid_done` signals for any asserted bits, represented by checking the reduction OR: `|centroid_done`. If any bit is a 1, the reduction OR returns a 1 and the logic transitions to the Hit state. In Hit, the value of the `centroid_done` vector is saved as `done_saved` to prevent double counting. The `length` is the number of completed centroid results, found by summing the bits in `centroid_done`. This value has a maximum of the number of Centroiding Computation blocks instantiated in the current design. In Header, the `length` field is written as a byte into the FIFO. In Scan, the logic iterates across the vector of `centroid_done` signals. When the current `centroid_done` is a 1, the logic transitions to the Load state, saves the corresponding `centroid_data`, and asserts the corresponding `centroid_clear` bit to deassert the `centroid_done` flag and avoid double-counting. In Load, the logic left-shifts the `result` signal out into the FIFO 5 times, once for each byte. It then returns to the Scan state, and the cycle repeats. Once the last Centroid Computation block has been scanned, the logic returns to Idle.

`centroid_done` bit to be deasserted. The process of reading a result is much simpler than in previous components of this thesis due to the decision to implement a fully-parallel datapath for each Centroiding Computation block. A single clock cycle moves the entire 40-bit value into a register `result`. `result` is then shifted out 8 bits at a time, starting with the most significant integer bits of $\hat{t}_x$ and finishing with the fractional bits of $\hat{x}$.

3.7 Ethernet Interface

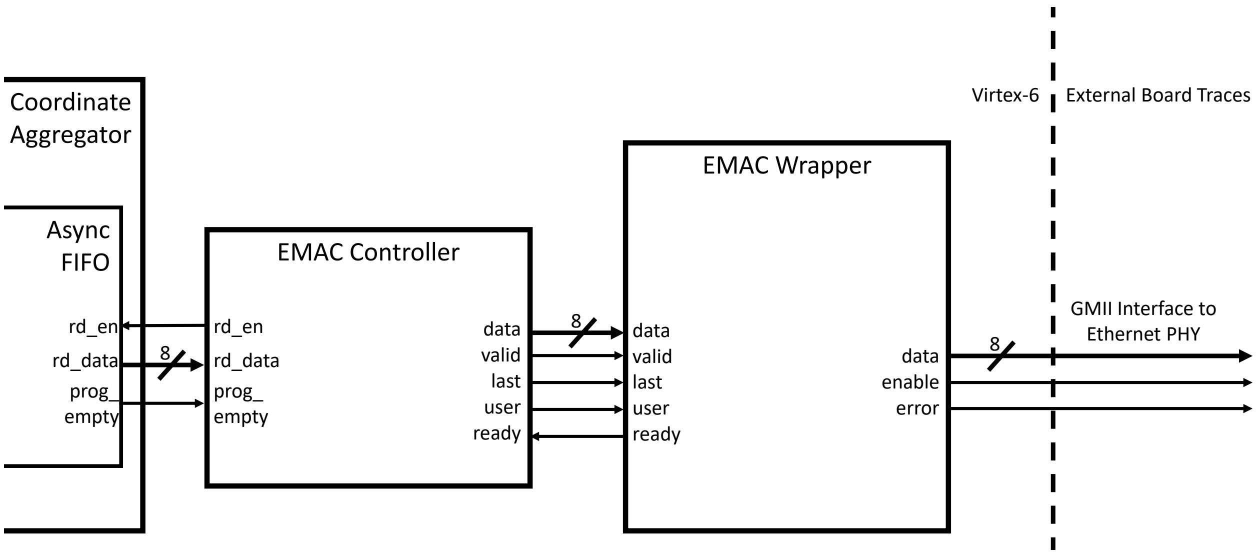

Figure 3.16: Ethernet Interface. The EMAC Controller reads the serialized centroid results out of the asynchronous FIFO and packages them into an Ethernet frame. Each frame is provided to the Xilinx EMAC Wrapper via the AXI4-S interface (`data`, `valid`, `last`, `user`, `ready`) [31]. The EMAC Wrapper is a verified, Xilinx-provided Ethernet MAC that interfaces with the external Ethernet PHY via the GMII interface [39]. It is responsible for driving each provided Ethernet frame according to the Ethernet specification (adding preamble, CRC, obeying the interframe gap requirement, etc). The standards-compliant Ethernet data is sent across the GMII interface and driven by the PHY according to the Ethernet standard's electrical specifications [32]. The ML605 board features an RJ45 connector which provides access to the generated Gigabit Ethernet signals.

The Ethernet Interface (Figure 3.16) consists of all the logic required to package the stream of centroid results into an Ethernet frame and transmit the frame to the Marvell 88E1111 Ethernet PHY, an external chip that handles all of the physical specifications of Ethernet (channel equalization, echo and crosstalk cancellation, etc) [32]. The primary component is the Xilinx Ethernet Medium Access Control (EMAC) Wrapper [39], a core provided via the Xilinx tools that makes use of the Virtex-6's hardened EMAC [40]. The Wrapper provides an industry-standard AXI4-Stream interface (AXI4-S) [31]. The only remaining piece of custom logic is the EMAC Controller 3.7.1, which simply packages the already 8-bit wide data in the asynchronous FIFO into an Ethernet frame under the AXI4-S specification.

3.7.1 EMAC Controller

The EMAC Controller (Figure 3.17) reads centroid result data from the asynchronous FIFO, packages it into an Ethernet frame, and encodes it into the AXI4-S interface. Like in the Channel Receiver (Section 3.3), the asynchronous FIFO's `prog_empty` flag is configured to deassert when a "full message" has been written into the FIFO. In this case, the threshold for deassertion is set at a high level that minimizes wasted bandwidth. When an Ethernet packet is transmitted with a small payload, the ratio of data to "packaging" (destination and source MAC addresses, CRC bits, etc) is low. This wastes precious bandwidth and should be avoided. On the other hand, if the threshold for deassertion is set too high, the FIFO could fill due to the faster 200 MHz write clock compared to the 125 MHz read clock. The balance of these two considerations depends ultimately on the rate at which centroids are computed – a parameter `PROG_EMPTY_THRESH` that will be tested experimentally when the MCP-PMT is complete. For the test setup presented in the next chapter, it is set at 64 bytes.

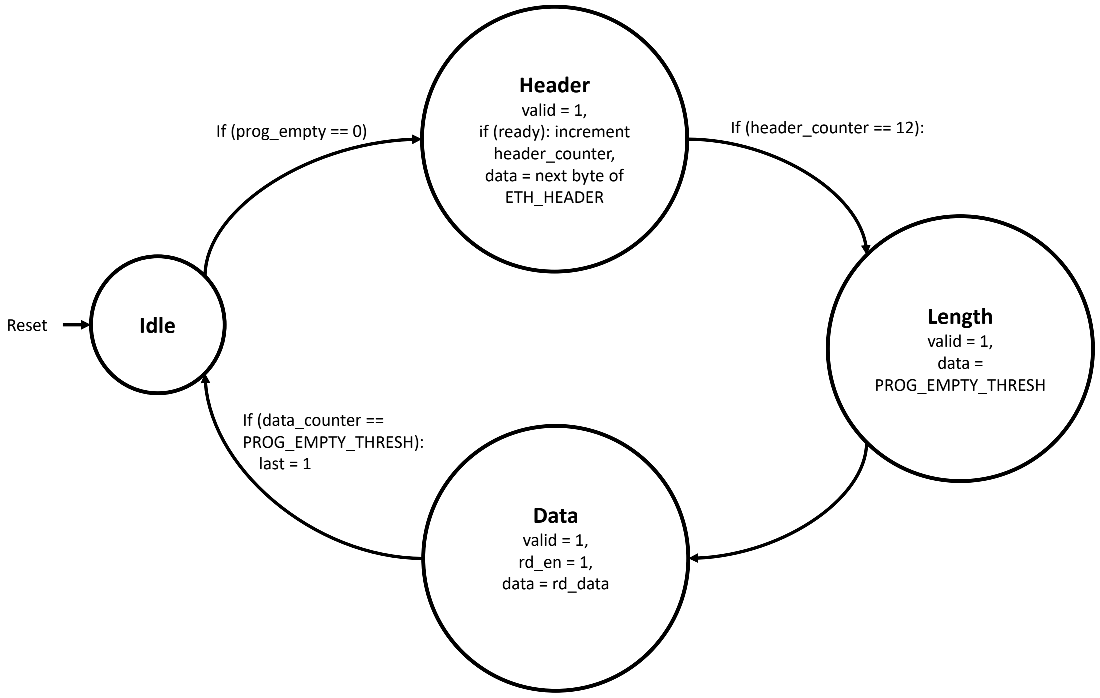

Figure 3.17: EMAC Controller Logic. The EMAC Controller logic reads `PROG_EMPTY_THRESH` bytes out of the asynchronous FIFO, packages them into an Ethernet frame, and sends the frame to the EMAC via the AXI4-S interface. The logic waits for the FIFO's `prog_empty` signal to deassert before beginning transmission of an Ethernet frame. First `valid` is asserted and the 12 bytes of `ETH_HEADER` are shifted out on `data` in the Header state. `ETH_HEADER` consists of a 6-byte destination MAC address, the broadcast address `FFFFFFFFFFFFh`, and a 6-byte source MAC address of the Ethernet Interface, `002320212223h`. If at any point, the EMAC deasserts `ready`, the logic will pause here. This allows for flow-control pauses in the EMAC. Next, the 2-byte Ethernet length field is filled with the `PROG_EMPTY_THRESH` constant, the number of bytes known to be in the FIFO (because of the deassertion of `prog_empty`). In Data, the logic reads out `PROG_EMPTY_THRESH` bytes from the FIFO and writes them onto the AXI4-S interface. On the last byte, the logic asserts `last` and transitions back to Idle.

3.8 Discussion

The Data Processing FPGA is a massively-parallel computational platform. At a most fundamental level, the design processes an impressive maximum input data rate of 50 channels $\times$ 125 Mbps/channel = 6.25 Gbps. This is possible due to efficient regional clock distribution, extensive use of pipeline registers, and the strategic offloading of algorithmic complexity to each Channel Receiver. In every instance where decision-tree style multiplexing was unavoidable, for example in the implementations of Algorithms 1 & 2, immense effort was made to minimize the "depth" of the underlying combinational logic by simplifying logical expressions and consolidating related signals to minimize resources. Because timing success was difficult to attain and utilization of the FPGA's resources was high, small improvements (especially in logic instantiated 50 times) led to big gains in device performance.

The computational and throughput performance of the design is characterized at length in Chapter 4. The impressive accuracy and precision of computing centroids validate the decisions to choose a relatively simple center of gravity algorithm and use only 8 bits of fixed point fractional precision throughout. While flashier algorithmic solutions exist, the design's throughput performance would have been incredibly difficult to achieve on such a massively parallel scale had more complicated computational tasks been required. From the non-restoring division implementation to the three-stage MAC block, every math component was designed to precisely compute results without burdening the tools with difficult routing tasks.

On a higher level, although the Data Processing FPGA presented here is a robust solution to the centroiding problem for the MCP-PMT system, it will undoubtedly be revised and improved once the system is operational. The fact that the upstream system is not yet finished was certainly one of the most difficult factors I faced during the course of this project. Numerous unknowns and untestable assumptions about the quality of the MCP-PMT's signal content discouraged investing hundreds of hours into a complex component that might be rendered useless by the final design. Instead, I focused on modularity and simple, elegant interfaces. If a component needs to be modified or completely replaced, the entire system won't collapse – the engineer merely has to obey the interface's specifications on data format and control signals.

The most compelling example of this modularity is the Centroiding Computation block. A parallel array of these blocks feature a simple set of control signals for reading the Channel Receiver's preliminary $(\hat{v}_i, \hat{t_i})$ results. On the back-end a fully parallel interface presents the computed $(\hat{x}, \hat{t_x})$ centroids to the Coordinate Aggregator. Nowhere is it specified *when* or *how* each Centroiding Computation block generates the centroids. This flexibility opens the door for future research on a heterogeneous computational platform, in which a range of different computational blocks compute results with varying degrees of accuracy, precision, and delay. One could imagine an intelligent Centroiding Arbiter switching between different algorithmic complexities depending on the cloud input rate.

