FPGA-Based Pedestal Variance Computation Accelerator for Continuous Light Monitoring in VERITAS

Matt Shepherd, Izabella Pastrana, Megan Lin

ESE 498/499 Capstone Design Project Final Report
Submitted to Professor Trobaugh, Professor Chakrabartty and the Department of Electrical and Systems Engineering

Group Members:
Megan Lin
Department: Electrical & Systems Engineering
B.S. Candidate in Systems Science and Engineering
lin.m.d@wustl.edu

Izabella Pastrana
Department: Electrical & Systems Engineering
B.S. Candidate in Electrical Engineering, Second Major in Physics
izabella.pastrana@wustl.edu

Matthew Shepherd
Department: Electrical & Systems Engineering
B.S. Candidate in Electrical Engineering, Computer Engineering, and Computer Science
mshepherd@wustl.edu

Project Advisors:
Dr. Shantanu Chakrabartty
Professor of Engineering, Washington University in St. Louis
shantanu@wustl.edu

Dr. James Buckley
Professor of Physics, Washington University in St. Louis
buckley@wuphys.wustl.edu

Date of Submission: April 29, 2019

Time Period: Spring Semester 2019
January 14, 2019 - May 7, 2019

0. Abstract

VERITAS is a telescope array that uses its four telescopes to detect Cherenkov light pulses, which are produced when gamma rays interact with the earth's atmosphere. It monitors the night sky for Cherenkov light pulses, but has to reject many events because the Night Sky Background crowds the lens with photons. The telescope has rejection criteria so it doesn't have to store non-events or noisy events to disk. Additionally, the rejection criteria only store short Cherenkov light pulse samples, meaning it is collecting useful data less than 20% of the time. This project aims to fix these issues by creating an FPGA-based system to accelerate variance calculations of the light signals, known as the pedestal variance. We examined three possible computation schemes: parallelized, parallelized-pipelined, and multi-level parallelized-pipelined. We found the parallelized scheme was optimal, and we tested the scheme in MATLAB, in hardware simulation, and in actual hardware. Lastly, we packaged the hardware description for easy installation in VERITAS.

1. Introduction

1.1 Background

VERITAS, the Very Energetic Radiation Imaging telescope Array System, is an array of four Imaging Cherenkov Telescopes (IACTs). These ground based telescopes are located at the Fred Lawrence Whipple Observatory in Mount Hopkins, Arizona [1]. Each of the four IACTs is a gamma-ray telescope that measures 12 meters in diameter and contains a 499-pixel imaging camera, comprised of 499 photomultiplier tubes (PMTs). Each of the PMTs observes a single pixel's light level and sends the data to a Flash Analog-to-Digital Converter (FADC) in order to be digitized. The FADC then creates 8-bit digitized samples at a rate of 500 MHz. These samples are temporarily stored in a 65 us buffer on a Field-Programmable Gate Array (FPGA) [2]. The system uses the telescopes to observe and collect Cherenkov light pulses, which are created when gamma rays interact with the earth's atmosphere. This allows VERITAS to observe electromagnetic radiation in the high-energy gamma ray regime (85 GeV to more than 30 TeV) from various astrophysical sources, like exploding stars (supernovae), black holes, and pulsars. However, the IACTs' cameras are constantly being crowded by photons from the Night Sky Background (NSB), consisting of starlight, scattered diffuse galactic light, etc. in the night sky. In order to filter out the random NSB events, VERITAS implements a three-level trigger system in order to isolate Cherenkov light pulses in real time [3]. Once the three-level trigger system's criteria is met, the FPGAs stop buffering the new Cherenkov light pulse samples to store the data onto a disk [4]. Simply put, Cherenkov light monitoring is not continuous and storing the Cherenkov light pulse data incurs some amount of deadtime for the system. VERITAS' three-level trigger system only allows for short Cherenkov light pulse samples, causing a low duty cycle of less than 20% [5].

In this project, the aforementioned problems are addressed: discontinuous light monitoring, reduction in deadtime when storing light samples, and improvement in duty cycle. The project's main goal was to design an FPGA-based system that allows for a continuous stream of light monitoring. This system will allow VERITAS to constantly collect useful data, which will reduce deadtime and improve duty cycle.