In terms of design methodology, I attempted to create a plug-and-play, modular design that emphasizes code reuse and parameterizability throughout. It therefore shares much more in common with the software engineering design ethos than the average hardware design project. A project of this magnitude on this timescale was only feasible for a single person because of the extensive use of parameters for almost every characteristic of the design – everything from the number of samples in the FADC's message to the number of Centroiding Computation blocks to instantiate. Changing a single parameter in the top level module propagates down through the design hierarchy automatically. Additionally, code was reused (either explicitly by module instantiation or implicitly by structure) in the Channel Computation and Centroiding Computation blocks. Both of these efforts were ultimately effective because they enabled rapid prototyping – a cycle of small, incremental changes to an already functional design – rather than long, risky development cycles.

As Chapter 4 will show, the dual concerns of low-level logic optimization and high-level, forward-looking design choices culminated in a high-performance computational platform that delivers both excellent spatiotemporal resolution and system stability in the complex scenario of thousands of stochastic photon arrivals distributed in space and time.

Chapter 4

Testing

The Data Processing FPGA is a complicated system, so testing it thoroughly requires multiple approaches. At a most fundamental level, the design estimates three quantities for each cloud it receives from the FADC crate: the center position of the cloud $\hat{x}$, the time of the cloud's arrival $\hat{t}_x$, and the cloud's spatial variance (or width) $\widehat{w_x^2}$. These three quantities are parameterizations of an underlying physical phenomenon, namely the charge cloud's arrival to the cross-strip anode (Section 1.2.1). The performance of each estimator is limited by numerous factors including: the parameters of the charge cloud; the cloud's proximity to other clouds (both spatially and temporally); the noise introduced in the digitization electronics; various parameters in both the FADC Channel FPGAs and the Data Processing FPGA; and – most fundamentally – the mathematical implications of the finite-sample center-of-gravity approach used to compute the estimates.

Because testing over all the possible combinations of each of the above factors is impossible, a handful of test scenarios have been chosen to test the performance of the Data Processing FPGA and to guide the integration of the earlier stages of the MCP-PMT into a functioning imaging system. The experiments demonstrate that the Data Processing FPGA is a robust estimator of the spatiotemporal coordinates of electron clouds (and by extension photons).

The final, all-encompassing "stress test" shows that the design maintains this performance even in the presence of thousands of stochastically generated clouds.

The following sections explain the underlying testing framework and then present the set of test scenarios that have been developed. Section 4.1 presents the physical test setup used. Section 4.2 shows how the assumptions of a cloud's Gaussian spatiotemporal charge distribution are implemented in Python to generate test data. Sections 4.3-4.4 each present a test scenario and the Data Processing FPGA's performance therein.

4.1 Test Setup

The test scenarios in Sections 4.3-4.4 are carried out in the system shown in Figure 4.1. Although an analog test making use of the FADC boards was planned, COVID-19 related lab closures meant that the FADC boards weren't accessible for incorporation in the test loop. A portable mock-up of their functionality was therefore designed and instantiated inside the Virtex-6 FPGA on-board the ML605. Additionally, only a single ML605 board was available for use. These two constraints led to the instantiation of both the X and Y dimension mock FADC boards and Data Processing FPGA instances within the same Virtex-6 FPGA. The test setup presented here is therefore a hybrid of the standard FPGA testing paradigm – a simulation – and the originally planned analog test featuring physical pulses injected into the FADC crate. Ironically, the result is a scheme that gets the best of both worlds: complete control over the input test vectors (usually only possible in simulation) *and* a full verification of the Data Processing FPGA's physical interfaces.

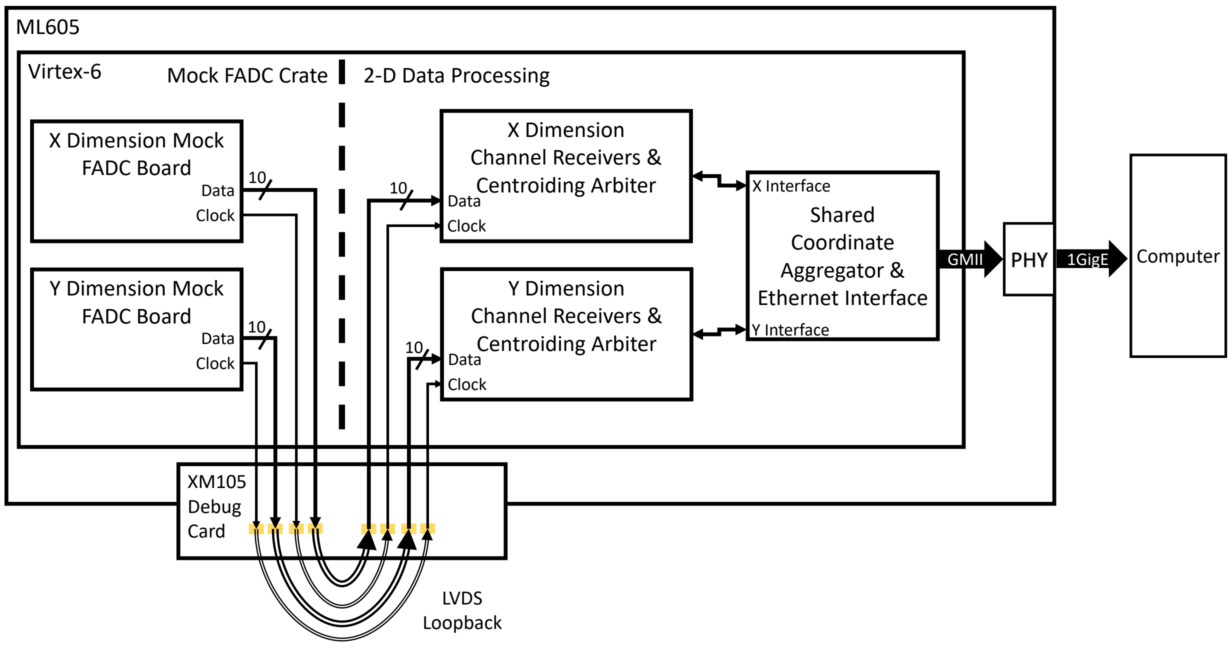
ML605
Virtex-6
Mock FADC Crate
2-D Data Processing
X Dimension Mock FADC Board
Data
Clock
10
Y Dimension Mock FADC Board
Data
Clock
10
X Dimension Channel Receivers & Centroiding Arbiter
Data
Clock
10
Y Dimension Channel Receivers & Centroiding Arbiter
Data
Clock
10
X Interface
Shared Coordinate Aggregator & Ethernet Interface
Y Interface
GMII
PHY
1GigE
Computer
XM105 Debug Card
LVDS Loopback

Figure 4.1: Test Setup. The Virtex-6 FPGA onboard the ML605 houses both the stimulus (mock FADC crate) and the device-under-test (2-D Data Processing). Because only a single ML605 board was available, both the X and the Y dimensions are instantiated inside the same FPGA. This increased pin consumption on the ML605 limited the possible number of channels in each dimension to 10. Each mock FADC board features 10 mock FADC Channels that drive out their messages over 10 LVDS links according to the serial protocol of Section 3.2.2. These links are connected with loopback cabling on the XM105 Debug Card to pins that lead directly to the Channel Receivers. The Centroiding Arbiter in each dimension performs the centroiding computation on the preliminary measurements of the Channel Receivers and drives the data to a shared Coordinate Aggregator and Ethernet Interface. The centroid estimates are streamed to a computer via a Gigabit Ethernet link and captured using Wireshark.

The cloud model of Section 2.1 is used to generate the pulse data from a sequence of cloud arrivals that has the characteristics specified by the test. For example, Section 4.3's test involves scanning a cloud across the cross-strip anode. For each cloud, a time of arrival and a set of pulses in each dimension are generated and saved into memory on the FPGA. The mock FADC boards send the pulses for each cloud to the Data Processing FPGAs over an external LVDS loopback when their internal timer reaches the time of arrival. The parameters of each cloud can thus be used as a base "truth" against which the estimators can be tested. At the same time, the integrity of the LVDS and Ethernet links and the overall stability of the system are verified by ensuring that all of the clouds were successfully processed by the FPGA and received by the computer.

4.1.1 Mock FADC Board

The Mock FADC board is responsible for transmitting the pulse data for each input cloud in the test across the LVDS interface. As shown in Figure 4.2, it features two kinds of RAMs, denoted Pulse and Time. Each Channel's Pulse RAM holds a sequence of pulses, one for each cloud in the test. The RAM width is 32 bits, and each pulse consists of 12 8-bit samples. Each pulse is therefore represented in 3 contiguous addresses within the RAM. The samples are loaded with the earliest placed at the lowest address. The Time RAM holds a list of start times, one for each cloud, that tell the Controller when each cloud should "arrive" and be sent across the LVDS loopback. The Controller simply waits for its internal timer to match the time specified by the next address in the Time RAM, then asserts `enable` and reads the next three 32-bit Pulse RAM words into the FADC Channel blocks. The Controller is very simple and does not consider whether or not the Channels are capable of transmitting a new cloud – it merely iterates through the Time RAM and loads the 12 samples belonging to each cloud into the FADC Channels each time a cloud "arrives." It therefore approximates the function of the ADC's on the FADC board.

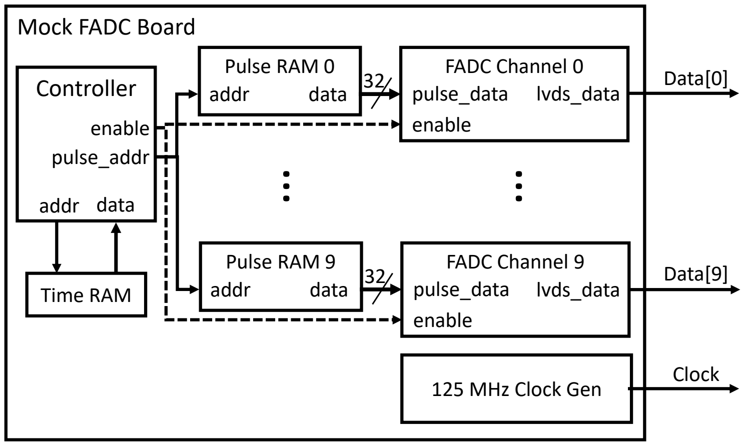

Figure 4.2: Mock FADC Board. The Mock FADC Board is built around Pulse RAMs that hold the pulse content of each input cloud and a Time RAM that hold a list of start times, one for each cloud. The Controller iterates through the Time RAM, asserting `enable` to begin transmission of a cloud's data when its internal timer matches the Time RAM's `data`. In this test setup, each pulse contains 12 8-bit samples. Each word in the pulse RAM is 32 bits, or 4 samples. The Controller therefore increments `pulse_addr` for three consecutive clocks when it asserts `enable` to shift all 12 samples into the FADC Channel. The FADC Channel then drives the 12 samples and a 16-bit timestamp onto the LVDS Interface (Section 3.2) according to the protocol in Section 3.2.2.

The FADC Channel Blocks, as mentioned in Section 3.2.2, are constrained by the fact that the transmission of a single pulse's 120-bit message takes 960 ns. Once a message is initiated by the Controller's assertion of `enable`, the Channel Blocks save the 12 samples and the timestamp of the first sample. In Section 4.4's test, cloud arrival times are random. This means the Controller will sometimes assert `enable` while the FADC Channel blocks are busy sending a message. In this case, the new samples are ignored and the message is finished as if a new cloud had never arrived. As explained in Section 4.4, this feature combined with the self-triggering check described below create difficult partial-cloud or multi-cloud situations.