1.2 Project Objectives

Our main objective was to allow for less frequent, less noisy ambient light sampling in order to improve the duty cycle of VERITAS. We achieved this light monitoring with no deadtime is by using FPGAs to execute running calculations of what is known as *pedestal variance.*

Each PMT in the array's camera is capacitively coupled with an FADC, which means the DC level of the PMT's raw input signal, known as the *pedestal*, is blocked; only the AC component of the raw input signals (from photon contributions from the NSB as well as electronic noise) propagate to the FADC and three-level trigger logic. However, these fluctuations of the AC component may dip below zero, while the FADCs can only measure positive values. To circumvent this issue, VERITAS engineers added an artificial pedestal (DC voltage) to the AC component so that the resultant signal was entirely positive. The total light level (including the NSB) is proportional to the average number of photons detected per sample and the average number of photons detected is small (photon arrival probability distribution resembles a Poisson distribution), so we know that the variance of the digitized signal, know as the *pedestal variance*, is ultimately proportional to the light level sensed by each pixel [6]. Thus, if we can measure the pedestal variance at a lower sampling frequency than the FADC, we can continuously measure the light level without overloading ourselves with data! Since pedestal variance is proportional to light level, continuously calculating the pixel pedestal variance is effectively continuously monitor the light level being sensed by that pixel. This would also detect all the light hitting the camera, not just Cherenkov pulses, as in the existing system.

Our secondary objective is that we want to improve the sensitivity of the normal gamma ray detection of VERITAS, which we can achieve quite easily by calculating the pedestal variance at a low frequency. As mentioned before, the pedestal variance exists is proportional to the changing NSB light, particularly light in the blue/UV spectrum. Blue/UV light is used in the three-level trigger logic to discern between Cherenkov pulses and NSB events, so better monitoring the pedestal variance could improve VERITAS monitoring at the trigger level [7]. When processing events recorded by the trigger logic, any pixel which produces a signal-to-noise ratio (SNR) above 5 is called a signal pixel. This means that the measured signal of photoelectrons on a pixel is at least 5 times greater than the pedestal variance (the noise) of that pixel. Pixels with SNR between 2.5 and 5 are only considered signals if their neighbors are signal pixels with SNR greater than 5. Pixels that do not satisfy signal conditions are considered background pixels. Thus, with fast calculations of pedestal variance, we could refine the trigger rejection criteria, ultimately improving event detection in VERITAS [8].

With these objectives in mind, we decided the goal of this project was to develop an FPGA hardware implementation that calculates the running pedestal variance by processing samples at the FADCs' 500 MHz sample rate. This is very feasible, considering that VERITAS already uses MicroSemi ProASIC3 FPGAs to process samples for their three-level trigger logic and they have enough room to fit a small module for calculating pedestal variance [9]. However, we do not have access to these FPGAs, so we developed the hardware using Verilog with a Nexys 4 DDR FGPA. Our idea was that we would design the module to work on the Nexys 4 DDR, and then we can hand off the Verilog files to our advisor Dr. James Buckley so he and his team can implement it on the Actel FPGAs that run VERITAS. We will discuss this further in our Deliverables section.

2. Methods

2.1 Project Requirements

In order to have our project be successful, we need it to satisfy the basic requirements, as well as optimize for the three main measures of hardware effectiveness: the power consumption, the logic size/area, and the clock speed.

As far as basic requirements go, we have only two main requirements: we have to calculate the pedestal variance, and we have to process each sample before the next sample arrives. In order to calculate the pedestal variance, we will compute the sum of all samples and the sum of all squared samples. Statistically, we are calculating and storing $E[X]*N$ and $E[X^2]*N$ where N is the fixed number of samples in a variance calculation window, so we can calculate $\sigma^2 = E[X^2] - (E[X])^2$ after the fact from what we store. Additionally, we will be receiving a single 32-bit data word consisting of four 8-bit Gray encoded FADC samples every 8 ns (125 MHz frequency), which is the same as one sample every 2 ns (500 MHz frequency). The hardware we design must be able to keep up with this data stream at a bare minimum.

In addition, we should optimize our hardware for power consumption, logic area, and clock speed. From our correspondence with Richard Bose, an engineer working on VERITAS, we know that we should first optimize for our logic area before we optimize for clock speed and power consumption. This is because the existing hardware uses up about 27% of the available logic gates on their Actel FPGAs, and if we should try to keep our hardware confined to the remaining 73% available [9]. If we cannot, however, then the VERITAS team will try to budget FPGA upgrades that can fit all the required hardware. On the other hand, if we can fit our logic into less than the full 73% remaining on their chips, then it will be possible to add even more data processing in the future, if the VERITAS team so chooses. As a secondary “preferred” goal, we want to use less that 80% of the area in total (including the existing 27%), because loading the FPGA by less than the maximum allows the team to add more functionality in the future if they like, but also because FPGAs have finite routing resources, and thus we may experience a performance dip (and thus, practical failure) if we were to load the FPGA maximally [10]. Therefore, we *should* try to use less than 53% of the total logical area, but we can probably violate this requirement without issue.

In terms of optimizing for clock speed or power consumption, there is no real need to optimize timing further than the 4-sample 125-MHz clock that is a basic requirement [9]. If we were to achieve a higher clock speed, it would be wasted, as we cannot change our sample processing stream. Additionally, the cost of powering the FPGAs is a percent of what it costs to power the PMTs and FADCs, and any hardware we design is unlikely to make a dent in that [9]. However, any improvement to power consumption will save money in the long term, so there is some benefit to optimizing for power (in contrast to optimizing for clock speed).

With these requirements in mind, we will now go into our current methods for achieving our objectives.

2.2 Gray Code

As mentioned in Section 2.1, the FADCs of the VERITAS system output the digitized light level (samples) in 8-bit Gray code format to the FPGAs, where the 8-bit number represents a measured voltage with units of digital counts. Gray code, named after Frank Gray, refers to an ordering of the binary numeral system such that only one bit differs between successive values. This ordering is most useful when values change rapidly (as in the case of a 500 MHz sampling rate) as it helps avoid errors due to the hardware--electromechancial switches in circuitry do not necessarily switch simultaneously--and interfacing constraints. While it is possible to perform Gray code addition [10], it is much simpler for us (and those onto whom we will pass our project) to perform our pedestal variance calculations in standard binary format. Either way, we must develop a fast Gray code to binary conversion scheme that will convert the 8-bit Gray code data to standard binary with which our computation scheme may actually work.

The common conversion algorithm is a linear algorithm, which is to say every additional bit requires an additional clock tick. Fortunately for us, there is a well known algorithm for performing such a conversion in logarithmic time, which means it will only cost us one additional clock tick to double the size of our Gray code conversion inputs [11]. This is also an improvement on the current implementation of Gray code conversion on the existing VERITAS FPGAs, so we can recommend this change to Richard Bose for the future.

2.3 Computation Schema

In order to process our samples sufficiently quickly, we have designed three possible computation schemes: a parallelized schema, a parallelized-pipelined schema, and a multi-level parallelized-pipelined schema.

2.3.1 Parallelized

Here is a diagram of our parallelized computation schema:

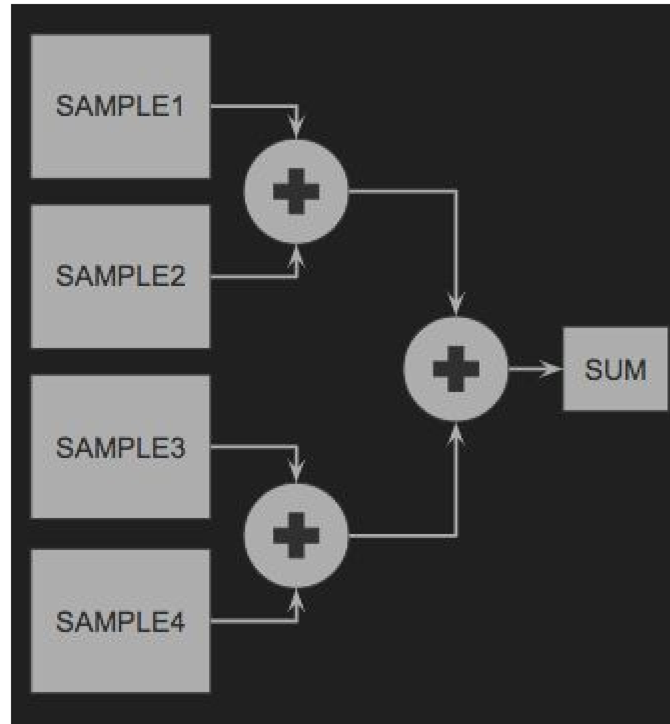

Figure 2.3.1 - Parallelized Computation Schema

As each set of four samples comes in, we will simply add each of them in the adder tree depicted above. This implementation is the slowest of the three, where each minimum clock period (assuming we implement Ripple-Carry-Adders (RCA)) must be (8-bit RCA) + (9-bit RCA) = 17-bit RCA = 17 gate delays. For the squared samples, this would be a (16-bit RCA) + (17-bit RCA) = 33-bit RCA = 33 gate delays. This is likely sufficient, as a typical gate delay is less than 200 ps, which would mean our minimum clock speed is 6.6 ns to process all the samples (less than 8 ns).

2.3.2 Parallelized-Pipelined

Here is a diagram of our parallelized-pipelined computation schema:

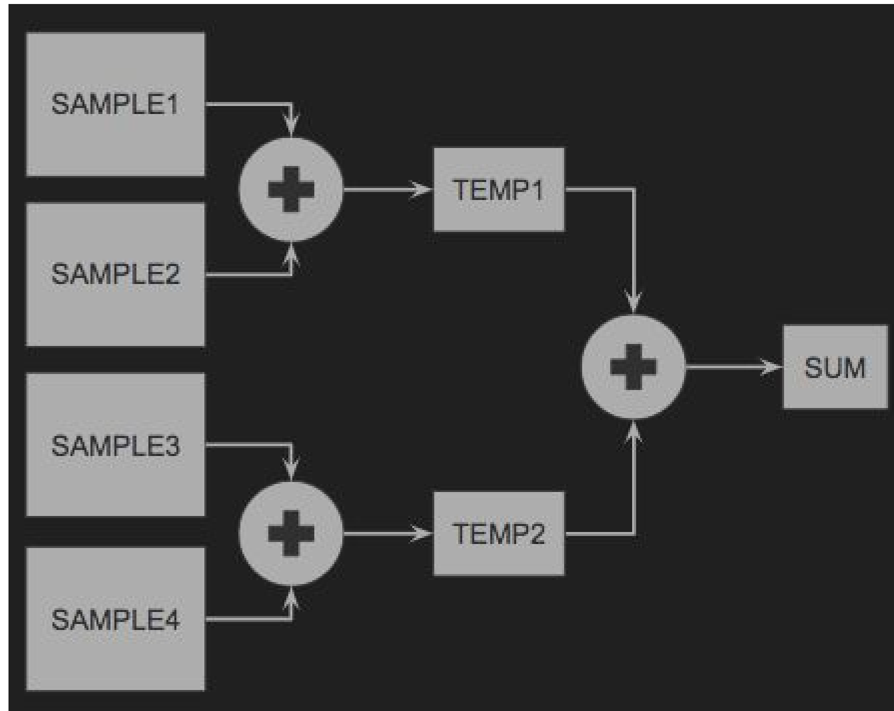

Figure 2.3.2 - Parallelized-Pipelined Computation Schema

As each set of four samples comes in, we add them in pairs and store the pairs in temporary registers (as shown). This allows us to speed up our clock so that it is limited by only the slowest stage of our pipeline. In this case, the minimum clock period (assuming we implement RCAs) is 9-bit RCA = 9 gate delays. For the squared samples, this would be a 17-bit RCA = 17 gate delays. By similar reasoning as the parallelized schema, we would expect our minimum clock speed to be 3.4ns to process all the samples. This approach, however, requires more power and area because we have to add the temporary registers. If the parallelized schema is fast enough, then it would be better to use it instead because we want to optimize power and area rather than clock speed.

2.3.3 Multi-level Parallelized-Pipelined

We have no diagram for our multi-level parallelized-pipelined schema because this schema can be parameterized. Essentially, our approach is the same as in the parallelized-pipelined, but in addition to pipelining our adder tree, we can also pipeline our RCAs. Depending on how much faster we want our clock to go, we can pipeline every n-th bit (for some $1 \le n \le k$, where k is the number of bits in the adder) so that we can have a minimum clock of n gate delays = 200n ps. This is consistent for both samples and squared samples.