Before the FADC Channel can begin transmission, it performs the self-thresholding check described in Section 1.2.2. This verification is done out of a concern for test integrity. Even though the mock FADC Channel knows that a cloud has "arrived" because `enable` was asserted, in the actual FADC Boards there is no `enable` trigger. Each Channel has to decide *independently* if there is a valid pulse in the stream of data from the ADC, a process called self-triggering. For this test setup, the digital threshold is set at decimal 22, a value slightly higher than the ADC's zero-volt pedestal of 17 (Section 3.3.1). If any of the 12 samples in a pulse are greater than or equal to 22, the FADC Channel begins sending the 8-bit header `AAh`. It proceeds to send the rest of the message (timestamp and 12 samples) to the Channel Receiver as specified in Section 3.2.2.

4.1.2 Data Processing FPGA Configuration

The baseline configuration of the Data Processing FPGA for these tests is presented here. Tests described in the later sections modify the following scheme. The minimum number of channels in a cloud to be considered "valid" by the Centroiding Arbiter's control logic is set to 3 (see `MIN`, Section 3.5.1), which ensures that any cloud in the following tests will be accepted by the Arbiter. The number of Centroiding Computation blocks per-dimension was set to 1. The reduced size of the anode means that the massive flux of adjacent clouds that formed the basis of featuring multiple Centroiding Computation blocks is not relevant here.

The Centroiding Arbiter was modified to include the estimated spatial variance $\widehat{w_x^2}$ in the transmitted data so the performance of the estimator can be evaluated. Because $\widehat{w_x^2}$ is $U(8.8)$, this simply adds two bytes to each dimension's result, for a total of 7. Because the Coordinate Aggregator is now streaming results from two separate dimensions, a modification is made to its “header” byte (Section 3.6). The most-significant 4 bits convey the number of subsequent centroid results in the X dimension and the least-significant 4 bits convey the number of subsequent Y dimension results. The results from the X dimension are loaded first. Because each dimension only has one Centroiding Computation block (in most tests), the max value either of these fields will take is 1. It is valuable nevertheless because there is occasionally a staggering effect where the results from the X and Y dimensions will be completed at different times. In this situation, the Coordinate Aggregator will load either an X or a Y result, with the modified header byte conveying to which dimension the result belongs.

4.1.3 Computer Setup

The computer connected to the Ethernet link of Figure 4.1 runs a Wireshark packet capture application, specifically the `tshark` command [41]. The application appends the bytes of each packet into a text file. A Python application, `parse_ethernet_bytes` then scans over the packets, trimming each packet's header and concatenating the payloads into an array. The result is a byte sequence conveying the entirety of the Data Processing FPGA's data from both dimensions. A second application iterates over the byte sequence and uses the Coordinate Aggregator's headers to delineate results. Because each $\hat{x}$, $\hat{t}_x$, and $\widehat{w_x^2}$ estimate is in a fixed point format with 8 fractional bits, they are interpreted as integers and then divided by $2^8 = 256$, yielding a Python floating-point approximation of the underlying fixed point number. The application reads the raw byte data comprising the 6 centroid estimates (3 for each dimension) and writes each of the 6 floating point results into separate data files. The files are then processed to compute and visualize the relevant statistics for each test scenario.

4.2 Python Implementation of Cloud Model

A Python implementation of the model of the charge cloud's collision with the cross-strip anode (Section 2.1) is used to create the data stored in the RAMs of Section 4.1.1. This section provides a short commentary on the Python implementation of the model and the means by which the data was loaded into the FADC Board Pulse RAMs and Time RAM.

A Python function, `generate_pulse_arrays`, generates the 20 pulses (one for each strip, stored in the Pulse RAMs) and a `start_time` (in nanoseconds, stored in the Time RAM) using the above theoretical model. It is called by the test script for each cloud in the test and takes as inputs the parameters of each cloud: $(\mu_x, \mu_y, \mu_t, w_{xy}^2, w_t^2, \sigma_z^2)$. Because the intent of these tests is not to model the effect of gain variability, the scaling constant A was fixed at 3000, putting the peak voltages of each cloud in the middle of the ADC's range as desired. The cross-strip anode parameters $(x_0, ...x_9, y_0, ...y_9, W_x, W_y, L)$ were also fixed constants.

When `generate_pulse_arrays` is called, it first determines `start_time`, the time when the Controller asserts `enable` and therefore the timestamp of the first sample in the pulse (Section 3.3.1). Crucially, the Controller operates on a 125 MHz clock, meaning it can only "start" in increments of 8 ns. Intuitively, the `generate_pulse_arrays` function should try to put the pulse center time μ_t as close to the middle of the 12-sample, 22 nanosecond window as possible – otherwise pulse content would be lost either before or after the window. It therefore assigns `start_time` using the modulus operator mod:

$$\texttt{start\_time} = \mu_t - (\mu_t \bmod 8) - 8 \tag{4.1}$$

This assigns `start_time` to the multiple of 8 that is between 8 and 16 nanoseconds lower than μ_t. This places μ_t as close to the middle of the window as possible. For example, if $\mu_t = 27$:

$$\texttt{start\_time} = 27 - (27 \bmod 8) - 8 = 16\,\text{ns} \tag{4.2}$$

The pulse vectors $V_{X,i,t}$ & $V_{Y,j,t}$ are computed every 2 nanoseconds beginning with `start_time`. These times are stored in a vector `sample_times`:

$$\texttt{sample\_times} = [\texttt{start\_time}, \texttt{start\_time} + 2, ..., \texttt{start\_time} + 22] \tag{4.3}$$

The `generate_pulse_arrays` function then creates the pulse envelope vector `pulse_vector` by evaluating $\exp\left(-(t-\mu_t)^2/2w_t^2\right)$ at each time in `sample_times`. The function then computes the strip scaling constants given by the integral of C over each strip's area. As mentioned above, separability makes this easier because the X and Y integrals can be computed independently and their results multiplied together. Computing an integral explicitly in Python's `Scipy` package is needlessly difficult because the integral over a portion of a normal distribution can be expressed in terms of the error function erf:

$$\int_a^b \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x-\mu_x)^2}{2w_x^2}\right) dx = \frac{1}{2}\left(1 + \text{erf}\left(\frac{b-\mu_x}{\sqrt{2}w_x}\right)\right) - \frac{1}{2}\left(1 + \text{erf}\left(\frac{a-\mu_x}{\sqrt{2}w_x}\right)\right) \tag{4.4}$$

$$= \frac{1}{2}\left(\text{erf}\left(\frac{b-\mu_x}{\sqrt{2}w_x}\right) - \text{erf}\left(\frac{a-\mu_x}{\sqrt{2}w_x}\right)\right) \tag{4.5}$$

The `math` package in Python offers a numerical approximation of the error function. It is used to compute the spatial integrals of Equations 2.2 & 2.3. The $\frac{1}{\sqrt{2\pi}}$ normalizing factor in the PDF of Equation 4.5 is not necessary in the strip voltage equations because the A scaling factor is responsible for scaling the charge cloud's gain to generate appropriately-sized pulses.

The product of each strip's X and Y integrals (Equations 2.2 & 2.3) and the voltage scaling constant A is then applied as a strip-dependent scaling factor on the generic pulse time envelope `pulse_vector`. This product of strip-dependent and strip-independent components yields a set of noiseless 12-sample pulse vectors – one for each strip. Finally, white Gaussian noise $Z \sim \mathcal{N}(0, \sigma_z^2)$ is added to every sample, creating the final pulse vectors of Equations 2.4 & 2.5.

Each of the 12-sample pulse vectors are then rounded to integers and converted to hexadecimal format. Each pulse vector is appended to a file holding a list of all the pulses on that strip (20 files total). Similarly, the first sample time `start_time` is loaded into a file holding each cloud's first sample time. These files are loaded into the Pulse RAMs and the Time RAM using the Verilog `readmemh` command. The Verilog compiler parses their contents and wires up the correct configuration of the Virtex-6's onboard RAM. Because each test has thousands of clouds, the number of bits in a Pulse RAM exceeds $1000\,\text{clouds} \times 12\,\text{samples} \times 8\,\text{bits} = 96\,\text{Kb}$. The Virtex-6 RAMs hold 36 Kb [33], so the tool automatically connects multiple individual RAMs and handles addressing and control signals "beneath the hood."

4.3 Position Scan Test

This test seeks to answer the question: how does the performance of the Data Processing FPGA relate to the position of the charge cloud? Answering this question provides two crucial results. First, if the Data Processing FPGA performs well across the full spectrum of input positions, it validates both the Centroiding Algorithm in the abstract and the hardware implementation of it on the FPGA. Second, the results should provide useful information for the future of the MCP-PMT imaging system. This is because the test setup is a good approximation of the physical device, so any interesting trends discovered could inform the MCP-PMT's assembly and/or future research projects.

Section 4.3.1 explains the method by which the cloud's position was varied and defines the relevant parameters. Section 4.3.2 presents the results and a detailed discussion of the Data Processing FPGA's performance in this test.

4.3.1 Test Operation and Parameters

The center position of the charge cloud (μ_x, μ_y) was scanned across the X dimension of the cross-strip anode at discrete positions from the center of the left-most X strip to the center of the right-most X strip. Starting from the left-most strip, each subsequent μ_x position is a fixed interval Δx farther to the right. At each position, $N = 25$ clouds were sampled to allow for computing the statistics described in Section 2.5. The cloud's center in the y dimension was fixed in the middle of the anode. The center time of each input cloud μ_t is $\Delta t = 1500\,\text{ns}$ after the previous cloud. This well exceeds the 960 ns dead-time incurred when an FADC Channel sends a message and guarantees that every input cloud will be fully processed.

Table 4.1: Scan Test Parameters.

Parameter	Value
A	3000
(μ_x, μ_y)	(varies, 4.5)
w_{xy}^2	1 strip^2
μ_t	varies
w_t^2	3 ns^2
σ_z^2	1 count^2
Δx	0.05 strips
N	25 clouds/position
Δt	1500 ns
C	4525 clouds

All subsequent discussions and tests use a position unit equal to 1 strip pitch – for brevity, 1 "strip." The bottom left corner of the 10×10 anode is therefore position $(0,0)$, the middle of the anode is position $(4.5,4.5)$, and the top right corner is position $(9,9)$. The position interval Δx is, in units of strip pitch, 0.05. This test therefore iterates across the range of positions: $[(0,4.5),(0.05,4.5),...,(8.95,4.5),(9,4.5)]$. Twenty five ($N = 25$) clouds are sampled at each position, for a total number of clouds in the test $C = (9/.05+1) \cdot 25 = 4525$ clouds.

4.3.2 Results and Discussion

Because the position of the input cloud is the variable in this test, the performances of the position-based estimators $\hat{x}$ and $\widehat{w_x^2}$ are presented here. Figure 4.3 presents the bias and standard deviation of the $\hat{x}$ estimator as a function of the input cloud position. Figure 4.4 presents the bias and standard deviation of the $\widehat{w_x^2}$ estimator as a function of the input cloud position. See Section 2.5 for definitions of bias and standard deviation.

Edge Effects

Examining Figures 4.3 & 4.4, one immediately notices the large bias in both estimators when the cloud moves to the edges of the anode, known as "edge effects." Recall from Table 4.1 that the cloud's spatial variance $w_{xy}^2 = 1$, and therefore its spatial standard deviation w_{xy} is also 1. Because the cloud is normally distributed, 95% of the cloud's charge distribution lies within ± 2 strips from the center of the cloud. As the cloud moves farther to the left from $\mu_x = 2$ or farther to the right from $\mu_x = 7$, more than $5\%/2 = 2.5\%$ of the charge content is "lost" beyond the edges of the anode.