This approach requires the most power and area out of the three because we have to add the temporary registers not just in the adder tree, but also inside the RCAs. If either of the other two schemas are fast enough, then we should use them, because, as we mentioned before, we want to optimize power and area rather than clock speed. Now that we have explained our

computation schemas, we will need to verify that they work as we intended. To do so, we wrote verification code in MATLAB.

2.4 MATLAB Verification

We needed to verify that our computation architecture outputs correct pedestal variance values. To ensure this, we tested the parallelized computation schema by implementing it in MATLAB and verifying that it worked for a test data set. The MATLAB implementation worked properly (more on how we found this in the Data Collection section), so we moved on to testing the schema in simulation.

2.5 Simulation Verification

After developing the Verilog required to simulate our simplest computation schema, we ran a simulation of the computation in Vivado. Specifically, we generated a large data set which we loaded into a data array in our testbench and then computed the sum of the samples and the sum of the squared samples We verified this data (more on this in the Data Collection section).

After verifying that our simulation accurately calculates the pedestal variance, we generated performance reports for our schema so we could evaluate the feasibility of our schema. We discuss this more in the Data Collection section. After evaluating the performance, we went on to test the implementation of our schema.

2.6 Implementation Verification

When we were testing the implementation of our computation schema, we ran into a significant issue: we did not have access to hardware that can emit samples at a rate of 500 MHz! We came up with a workaround: we would hardwire 3 out of 4 samples per dataword to logic 0 internally to the board, and then we would have an Arduino RedBoard connected to a AdaFruit SI1145 UV light sensor over I2C bus write the sensed values through its digital pins. Then, we could print the output statistics over USB and read them out on the screen. This worked okay at best, but we were able to verify our implementation.

Next, we will explain how we generated our testing data set.

2.7 Testing Data Set Generation

In order to simulate the data of Cherenkov light pulse samples, Dr. Buckley recommended that we generate a data set. In order to generate the raw light input, we created a Gaussian normal distribution. Utilizing MATLAB, a Gaussian normal distribution was created with a mean of 65, in order for the distribution to span from 0 to 130. Additionally, the standard deviation, σ, was set to 7. The five million x-value points, consisting of a randomly generated integer spanning from 0 to 130, were extracted in order to run into our schema. The numbers listed above were chosen at random, and can be arbitrarily chosen as any non-negative integer value.

3. Data Collection

Our data was collected in two parts: 1) the verification of our computation and 2) the performance estimation via implementation reports. The verification of our computation was done in Excel; we computed the correct output given the data we generated and then compared it with the output of our simulations (both MATLAB and Verilog). After verifying the functionality, we generated implementation reports using Vivado. These reports include information about the timing, power, and area utilization of our implementation. This data is displayed in Tables 3.1, 3.2, and 3.3.

```
----------------------------------------------------------------------
| Timing Details
----------------------------------------------------------------------
From Clock:  clk
  To Clock:  clk

Setup :
0 Failing Endpoints,  Worst Slack 1.367ns,  Total Violation 0.000ns
Hold  :
0 Failing Endpoints,  Worst Slack 0.052ns,  Total Violation 0.000ns
PW    :
0 Failing Endpoints,  Worst Slack 3.500ns,  Total Violation 0.000ns
----------------------------------------------------------------------
-
```

Table 3.1 *Parallelized (simple) schema timing report*

```
1. Summary
----------
+--------------------------+--------------+
| Total On-Chip Power (W)  | 0.156        |
| Design Power Budget (W)  | Unspecified* |
| Power Budget Margin (W)  | NA           |
| Dynamic (W)              | 0.065        |
| Device Static (W)        | 0.091        |
| Effective TJA (C/W)      | 4.6          |
| Max Ambient (C)          | 84.3         |
| Junction Temperature (C) | 25.7         |
| Confidence Level         | Low          |
| Setting File             | ---          |
| Simulation Activity File | ---          |
| Design Nets Matched      | NA           |
+--------------------------+--------------+
```