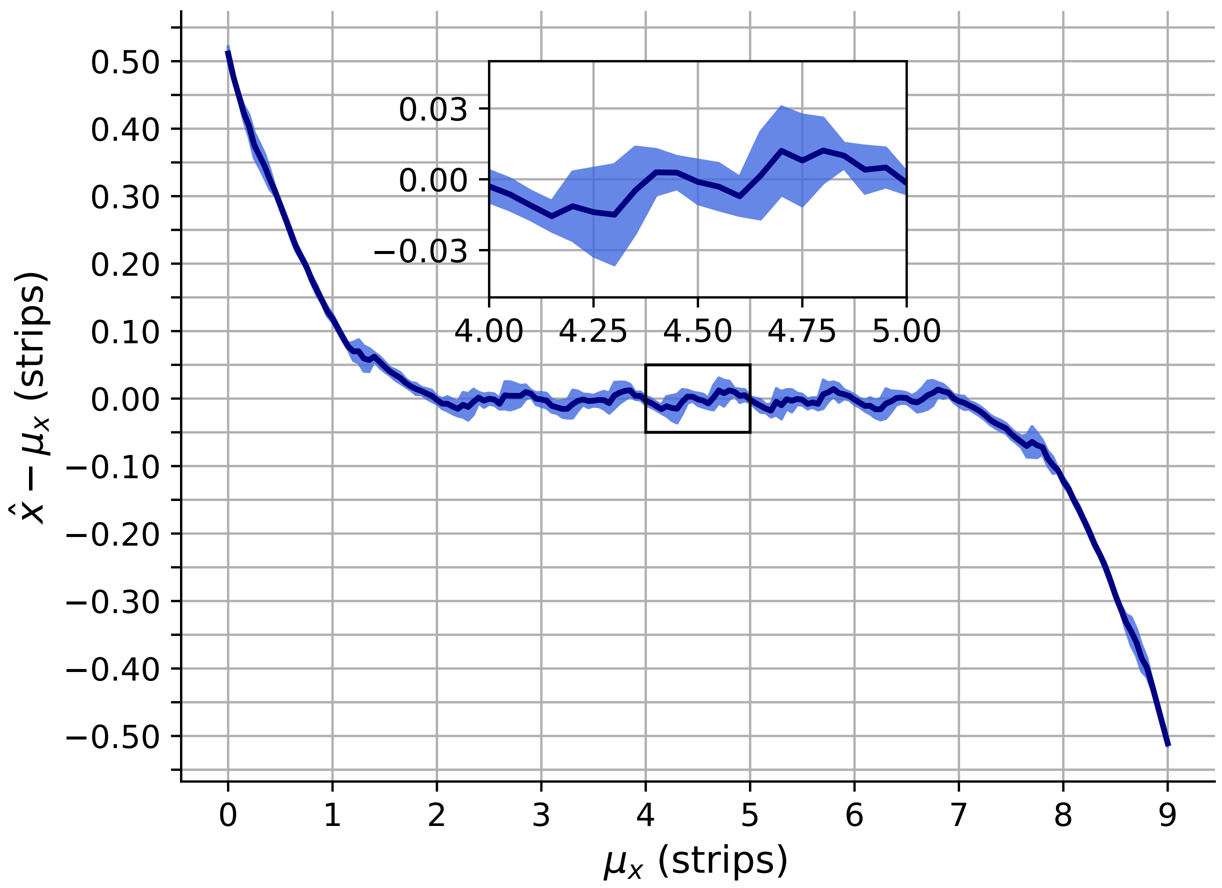

Figure 4.3: Performance of $\hat{x}$ estimator in Position Scan Test. The bias $B(\hat{x})$ and standard deviation $s(\hat{x})$ are plotted as a function of μ_x, the cloud's X-dimension center position. The standard deviation is plotted as a shaded region $\pm s(\hat{x})$ above and below the bias. The position μ_x is scanned from the left edge of the anode to the right edge with interval $\Delta x = .05$, taking $N = 25$ clouds at each position. Regions of large bias occur on the edges of the anode where the cloud's charge falls beyond the outermost strip. Regions of moderate bias and standard deviation occur periodically in the middle of the anode due to any particular FADC Channel crossing the threshold for self-triggering. Across the middle of the anode, $2 < \mu_x < 7$, the average standard deviation is $\bar{s}(\hat{x}) = 0.011$ strips (2.8 µm). See Section 4.3.2 for an extended discussion of these results.

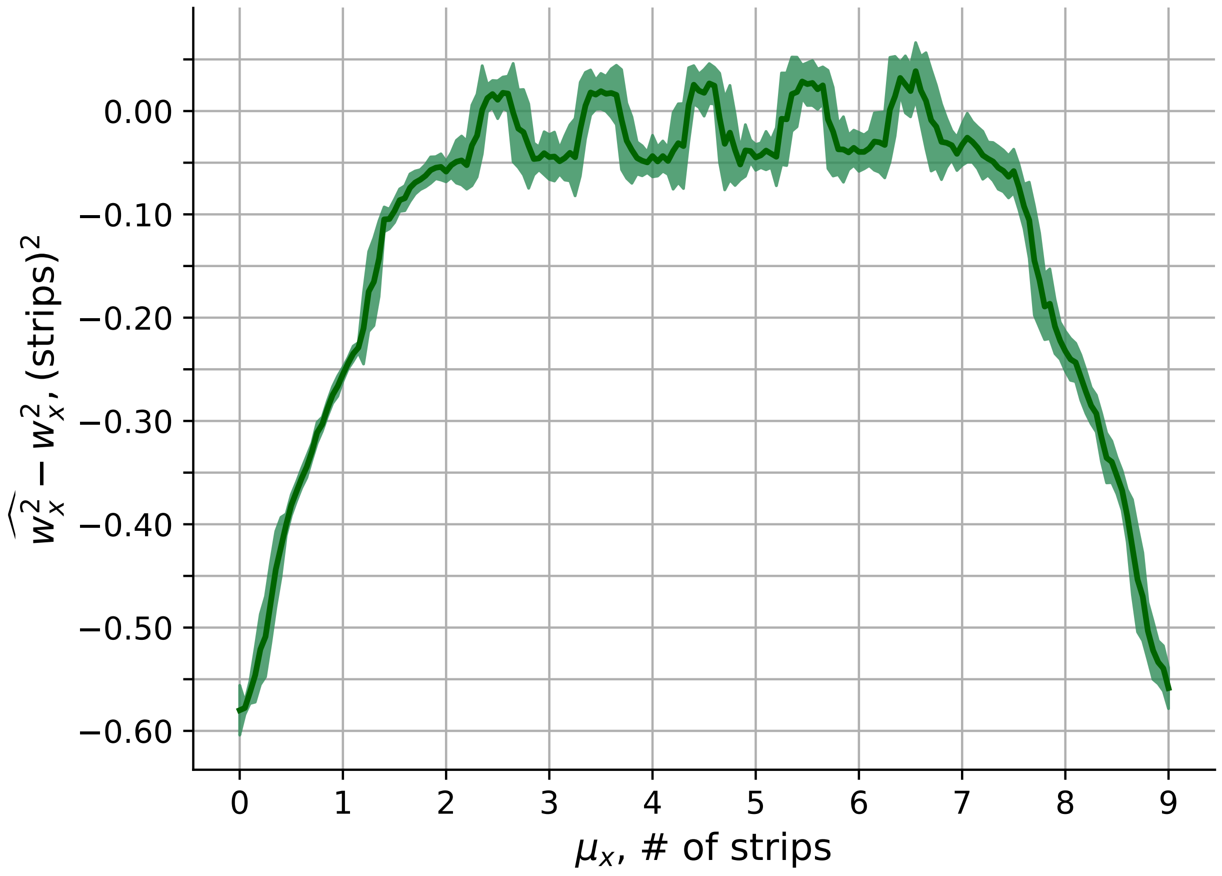

Figure 4.4: Performance of $\widehat{w_x^2}$ estimator in Position Scan Test. The bias $B(\widehat{w_x^2})$ and standard deviation $s(\widehat{w_x^2})$ are plotted as a function of μ_x, the cloud's X dimension center position. The standard deviation is plotted as a shaded region $\pm s(\widehat{w_x^2})$ above and below the bias. The position is scanned from the left edge of the anode to the right edge with interval $\Delta x = .05$, taking $N = 25$ clouds at each position. Each cloud's spatial variance $w_{xy}^2 = 1$. Large negative bias occurs on the edge of the anode as the cloud's charge content is lost beyond the outermost strip, causing $\widehat{w_x^2}$ to underestimate the cloud's width. Across the middle of the anode, $2 < \mu_x < 7$, the average bias is $\bar{B}(\widehat{w_x^2}) = -0.016$ strips^2 and the average standard deviation is $\bar{s}(\widehat{w_x^2}) = 0.026$ strips^2. The periodic oscillations in $B(\widehat{w_x^2})$ are a result of the FADC Channel's self-triggering behavior: see Section 4.3.2 for an extended discussion of these results.

As the cloud continues moving to the edges, more and more charge content is lost until, at the edge, only 50% of the cloud remains on the anode's active area. Each step beyond $\mu_x < 2$ or $\mu_x > 7$ therefore creates more and more inward-trending imbalance in the anode's perception of the cloud. This edge effect causes $\hat{x}$ to systematically *underestimate* the cloud's distance from the center of the anode. This presents as positive bias on the left edge and negative bias on the right edge. The effect on $\widehat{w_x^2}$ is negative on either side, as it is an estimate of w_{xy}^2. When the cloud moves to the edges, it appears narrower due to the loss of charge beyond the edge strip.

Interior Performance

The performance of the estimators in the region where the charge cloud falls nearly entirely on the anode, $(2 < \mu_x < 7)$, is excellent. In this range, the average bias in $\hat{x}$, $\bar{B}(\hat{x})$ is -0.002 strips. The average standard deviation of $\hat{x}$ is $\bar{s}(\hat{x}) = 0.011$ strips. Across the same range, the average bias in $\widehat{w_x^2}$ is $\bar{B}(\widehat{w_x^2}) = -0.016$ strips^2. The average standard deviation in $\widehat{w_x^2}$ is $\bar{s}(\widehat{w_x^2}) = 0.026$ strips^2. These values are presented in Table 4.2.

Table 4.2: Scan Test Results for $(2 < \mu_x < 7)$.

Estimator	Average Bias $\bar{B}$	Average Standard Dev $\bar{s}$
$\hat{x}$	-0.002 strips	0.011 strips
$\widehat{w_x^2}$	-0.016 strips^2	0.026 strips^2

Image Resolution

To determine the image resolution implied by this test, assume that the full 50×50 anode is used and achieves similar results. We assume also that clouds with positions in the outer two strips on both sides of both dimensions are discarded due to the edge effects – although in the future they could be corrected (Chapter 5). The average value of $s(\hat{x})$ across the retained interior region, $(2 < \mu_x < 47)$, is $\bar{s}(\hat{x}) = 0.011$ strips. Choosing a pixel width of $2\bar{s} = 0.022$, a single dimension of the image has $p = 45$ strips $\times$ 45 pixels/strip $=$ 2025 pixels. The image resolution is therefore $p^2 = 4.1$ megapixels, achieving the desired megapixel resolution mentioned in Chapter 1. Although the parameters of actual clouds in the MCP may differ from those in Table 4.1, this is a conservative value (due to discarding the outer strips) that should be within an order of magnitude of the performance achieved in real operation.

Bias Oscillation

The spatial variance estimate $\widehat{w_x^2}$ also performs well, but because it is an estimate of a second moment it is more sensitive and therefore has a more exaggerated bias oscillation. Examining the oscillation in $B(\widehat{w_x^2})$ helps explain the less-obvious oscillation in $B(\hat{x})$: put simply, they are both consequences of the self-triggering functionality of the FADC Channels. If a Channel has very little signal content in its pulse – specifically if it never reaches the threshold of decimal 22 – it won't self-trigger and the Data Processing FPGA will never see the Channel's pulse. This means pulses on the edge of the cloud aren't included in the computation.

The pattern of oscillation can be understood by examining the effect of perturbations to the left or the right of a charge cloud centered at $\mu_x = 4.5$. The bias in both estimators is very close to 0 at $\mu_x = 4.5$ because the cloud is symmetrically positioned on the strips. By examining the pulse data, Channels 2–7 are "on" – meaning their pulse exceeds the threshold and they send a message – at this position. As the cloud moves to the right towards $\mu_x = 5$, Channel 2's pulse height begins to drop, first to the level of, and then below, the threshold of 22. At $\mu_x = 5$, Channel 2 never triggers and the "cloud" only consists of Channels 3–7. This loss of a strip causes a systematic negative bias in $\widehat{w_x^2}$ at $\mu_x = 5$ that exceeds the positive bias at $\mu_x = 4.5$.

The effect of the strip loss on $B(\hat{x})$ is much less pronounced, and can be better understood in terms of symmetry. At $\mu_x = 4.5$, the cloud is symmetric with the strips and $\hat{x}$ is unbiased. Closely examining the region around $\mu_x = 4.5$ in the inset of Figure 4.3, it is clear that a slight move to the right causes the bias to move slightly negative – an *underestimation* of the cloud's distance from the center of the anode. Moving further to the right, Channel 2 begins to turn off, erasing the leftward bias that is inherent to the algorithm and causing a larger rightward asymmetry. This leads to a positive $B(\hat{x})$. By $\mu_x = 5$, however, the loss of Channel 2 is balanced by the fact that Channel 8 still hasn't turned on, the cloud is symmetrically positioned on the anode, and $\hat{x}$ is unbiased. The region $4 < \mu_x < 4.5$ has exactly the opposite trend, and the pattern of $4 < \mu_x < 5$ is repeated across the other 4 interior strips.

Finally, at any position where a Channel "turns on" or "turns off," there's a corresponding increase in the standard deviation of both estimators. This is because when the pulse height is equal to the threshold, 22, the additive readout noise with standard deviation $\sigma_z = 1$ is solely responsible for whether the pulse will be included or excluded in the Data Processing FPGA's estimates. This creates a roughly bimodal distribution in the error which is not perfectly represented in the plots (which show a $\pm s$ range around the mean).

In summary, this test demonstrates both the Data Processing FPGA's excellent performance in the central region of the anode – achieving an implied 4.1 megapixel resolution and satisfactory performance in $\widehat{w_x^2}$ – and the need for investigations into a correction for the high bias induced by edge effects.

4.4 Image Test

This test evaluates the system-level performance of the Data Processing FPGA by providing a more heterogeneous stimulus than the previous test. This comes via an image of the Washington University in St. Louis monogram, Figure 4.5. The position of each pixel in the logo is used as the (μ_x, μ_y) center coordinate of a *single* input electron cloud. This ensures that the recovered image will visually convey the performance of the design – an unsatisfactory photon efficiency would lead to a sparse, unrecognizable version of the input image.

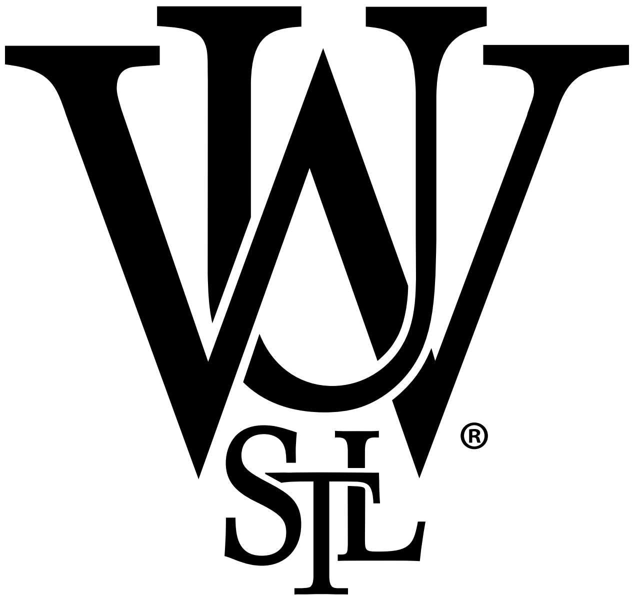

Figure 4.5: Washington University in St. Louis Monogram. In this test, a downsampled version (Figure 4.7) of this 1.2 megapixel monogram [42] is provided as stimulus to the Data Processing FPGA.

The time of arrival for each pixel's cloud is randomly assigned subdividing a total acquisition window into N events. This creates a continuous stream of input clouds with widely varying temporal spacing. In the previous test, care was taken to ensure that each input cloud was processed so that an identical number of samples could be taken at each position. For this test, the goal is to challenge the system with a stochastic, complex stimulus that more closely resembles the light generated by an actual biological sample (e.g., the spontaneous emission of a collection of fluorophores).

Section 4.4.1 explains the method by which the cloud's position was varied and defines the relevant parameters. Section 4.4.2 presents the results and a detailed discussion of the Data Processing FPGA's performance in this test.

4.4.1 Test Operation and Parameters

The Image Test provides a version of the Washington University in St. Louis monogram (Figure 4.5) as stimulus to the Data Processing FPGA. A Python application stores the monogram into an array using Numpy's `asarray` method [43]. It then downsamples the monogram by taking every 8'th row and every 8'th column of the original array, reducing the new array size to 138×138. This reduces the number of "black" pixels in the array, and therefore clouds in the test, to $C = 4198$ – an amount that can be stored in the Virtex-6's onboard RAM. The original image size featured so many pixels that the combined size of the Pulse RAMs and Time RAM far exceeded the Virtex-6's capacity of 14,976 Kb [33].

Starting from the top-left corner of the image, each black pixel in the downsampled array is translated into a (μ_x, μ_y) position for a single input cloud and stored in a list. This position is assigned such that the pixels stay almost completely within the low-bias region in the middle of the anode, $2 < \mu_x < 7$ and $2 < \mu_y < 7$. Because of whitespace on the edge of the monogram image, the required scaling in each dimension of the image is $s = 70\%$ of the width of the anode, so that the total area covered by the image is $s^2 = 49\%$. Scaling the 138×138 array to 70% of the anode in each dimension results in a pixel pitch of $(9 \cdot 0.7)/138 = 0.046$ strips/pixel or 11.6 µm/pixel. The list of (μ_x, μ_y) positions is then shuffled to randomize the pattern of cloud arrivals. This ensures that sequential clouds will arrive in diverse spatial arrangements: practically on top of each other, separated but sharing significant overlap, and well separated.

Next, the application assigns a center time μ_t to each cloud. The fundamental parameter used for this step is the desired average temporal spacing between clouds, $\overline{\Delta t} = 1500\,\text{ns}$. Recall that a cloud's local deadtime during which each FADC Channel transmits the pulse message is 960 ns. Recall also from Section 4.3.2 that each cloud with $w_{xy}^2 = 1$ and $A = 3000$ causes 5 or 6 FADC Channels in each dimension to trigger and send a message. Because the test anode is only 10 strips wide, each cloud will cause the majority of the Channels to trigger and, for the next $\sim 960\,\text{ns}$, any arriving clouds will be either entirely or partially dropped. This motivates setting $\overline{\Delta t}$ significantly above 960 ns to avoid loss of detection. Each of the 4198 μ_t center times is therefore a sample of the uniform distribution $U(0, N \cdot \overline{\Delta t}) = U(0, 4198 \cdot 1500) = U(0, 6297000)$.

The list of μ_t center times is then sorted smallest-first and associated with the list of (μ_x, μ_y) positions. The resultant list of (μ_x, μ_y, μ_t) cloud centers is passed to the Python function `generate_pulse_arrays` of Section 4.2, which generates the pulse data and `start_time` for each cloud. A logistical check is then performed on the cloud data: any cloud with a `start_time` value which is within 24 ns of the previous cloud's `start_time` must be removed. This is because it takes 3 clocks for the Controller of Figure 4.2 to check the output of the Time RAM and conditionally enable the FADC Channels. Each clock is 8 ns, so this process takes 24 ns. If a new cloud was scheduled to start during that time, it will be "missed" by the Controller and the test will essentially hang – the Controller's timer will have already passed the next specified `start_time`. This is merely a logistical correction and does not effect the test results, as the dropped cloud would have been rejected by the busy FADC Channels even if it was included.

The completed lists of `start_times` and pulse data are then written into files and, via the Xilinx tools, loaded into the Virtex-6 RAMs as described in Section 4.2. The parameters used above are summarized in Table 4.3.

Table 4.3: Image Test Parameters.

Parameter	Value
A	3000
(μ_x, μ_y)	(varies, varies)
w_{xy}^2	1 strip^2
μ_t	varies
w_t^2	3 ns^2
σ_z^2	1 count^2
$\overline{\Delta t}$	1500 ns
C	4198 clouds

4.4.2 Results and Discussion

The following discussion first examines the stability of the Data Processing FPGA in this test by examining the temporal spacing, or time deltas, between the input and recovered clouds. It then moves into a more holistic discussion of the performance by examining the reconstructed image and presenting statistics of the estimators.

Time Deltas

The ability of the Data Processing FPGA to handle the $\overline{\Delta t} = 1500\,\text{ns}$ average input cloud rate can be directly observed by plotting histograms of the time deltas between consecutive clouds in both the input and recovered data sets as in Figure 4.6. The histogram follows an exponential distribution and is thus linear when plotted on a log scale.

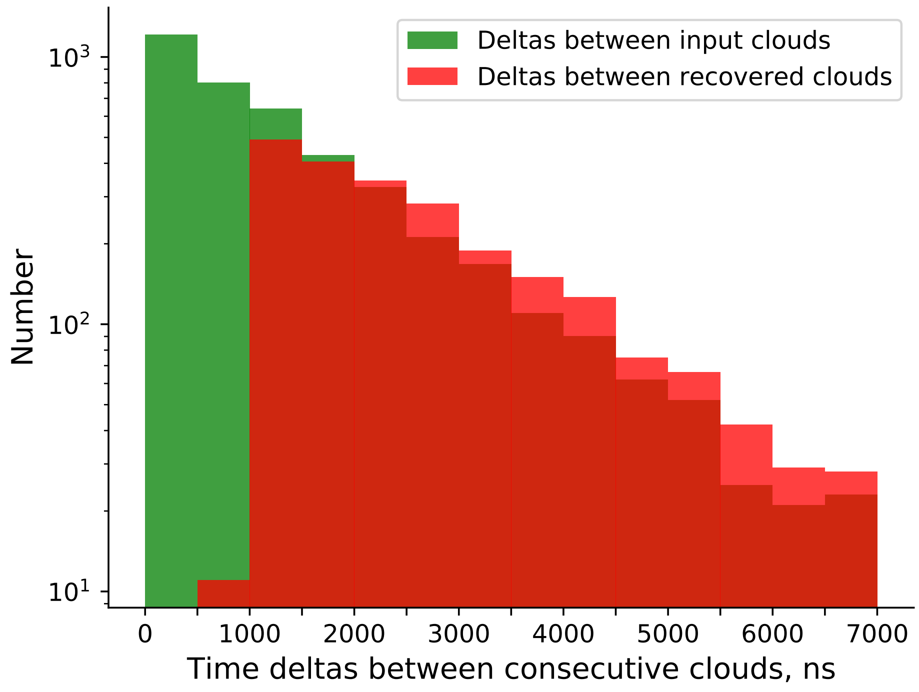

Figure 4.6: Time Deltas in Image Test. The center times μ_t for each input cloud are samples from a uniform distribution. The histogram of the time deltas between individual clouds follows an exponential distribution and is thus linear when plotted on a log scale. The Data Processing FPGA recovers almost no clouds with a delta below 1000 ns. This is a result of the 960 ns FADC Channel message duration – any clouds that arrive while a Channel is sending a message are dropped. The rejection of each short delta cloud results in longer deltas, hence the heightened bin heights on the far right side of the spectrum.

The most obvious feature of Figure 4.6 is that almost all input clouds with a Δt less than 1000 ns from the previous cloud are *not* recovered by the Data Processing FPGA. This is due to the FADC Channel's 960 ns message length and is expected. The ~ 10 remaining deltas in the 500-1000 bin were therefore just over the 960 ns minimum.