Table 3.2 *Parallelized (simple) schema power report*

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs	379	0	63400	0.60
LUT as Logic	379	0	63400	0.60
LUT as Memory	0	0	19000	0.00
Slice Registers	332	0	126800	0.26
Register as Flip Flop	332	0	126800	0.26
Register as Latch	0	0	126800	0.00
F7 Muxes	0	0	31700	0.00
F8 Muxes	0	0	15850	0.00

Table 3.3 *Parallelized (simple) schema utilization report*

We collected the timing and power data directly from the implementation reports from Vivado, but we had to do some additional conversions to find the expected slice utilization. One limitation of our project was that we could not generate a utilization report with utilization percents for the ProAsic3 A3P250 Device that project VERITAS uses. Thus, we had to rely on the Nexys 4 DDR utilization reports to calculate how much logic and memory we used, and then convert that over to A3P250 utilization. Table 3.1 shows the utilization report from our Nexys 4 DDR implementation.

Here we can see that we used 379 look-up tables (LUTs) and 332 flip-flops. The A3P250 has 250,000 logic gates, and 6,144 flip flops [12]. We can easily find the flip-flop utilization (332/6144 ≈ 5.4%), but to convert LUTs to logic gates is tricky. Each LUT effectively implements logic that requires some number of logic gates to implement. The number of gates each LUT replaces varies depending on the logical function it implements, so we cannot convert LUTs to gates directly. However, each LUT has 6 inputs, and thus it can replace a finite number of 6-input functions. Specifically, there are $2^{(2^{\wedge}6)} = 2^{64} \approx 1.84*10^{19}$ possible functions each LUT can implement. However, this number corresponds to the number of unique combinations of gates, not the maximum number of gates in a given function. If we consider a logic function that uses all possible logic gates (despite being a nonsensical gate structure; we resolve this later), we can get a hard upper-bound on the number of gates we can represent with a LUT.

If we consider a 6-input gate, we can input each of the 6 signals or their NOTs, which gives us 2^6 possibilities. To implement such a gate, we would need 6-1 = 5 gates in the worst case. So all possible 6-input gates can be made from $2^6 * 5 = 320$ gates. We can repeat this process for 5-, 4-, 3-, and 2- input gates (1-input gates are trivial, and wouldn't count towards gate conversion), to get 516 gates per LUT as a *hard* upper bound. This means we are using no more than (379 *

516) / 250000 = 78% of the board. However, we can further tighten this upper bound by dividing by 2, since if the circuits that contain mirrors of themselves (i.e. A AND NOT A) are reducible to 0. (Note that there are a lot of assumptions built into this upper bound, but most are reasonable to make, such as assuming A AND NOT A will not exist in our function.) Thus, as a ballpark estimate for the hard upper bound of our gate usage, we are using 39% of the A3P250's logic. However, it is much more likely that all of our LUTs implement the logic of 100 or fewer gates, which would give us a (379 * 100) / 250000 ≈ 15% of our logic utilized. This is where the numbers in the table above came from, and either way, we succeeded in meeting the preferred requirements of our logic area.

The next section aggregates our data collection into a table (Table 4.1) and discusses our results in the context of our objectives.

4. Results

Timing	Power	Size
Setup: 1.367 ns Hold: 0.052 ns PW: 3.500 ns	Overall: 0.156 W	Flip-Flop Utilization: 5.4% Certain Logic Utilization: <40% Likely Logic Utilization: <15%

Table 4.1 *Parallelized (simple) schema performance results*

The parallelized schema processes samples at a rate of 500 MHz, or more specifically, it processes one 32-bit 4-sample data word every 8 ns (125 MHz). Our timing report showed that we met slack with about 0.052 ns to spare (on our Nexys 4 DDR), and because we met slack requirements, our parallelized schema works. Additionally, we met our secondary preferred goal of utilizing less than 53% of the total logic area on chip.

5. Discussion

Our main objective was to calculate the pedestal variance of a stream of light level data samples coming into our system at a rate of 500 MHz, processing each sample before the next sample arrives. Our pedestal variance computation accelerator achieves this objective. Additionally, we can see from our table that we met our timing requirements quite tightly. This is good for us, since it means that we do not need to accelerate our hardware past this simple schema, and thus we do not need to use up more power or logic. However, we have some issues with our method that may overshadow these good results.