The linearity of the recovered histogram beyond the minimum delta is another key feature, as any significant variation from a linear trend indicates an artificial "dead space" that would almost certainly be the result of a flaw in the Data Processing FPGA's logic. The absence of any gaps in the recovered histogram is therefore validation that the various pieces of control logic and the time-wrapping scheme chosen were capable of handling the stochastic stimulus provided in this test.

The presence of taller bins in the recovered histogram beyond $\Delta t = 2000$ and shorter bins in the range $1000 < \Delta t < 2000$ is not an indication of system malfunction. Rather, they are a natural consequence of the dropping of clouds with $\Delta t < 960\,\text{ns}$. For example, assume the minimum Δt is 3 for brevity. Any clouds with $\Delta t < 3\,\text{ns}$ will be dropped. Then, Table 4.4 gives an example of a situation in which an acceptable delta would be "lost" and a longer delta would be "created" by the rejection of unrecoverable short-delta clouds.

Table 4.4: Effect of cloud detection loss on measured time deltas for 5 example clouds

Input Cloud Times	1	4	5	6	10
Input Deltas		3	1	1	4
$\hat{\mu}_t$	1	4			10
Recovered Deltas		3			6

The rejection of clouds at times 5 ns and 6 ns results in the lengthening of the final delta from 4 ns to 6 ns. A histogram of this data would therefore be shifted to the right, adding height to bins with higher Δt – just as observed on a large scale in Figure 4.6. Although the final input delta of 4 ns was above the minimum, it was absorbed into a longer delta – this explains the slightly shorter bins in the range $1000 < \Delta t < 2000$.

Crucially, the sum of all deltas in both the input and recovered sets of Table 4.4 is the same: 9 ns. Observing the sums of the deltas of Figure 4.6: the input set sums to 6 293 081 ns, and the recovered set sums to 6 292 811 ns. The difference of 270 ns is easily explained: the last cloud in the test is 270 ns after the second-to-last and is therefore dropped. This verifies that the Data Processing FPGA continued functioning exactly as expected for the entire 6.29 ms test, demonstrating complete system stability that is essential for extended imaging sessions.

Image Comparison

A side-by-side view of the input image and the recovered image is given in Figure 4.7. The Data Processing FPGA recovers 2288 of the 4198 input clouds for a photon detection efficiency of 54.5%. This is acceptable and even expected, as 46.5% of the input clouds have a time delta less than the FADC's message length of 960 ns. Practically, this has the effect of reducing the definition of delicate features in the input image, thereby effectively decreasing the resolution of the reconstructed image. For example, the recovered inset image of Figure 4.7 follows the general trend of the input but appears much less defined.

This is expected when considering that almost half of the photons in the input image were lost due to the dead time of the FADC Channels. This blurring is also visible by zooming in on the intersection of the "T" and "L". The white space between the serif of the T and the middle section of the L is, due to the specific arrangement of errors in the position determination, completely lost. There is only a single pixel of separation between these two shapes, meaning that the grid size implied by the combination of downsampling and scaling above (0.046 strips/pixel = 11.6 µm/pixel) was approximately the resolution limit of the system!

The radius of each of the green circles on the recovered image is the localization precision of the system across the recovered clouds; that is, the worse of the two standard deviations of the errors in $\hat{x}$ and $\hat{y}$. These two values and the other characteristics of the spatial estimators are presented in Table 4.5. Table 4.5 uses slightly different terminology than the Position

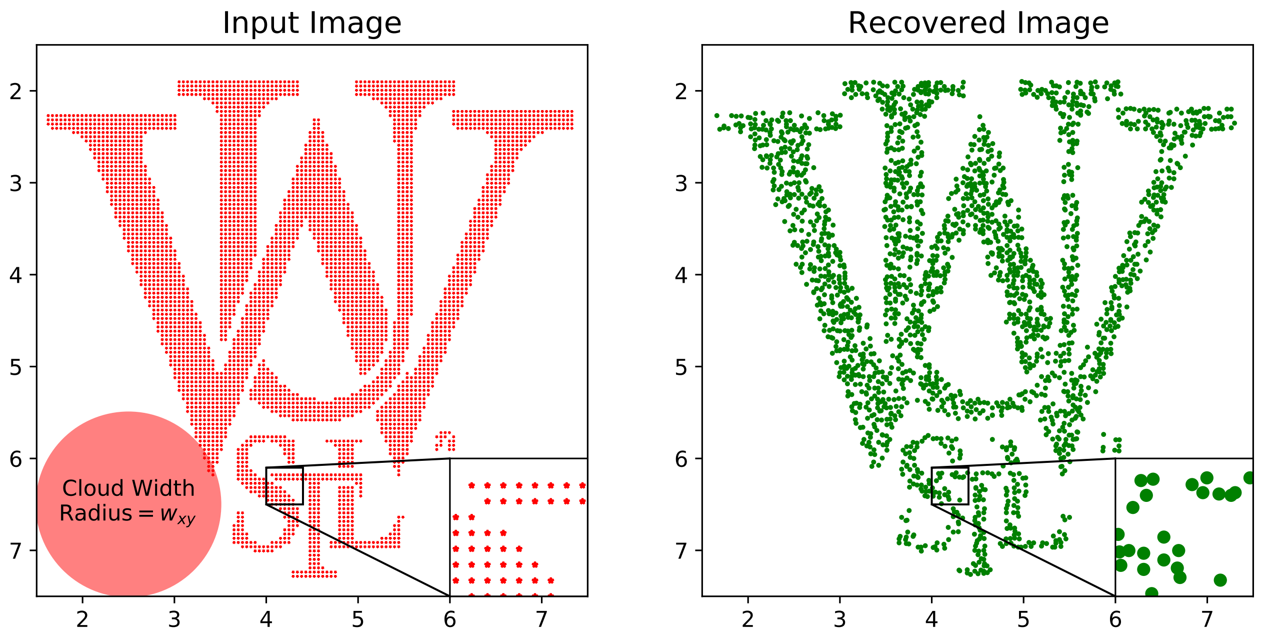

Figure 4.7: Input and Recovered Images. The input image (in red) is a downsampled version of Figure 4.5. The pitch of the input pixels is 0.046 strips/pixel, or 11.6 µm/pixel. The exact center of each pixel is provided as the center position (μ_x, μ_y) of a single input cloud. Each input cloud has a spatial standard deviation $w_{xy} = 1$ strip. The Data Processing FPGA recovers 54.5% of the clouds in the input image. The insets show the adverse effect that this cloud loss has on the recovery of fine details in the input. Each recovered position is plotted as a circle with radius equal to $s(\hat{y}) = .016$ strips (4.1 µm). See Section 4.4.2 for a detailed discussion of these results.

Scan Test's Table 4.2. Here, the bias B is computed as the average of the error at each recovered cloud location. The standard deviation of the error $s(e)$ is simply the application of Equation 2.18 to the errors at each recovered cloud location.

Table 4.5: Image Test Spatial Results.

Estimator	Bias B	Standard Deviation of Errors $s(e)$
$\hat{x}$	-0.001 strips	0.014 strips
$\widehat{w_x^2}$	-0.016 strips^2	0.04 strips^2
$\hat{y}$	-0.003 strips	0.016 strips
$\widehat{w_y^2}$	-0.006 strips^2	0.042 strips^2

The performance of the position estimates $\hat{x}$ and $\hat{y}$ was excellent, with the Y dimension's estimates faring only slightly worse than the X. The exposed area of the Y dimension of the cross strip anode is slightly smaller than the X dimension's, measurably decreasing SNR and resulting in slightly worse performance. Figure 4.3 suggests that the bias in the middle section of the anode is strip-periodic and averages to zero, and these results are a significant 2-dimensional confirmation that there are no cross-correlated biases in the position centroid algorithm – both position estimators have essentially zero average bias.

The cloud spatial variance (or width) estimates $\widehat{w_x^2}$ and $\widehat{w_y^2}$ also performed well, with a surprisingly low standard deviation of errors of 0.04 strips^2. The bias across both dimensions also closely matched the results in Table 4.2. These results must be given a caveat: they were achieved by rejecting clouds with outlying variances in a post-processing Python application. Recall that in the future, the Data Processing FPGA will drop clouds that have a variance outside of a parameterized range. The primary rationale for this feature is to reject simultaneous, adjacent cloud arrivals that appear (to the Data Processing FPGAs) to be a single cloud.

Although the test setup used here is not capable of providing two simultaneous clouds to the FADC Channels, combinations of clouds and subsets of clouds are, in fact, presented to the Data Processing FPGA in this test: recall that the FADC Channels repeatedly send an 8-bit framing word `8Ch` when not transmitting a message, and can only begin transmitting a message after completing an entire 8-bit framing word. Because the link speed is 125 Mbps, each framing word takes 64 ns. Therefore, there is up to a 64 ns delay from from the time the Controller asserts `enable` to when the FADC Channel actually begins transmission.

To see why this allows for combinations of clouds to be presented as a single cloud, consider the following: assume that a cloud arrives and triggers Channels 1-5, but each Channel must wait 64 ns before beginning a new message because they just began a new framing word. A second cloud arrives 32 ns later and causes a trigger on Channels 4-8, but only Channels 6-8 are "empty" because of the previous cloud. Channels 6-8 therefore wait 32 ns and then begin transmission. Thus, Channels *1-8* all begin transmission simultaneously and are processed by the Data Processing FPGA as a single cloud!

An even more common situation occurs when only a subset of a cloud's pulses are transmitted. The arrangement of clouds that cause this can be seen by assuming that the `start_time` of the second cloud of the previous paragraph is now 800 ns later than the first. Channels 6-8 immediately send their recovered pulses, but Channels 4 & 5 are still busy with the first message. Because 3 is the parameterized minimum number of channels in a cloud (`MIN` of Section 3.5.1), the "cloud" consisting of Channels 6-8 is accepted and assigned to the Centroiding Computation block.

The case of simultaneous transmissions of separate, overlapping clouds results in a width far exceeding the normal range. A subset transmission, on the other hand, results in width estimates that are far below the normal range. Because the second cloud could've arrived at any point during the transmission of the first, the subset situation is indeed much more common, as shown in Figure 4.8.

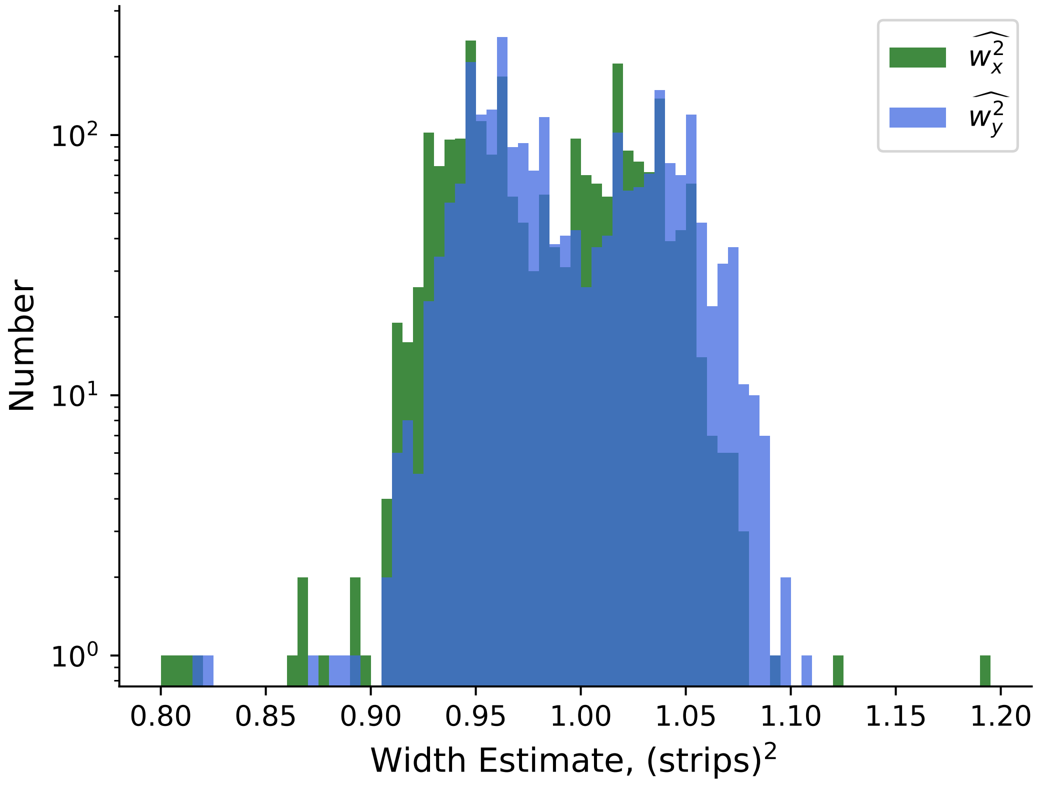

Figure 4.8: Image Test Width Estimates. The presence of outliers in the recovered width estimates in both dimensions motivates the rejection of clouds with a width estimate outside of the range $[0.9, 1.1]$. These outliers are the result of the combinations of clouds or subsets of clouds being presented to the Data Processing FPGA as a single cloud. Since the Data Processing FPGA is currently not configured to reject clouds on the basis of variance, the clouds corresponding to the outliers in this histogram were trimmed in post-processing.