It would not be unreasonable to protest our results, claiming that these timing reports were generated for a completely different device than the target device. This is true, and we would

not have done so if we had access to both the target device and to Libero, the integrated development environment for Actel/MicroSemi FPGAs. However, we defend our results by claiming that Actel FPGAs are specifically optimized for speed and power, whereas Xilinx FPGAs are designed to have lots of built-in functionality. Thus, because we have satisfied the timing requirements on a Xilinx board, we very likely will satisfy the timing on an Actel board, though some additional work may be necessary to make sure our Verilog uses all the Actel macros available to accelerate processes. The power optimization was always optional, and we should use less power on an Actel board anyway. Lastly, we converted our area utilization from Nexys 4 DDR to A3P250 in the Data Collection section, so that result should be the most accurate. Thus, even though our data collection methods are not ideal, they are defensible, and should yield less optimal results that the actual implementation would. Since these less optimal results still satisfy our requirements, the actual results should also satisfy our requirements.

Now, we will discuss the benefits and costs of our design more broadly. We outlined our development methods in the Methods section of our report. There, we mentioned one of the primary strengths of our FPGA-based approach: that the VERITAS system already uses FPGAs to handle the data immediately coming in from the FADCs. Thus, installing our design into the VERITAS system would require much less time and effort than, say, implementing an ASIC for the same pedestal variance calculations. Really, the biggest drawback of our design methods were that we couldn't develop for the target FPGA directly. We could only synthesize and implement our design on a Nexys 4 DDR board, rather than on the A3P250 boards that VERITAS has installed (and on which our final design will be implemented). We mentioned above as to why this doesn't invalidate our results, but it does necessitate further work to be done to actually implement our design into VERITAS.

As far as future work on this project is concerned, we should stay in contact with Richard Bose so we can help him synthesize and implement our module on the actual Actel FPGAs that VERITAS employs, and to ensure that our module interfaces appropriately with the existing hardware design on the VERITAS FPGAs. Other than that, we have accomplished all that we wanted with our project, and it was a great success!

6. Conclusions

To recap, we developed a new FPGA-based pedestal variance computation accelerator for VERITAS. We developed and verified our design using Vivado and a Nexys 4 DDR FPGA, and found that our design performs within the requirements; we can calculate pedestal variance and process new samples quickly enough to be useful. More work is required to implement the design, but implementing it on the four telescopes of the VERITAS array would improve the overall telescope array performance, as well as allow new types of data to be collected. Specifically, the fast-calculation of pedestal variance would provide VERITAS with a way to continuously monitor light levels with no deadtime, so it would allow VERITAS to detect all light hitting the telescopes' cameras, not just Cherenkov pulses.

With these expansions on the already stunning capabilities of VERITAS, the collaboration will increase the types of data VERITAS can collect. For example, with our upgrades, it may be able detect the first optical counterparts of the mysterious fast radio bursts seen around the universe, or perhaps even measure the distribution and velocity of asteroids in the Kuiper Belt.

7. Deliverables

7.1 Documentation of Our Project

For the purposes of our Senior Design, we must turn in the documentation of our design process. First, we have to turn in this final report, which details our work on this project throughout the semester. Second, we have to turn in a project website; our website can be found at the following URL: https://sites.wustl.edu/varianceaccelerator. Our site presents similar content to that found in our final report but is augmented with more information, and is presented in a more user-friendly, less technical format. The project webpage is still in progress as the time of this writing, but we are set to complete the webpage for final publication by the deadline on April 30th. Lastly, we will give a presentation of our work with a demonstration; this will happen on April 30th as well.

7.2 Hardware Descriptions -- Verilog

Other than our responsibilities for our Senior Design class, we also have to deliver our pedestal variance computation solution to Dr. Buckley. This solution is the system we have described in our report, but concretely, it is the Verilog hardware descriptions of the components that make up our accelerator, as well as the top-level connection of them all. All associated files are listed in Table 7.2.1, along a description of the module in each file.