The variance estimates on either extreme of Figure 4.8 are due to erroneous transmission of either combinations of clouds or subsets of clouds, but in either case they result in massive position errors. If these "clouds" were included, the recovered image of Figure 4.7 would have speckled erroneous clouds scattered throughout its whitespace and the standard deviations of $\hat{x}$ and $\hat{y}$ in Table 4.5 drastically increase. A software application therefore parses the list of recovered estimates and removes clouds which have a width estimate outside of the range

$[0.9, 1.1]$ in either dimension. This application therefore accomplishes the future function of the variance discrimination in state "Variance" of Figure 3.12.

Finally, the performance of the temporal estimators is presented in Table 4.6.

Table 4.6: Image Test Temporal Results.

Estimator	Bias B	Standard Deviation of Errors $s(e)$
$\hat{t}_x$	$-25.5\,\text{ps}$	$59.4\,\text{ps}$
$\hat{t}_y$	$-26.4\,\text{ps}$	$61.9\,\text{ps}$
$\hat{t} = (\hat{t}_x + \hat{t}_y)/2$	$-25.9\,\text{ps}$	$55.9\,\text{ps}$

The temporal estimates of each individual dimension have a lower precision than their combined average. This combination achieves a precision of $s = 55.9\,\text{ps}$. The ADC's sampling rate is 500 MHz, meaning that the temporal precision is $2\,\text{ns}/55.9\,\text{ps} = 35.8$ times smaller than the sampling rate! This result is impressive considering the somewhat artificial means by which the test software and Mock FADC Crate approximated the function of the real MCP-PMT system. For example, the software model of Section 4.2 attempted to place the center of the pulse in the middle of the $22\,\text{ns}$ "window", but was constrained to moving the window in chunks of $8\,\text{ns}$. In the real system, the FADC Channel FPGA can be far more intelligent and place the detected peak time in the middle of the 6'th and 7'th samples. This would drastically reduce asymmetry in the time sampling, lowering the bias and standard deviation of the $\hat{t}$ estimator.

The excellent performance of all three estimators in both dimensions, even under the challenging stochastic stimulus provided in this test, is a resounding validation of every step of the Data Processing FPGA. Essential to the high performance of the estimators was the proper handling of the complexities of math on an FPGA including: a diverse array of fixed point numbers, very wide operands in both MAC blocks and Divider modules, 3-stage multipliers, etc. On the other hand, the high-level design choices – like adding a minimum number of channels per cloud, adding result headers within the Ethernet packets, and using the programmable empty flags on the Xilinx FIFOs – led to the stability of the design when facing such a complex set of inputs.

Chapter 5

Outlook

The Data Processing FPGA has now been fully validated as a functioning component within a larger imaging system. By creating a test environment featuring a mix of software and hardware components that imitates that unfinished MCP-PMT's operation, a massive amount of debugging and iterative improvements have already been accomplished. This leaves the future open to perfecting the recipe of parameters, control logic, and software visualization that will comprise the final product. On a most basic level, the performance of the Data Processing FPGA (and the FADC Channel FPGAs) depends on a full characterization of the MCP-PMT to determine parameters like the minimum number of channels in a cloud, the acceptable spread of spatial variances, the pulse width, the noise levels, and the self-triggering threshold.

With the physical system characterized, the FADC Channel FPGA design and the Data Processing FPGA will have to be optimized to match the physical system. For example, if the longest pulse from a cloud is no more than 15 ns, the serial protocol should be revised to include only 7 or 8 samples, and the FADC Channel FPGA should place the pulse peak as close to the center as possible. Further optimizations could include a variable-length message – this would reduce the deadtime associated with a cloud's edge channels and allow the system to capture more spatially adjacent but temporally separated clouds. Additionally, the clock domains of each of the FADC Boards will need to be synchronized, if they aren't already – the exact phase relationship between the 125 MHz clock on each FADC board is suspected to be synchronous due to trace-length matching, but this hasn't been verified.

A minor error in the Centroiding Algorithm was discovered in the final week of the work of this thesis involving the handling of the pedestal subtraction in the Channel Receiver, Section 3.3. By subtracting the pedestal and taking the floor of each sample before the accumulation/MAC, the zero-mean Gaussian noise on the pedestal was essentially translated to a single-sided, positive mean distribution. Although the effects of this error have not been characterized, it is feasible that the error may induce a bias in the Channel Computation block's estimate of the pulse center time that would, of course, propagate to the Centroiding Computation block's $\hat{t}_x$ computation. To solve this problem, the 12 unmodified samples from the FADC Channel should first be accumulated into sum(v) and MAC'd into $v \cdot t$. Then, the accumulation of the pedestal values, $12 * 17 = 204$, and the MAC of the pedestal values, $0 * 17 + 2 * 17 + ... + 22 * 17 = 2244$, should be subtracted from sum(v) and $v \cdot t$, respectively. This fix allows the negative side of the noise distribution on the pedestal to be included in the computation and will, ultimately, reduce the bias induced by the asymmetric sampling of the pulse by the FADC Crate.

The bias induced by edge effects on the MCP-PMT's cross-strip anode should be characterized and loaded into memory on the final Data Processing FPGA. The Centroiding Computation block can then make corrections for this bias as soon as the estimate $\hat{x}$ is computed. Furthermore, if the operational form of the anode features a more exaggerated bias oscillation across the middle strips, a full-field bias correction function should be stored into memory. Each $\hat{x}$ result could therefore be used as an address into a RAM which stores the correction factor for each of the 49 strips $\cdot$ 256 positions/strip $= 12544$ resolvable positions.

Additional work is also needed to expand the physical interface of the Data Processing FPGA to 50 channels. The ML605 board has sufficient I/O for 50 channels via the XM105 FMC Cards, but the physical form factor of the board is not designed to function as part of a backplane for an imaging system. It may be necessary to design a custom layout, in which case a more modern Virtex-7 or Virtex-Ultrascale FPGA should be purchased to allow for bigger designs and obtain access to the next generation Xilinx tool – Vivado – which is much more user-friendly than its predecessor.

Finally, a full-stack software suite is needed for use by the researchers who will ultimately carry out experiments with the MCP-PMT camera. Although the `tshark` command line application used for these tests was reliable and intuitive, a more fully-featured network application should be written to parse the Ethernet bytes on the wire into data as fast as possible. Additionally, the visualization software that generated the figures used in this thesis will need to be replaced by an expansive GUI with numerous industry-standard options for viewing single-photon fluorescence images.

The new software network application should be paired with a more optimized EMAC Controller on the FPGA which performs load-balancing to minimize the bandwidth-cost of each Ethernet packet's header. This updated EMAC Controller could also feature an "RX" component which *receives* messages from the computer, allowing for real-time reconfiguration of system parameters and queries for the status of various FIFOs or computational blocks in the system.

The building blocks of hardware computation used in this project – pipelined multiply-and-accumulators and signed, modular, non-restoring dividers – were designed with minimal emphasis on pure performance. The focus was instead on modularity and simplicity, as the most significant challenge of this thesis was architecting the system. The "secret sauce" of achieving greater performance on FPGA-based systems is pipelining the datapath – for example, removing the finite state machine in the Centroiding Computation block and relying strictly on parameterized delay lines to control the computation. This reduces routing complexity, creating potential to clock the design at much higher frequencies. Unfortunately, it also has the consequence of making it more difficult for anyone but the designer to modify the code. As the underlying MCP-PMT system wasn't completed by the end of the work of this thesis and many questions regarding its characteristics remain unanswered, the exciting task of fully optimizing the Data Processing FPGA is left to the engineers and scientists who inherit the codebase.

Finally – and most excitingly to the author – the "plug-and-play" infrastructure of the Centroiding Computation blocks opens the door for research into a heterogenous, smart-compute system. Research is currently underway to develop an overparameterized gradient-descent optimization algorithm that minimizes a loss function based on the forward model of the MCP-PMT camera. This algorithm would be able to resolve multiple simultaneous cloud collisions – something the current center-of-gravity algorithm is simply not capable of. A smart compute platform could dynamically assign cloud data to a diverse array of computational solutions. The potential for bidirectional communication with the computer via the Gigabit Ethernet link means the parameters of these more complicated algorithms could be tuned by the researcher in real time. Researchers could even implement their own algorithms in Verilog and add them to the system – they simply have to obey the interface specifications of the Centroiding Computation block.

In short – the potential for new research on the MCP-PMT system is still very large. The results of the work of this thesis has only served to excite more curiosity about the performance of the eventual system, and various types of expertise will be necessary to optimize the physical MCP-PMT, the electronics of the FADC Crate and Data Processing FPGAs, and the network and software applications on the host computer.

References

[1] Andreas Gahlmann and William Moerner. Exploring bacterial cell biology with single-molecule tracking and super-resolution imaging. *Nature reviews. Microbiology*, 12:9–22, 2013. doi: 10.1038/nrmicro3154.

[2] Zhe Liu, Luke D. Lavis, and Eric Betzig. Imaging live-cell dynamics and structure at the single-molecule level. *Molecular Cell*, 58(4):644 – 659, 2015. ISSN 1097-2765. doi: https://doi.org/10.1016/j.molcel.2015.02.033. URL http://www.sciencedirect.com/science/article/pii/S1097276515001653.

[3] W. E. (William E.) Moerner. Nobel lecture: Single-molecule spectroscopy, imaging, and photocontrol: Foundations for super-resolution microscopy. *Rev. Mod. Phys.*, 87: 1183–1212, 2015. doi: 10.1103/RevModPhys.87.1183. URL https://link.aps.org/doi/10.1103/RevModPhys.87.1183.

[4] Taekjip Ha. Need for speed: Mechanical regulation of a replicative helicase. *Cell*, 129 (7):1249 – 1250, 2007. ISSN 0092-8674. doi: https://doi.org/10.1016/j.cell.2007.06.007. URL http://www.sciencedirect.com/science/article/pii/S0092867407007738.

[5] Jason Gorman, Feng Wang, Sy Redding, Aaron J. Plys, Teresa Fazio, Shalom Wind, Eric E. Alani, and Eric C. Greene. Single-molecule imaging reveals target-search mechanisms during dna mismatch repair. *Proceedings of the National Academy of Sciences*, 109(45):E3074–E3083, 2012. ISSN 0027-8424. doi: 10.1073/pnas.1211364109. URL https://www.pnas.org/content/109/45/E3074.

[6] Koh O. Nagata, Chieko Nakada, Rinshi S. Kasai, Akihiro Kusumi, and Kazumitsu Ueda. Abca1 dimer–monomer interconversion during hdl generation revealed by single-molecule imaging. *Proceedings of the National Academy of Sciences*, 110(13):5034–5039, 2013. ISSN 0027-8424. doi: 10.1073/pnas.1220703110. URL https://www.pnas.org/content/110/13/5034.

[7] Kevin Spehar, Tianben Ding, Yuanzi Sun, Niraja Kedia, Jin Lu, George Nahass, Matthew Lew, and Jan Bieschke. Super-resolution imaging of amyloid structures over extended times using transient binding of single thioflavin t molecules. *ChemBioChem*, 19, 2018. doi: 10.1002/cbic.201800352.

[8] Tianben Ding, Tingting Wu, Hesam Mazidi, Oumeng Zhang, and Matthew D. Lew. Single-molecule orientation localization microscopy for resolving structural heterogeneities

between amyloid fibrils. *Optica*, 7(6):602–607, 2020. doi: 10.1364/OPTICA.388157. URL http://www.osapublishing.org/optica/abstract.cfm?URI=optica-7-6-602.

[9] Takashi Tachikawa and Tetsuro Majima. Single-molecule fluorescence imaging of tio2 photocatalytic reactions. *Langmuir : the ACS journal of surfaces and colloids*, 25: 7791–802, 2009. doi: 10.1021/la900790f.

[10] Meikun Shen, Tianben Ding, Steven T. Hartman, Fudong Wang, Christina Krucylak, Zheyu Wang, Che Tan, Bo Yin, Rohan Mishra, Matthew D. Lew, and Bryce Sadtler. Nanoscale colocalization of fluorogenic probes reveals the role of oxygen vacancies in the photocatalytic activity of tungsten oxide nanowires. *ACS Catalysis*, 10(3):2088–2099, 2020. doi: 10.1021/acscatal.9b04481. URL https://doi.org/10.1021/acscatal.9b04481.

[11] Genevieve Gariepy, Francesco Tonolini, Robert Henderson, Jonathan Leach, and Daniele Faccio. Detection and tracking of moving objects hidden from view. *Nature Photonics*, 10(1):23–26, 2016. ISSN 1749-4893. doi: 10.1038/nphoton.2015.234. URL https://doi.org/10.1038/nphoton.2015.234.

[12] Matthew O'Toole, David B. Lindell, and Gordon Wetzstein. Confocal non-line-of-sight imaging based on the light-cone transform. *Nature*, 555(7696):338–341, 2018. ISSN 1476-4687. doi: 10.1038/nature25489. URL https://doi.org/10.1038/nature25489.

[13] Klaus Suhling, Liisa M. Hirvonen, James A. Levitt, Pei-Hua Chung, Carolyn Tregidgo, Alix Le Marois, Dmitri A. Rusakov, Kaiyu Zheng, Simon Ameer-Beg, Simon Poland, Simao Coelho, Robert Henderson, and Nikola Krstajic. Fluorescence lifetime imaging (flim): Basic concepts and some recent developments. *Medical Photonics*, 27:3 – 40, 2015. ISSN 2213-8846. doi: https://doi.org/10.1016/j.medpho.2014.12.001. URL http://www.sciencedirect.com/science/article/pii/S2213884614000033.

[14] Simon P. Poland, Nikola Krstajić, James Monypenny, Simao Coelho, David Tyndall, Richard J. Walker, Viviane Devauges, Justin Richardson, Neale Dutton, Paul Barber, David Day-Uei Li, Klaus Suhling, Tony Ng, Robert K. Henderson, and Simon M. Ameer-Beg. A high speed multifocal multiphoton fluorescence lifetime imaging microscope for live-cell fret imaging. *Biomed. Opt. Express*, 6(2):277–296, 2015. doi: 10.1364/BOE.6.000277. URL http://www.osapublishing.org/boe/abstract.cfm?URI=boe-6-2-277.

[15] Wolfgang Becker, Liisa M. Hirvonen, James Milnes, Thomas Conneely, Ottmar Jagutzki, Holger Netz, Stefan Smietana, and Klaus Suhling. A wide-field tcspc flim system based on an mcp pmt with a delay-line anode. *Review of Scientific Instruments*, 87(9):093710, 2016. doi: 10.1063/1.4962864. URL https://doi.org/10.1063/1.4962864.

[16] Rene Glazenborg, James Marr, Adrian Martin, Raquel Ortega, Emile Schyns, Oswald Siegmund, and John Vallerga. Imaging photon camera with high spatiotemporal resolution. *European Microscopy Congress 2016: Proceedings*, pages 471–472, 2016. doi: 10.1002/9783527808465.EMC2016.6905. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9783527808465.EMC2016.6905.

[17] X. Michalet, R. A. Colyer, G. Scalia, A. Ingargiola, R. Lin, J. E. Millaud, S. Weiss, Oswald H. W. Siegmund, Anton S. Tremsin, John V. Vallerga, A. Cheng, M. Levi, D. Aharoni, K. Arisaka, F. Villa, F. Guerrieri, F. Panzeri, I. Rech, A. Gulinatti, F. Zappa, M. Ghioni, and S. Cova. Development of new photon-counting detectors for single-molecule fluorescence microscopy. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 368(1611):20120035, 2013. doi: 10.1098/rstb.2012.0035. URL https://royalsocietypublishing.org/doi/abs/10.1098/rstb.2012.0035.

[18] W. H. McCarroll, R. J. Paff, and A. H. Sommer. Role of cs in the (cs)na2ksb (s-20) multialkali photocathode. *Journal of Applied Physics*, 42(2):569–572, 1971. doi: 10.1063/1.1660065. URL https://doi.org/10.1063/1.1660065.

[19] Dmitry A. Orlov, Jeff DeFazio, Serge Duarte Pinto, Rene Glazenborg, and Emilie Kernen. High quantum efficiency S-20 photocathodes in photon counting detectors. *JINST*, 11 (04):C04015, 2016. doi: 10.1088/1748-0221/11/04/C04015.

[20] J. Adams and B. W. Manley. The mechanism of channel electron multiplication. *IEEE Transactions on Nuclear Science*, 13(3):88–99, 1966.

[21] Ming Wu, Craig A. Kruschwitz, Dane V. Morgan, and Jiaming Morgan. Monte carlo simulations of microchannel plate detectors. i. steady-state voltage bias results. *Review of Scientific Instruments*, 79(7):073104, 2008. doi: 10.1063/1.2949119. URL https://doi.org/10.1063/1.2949119.

[22] C. D. Ertley, O. H. W. Siegmund, J. Hull, A. Tremsin, A. O'Mahony, C. A. Craven, and M. J. Minot. Microchannel plate imaging detectors for high dynamic range applications. *IEEE Transactions on Nuclear Science*, 64(7):1774–1780, 2017.

[23] M Akatsu, Y Enari, K Hayasaka, T Hokuue, T Iijima, K Inami, K Itoh, Y Kawakami, N Kishimoto, T Kubota, M Kojima, Y Kozakai, Y Kuriyama, T Matsuishi, Y Miyabayashi, T Ohshima, N Sato, K Senyo, A Sugi, S Tokuda, M Tomita, H Yanase, and S Yoshino. Mcp-pmt timing property for single photons. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 528(3):763 – 775, 2004. ISSN 0168-9002. doi: https://doi.org/10.1016/j.nima.2004.04.207. URL http://www.sciencedirect.com/science/article/pii/S0168900204009817.

[24] Eric W. Weisstein. Gaussian function. From MathWorld—A Wolfram Web Resource, 2002. URL https://mathworld.wolfram.com/GaussianFunction.html. Last visited on 5/25/2020.

[25] James H. Buckley, Paul F. Dowkontt, Karl Peter Kosack, and P. F. Rebillot. The veritas flash adc electronics system. In *International Cosmic Ray Conference*, volume 5 of *International Cosmic Ray Conference*, page 2827, 2003.

[26] John Vallerga, Anton Tremsin, Rick Raffanti, and Oswald Siegmund. Centroiding algorithms for high speed crossed strip readout of microchannel plate detectors. *Nuclear instruments & methods in physics research. Section A, Accelerators, spectrometers, detectors and associated equipment*, 633(S1):S255–S258, 2011. ISSN 0168-9002. doi: 10.1016/j.nima.2010.06.181. URL https://pubmed.ncbi.nlm.nih.gov/21918588. 21918588[pmid].

[27] Gregorio Landi. Properties of the center of gravity as an algorithm for position measurements. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 485(3):698 – 719, 2002. ISSN 0168-9002. doi: https://doi.org/10.1016/S0168-9002(01)02071-X. URL http://www.sciencedirect.com/science/article/pii/S016890020102071X.

[28] E. T. Jaynes. *Probability Theory: The Logic of Science*. Cambridge University Press, 2003. doi: 10.1017/CBO9780511790423.

[29] *UG537 FMC XM105 Debug Card User Guide*. Xilinx, 2011. URL https://www.xilinx.com/support/documentation/boards_and_kits/ug537.pdf. v1.3.

[30] *UG534 ML605 Hardware User Guide*. Xilinx, 2019. URL https://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf. v1.9.

[31] *AMBA 4 AXI4-Stream Protocol Specification*. ARM, 2010. URL https://static.docs.arm.com/ihi0051/a/IHI0051A_amba4_axi4_stream_v1_0_protocol_spec.pdf. v1.0.

[32] *MV-S105540-00 88E1111 Product Brief*. Marvell, 2013. URL https://www.marvell.com/content/dam/marvell/en/public-collateral/phys-transceivers/marvell-phys-transceivers-alaska-88e1111-technical-product-brief-2013-10.pdf. Rev. A.

[33] *DS150 Virtex-6 Family Overview*. Xilinx, 2015. URL https://www.xilinx.com/support/documentation/data_sheets/ds150.pdf. v2.5.

[34] Randy Yates. Fixed-point arithmetic: An introduction. Technical report, Digital Signal Labs, 2013. Rev. PA8.

[35] John Goldie and Mike Wilson. Lvds performance: Bit error rate (ber) testing. Technical report, National Semiconductor, 1996. Test Report #2.

[36] *UG361 Virtex-6 FPGA SelectIO Resources*. Xilinx, 2014. URL https://www.xilinx.com/support/documentation/user_guides/ug361.pdf. v1.6.

[37] *UG362 Virtex-6 FPGA Clocking Resources*. Xilinx, 2014. URL https://www.xilinx.com/support/documentation/user_guides/ug362.pdf. v2.5.

[38] D.G. Bailey. Space efficient division on fpgas. In *Electronics New Zealand Conference (ENZCon'06)*, page 206 – 211, 2006.

[39] *LogiCORE IP Virtex-6 FPGA Embedded Tri-Mode Ethernet MAC Wrapper*. Xilinx, 2012. URL https://www.xilinx.com/support/documentation/ip_documentation/v6_emac/v2_3/ug800_v6_emac.pdf. v2.3.

[40] *UG368 Virtex-6 FPGA Embedded Tri-Mode Ethernet MAC*. Xilinx, 2011. URL https://www.xilinx.com/support/documentation/user_guides/ug368.pdf. v1.3.

[41] Gerald Combs. *tshark - Dump and analyze network traffic*. Wireshark. URL https://www.wireshark.org/docs/man-pages/tshark.html.

[42] Washington university symbols, 2020. URL https://publicaffairs.wustl.edu/resources/branding-logo-toolkit/download-logos/. Last visited on 9/16/2020.

[43] Travis E. Oliphant. *Guide to NumPy*. CreateSpace Independent Publishing Platform, 2015.

[44] Sheldon Waite and Erdal Oruklu. FPGA-Based Traffic Sign Recognition for Advanced Driver Assistance Systems. *Journal of Transportation Technologies*, Vol.03No.01:16, 2013. URL 10.4236/jtts.2013.31001.

[45] Gabriel García, Carlos Jara, Jorge Pomares, Aiman Alabdo, Lucas Poggi, and Fernando Torres. A survey on fpga-based sensor systems: Towards intelligent and reconfigurable low-power sensors for computer vision, control and signal processing. *Sensors*, 14(4): 6247–6278, 2014. ISSN 1424-8220. doi: 10.3390/s140406247. URL http://dx.doi.org/10.3390/s140406247.

Data Processing FPGA for Camera, Hyde, M.S. 2020