File Name (Module Name)	Description
g2b.v (GrayConverter)	The module receives a 32-bit 4-sample data word, in which each sample is Gray encoded, and then converts the samples to binary.
g2bModule.v (g2bModule)	This module converts a single 8-bit Gray encoded sample into standard binary encoding.
simple_adder_tree.v (SimpleAdderTree)	This module adds 4 binary inputs together very quickly. It has parameterized input size so we can use it to add up both the samples and the squared samples.
simple_square_sample.v (SimpleSampleSquare)	This module squares each sample in a binary encoded sample word.
SimpleSumStore.v (SimpleSumStore)	This module simply adds the summed samples and the summed squared samples to running totals, which we can use at the end of a sampling window.

Table 7.2.1 *Module Descriptions*

These module will compile onto an A3P250, but some extra work may be needed to integrate this system with the rest of the VERITAS.

8. Schedule/timeline for completion of the project

8.1 Gantt Chart Schedule

Figure 8.1 *Initial Schedule*

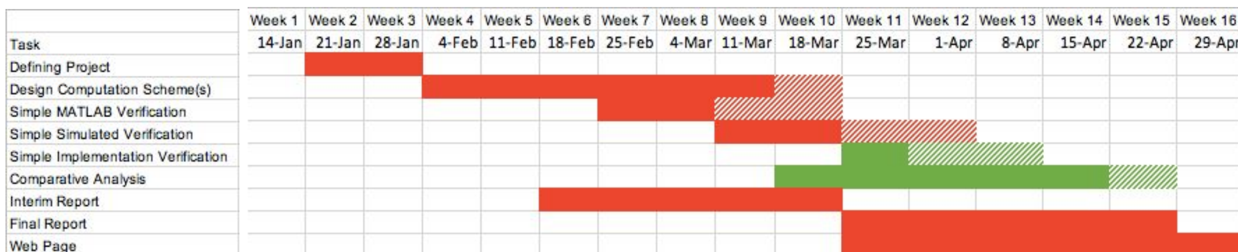

Figure 8.2 *Final Schedule*

We did change our schedule over the course of the project’s development cycle. Of note, the main change was that we declined to expand upon our design further than the simple implementation. The main reason for this, as mentioned in the Discussion, was that the simple implementation met our requirements quite tightly (i.e. we had minimal slack), and so further accelerating our design past the simplest acceleration schema would necessarily yield us worse results. Another change of note was that the development of the first simple version was pushed back significantly. This was because of issues in scheduling meetings with the clients, James Buckley and Richard Bose. Lastly, we downgraded the importance of our Implementation Verification because we realized that, since we do not have access to any Actel FPGAs, we cannot actually verify that our HDL works in the real implementation. Note that we did still verify the implementation on the Nexys 4 DDR board to which we had access.

8.2 Specific Contributions

- Defining Project
    - The project was defined in collaboration specifically with Dr. Buckley

    - Izabella and Matt met with Dr. Buckley several times at the beginning of the semester; Izabella scheduled and coordinated all meetings with both Dr. Buckley and Richard Bose
    - Izabella worked hard to research and understand the underlying physics that motivates both VERITAS and the variance computation accelerator project
    - Matt and Izabella read through existing FPGA hardware design (received from Richard Bose) to understand data flow in VERITAS
- Design Computation Scheme
    - Matt took lead on designing the computation scheme, but he received significant input from Dr. Buckley and Dr. Chakrabartty
- Simple MATLAB Verification
    - As a part of designing the computation scheme, Matt also designed the MATLAB verification
    - Megan helped by generating a data set for Matt to run through the MATLAB verification
- Simple Simulated Verification
    - Izabella worked on designing a Verilog module to convert Gray encoded data words into binary encoded data words
    - Matt worked on designing the accelerated math Verilog modules, specifically:
        - An adder (SimpleAdderTree)
        - A data word squarer (SimpleSampleSquare)
        - And an accumulator (SimpleSumStore)
    - Megan generated a large data set to run in this simulation
    - Matt also worked with Excel to design a verification spreadsheet; Izabella helped with converting Gray code to binary in Excel; Megan helped with data verification
- Simple Implementation Verification
    - Matt worked hard in the lab to interface a UV sensor, Arduino, Nexys 4 DDR, and Desktop Computer! (It was surprisingly difficult!)
- Comparative Analysis
    - Although Matt broke the news that the simple implementation met slack, all read through the implementation reports to validate different parts of the design
- Interim Report
    - All collaborated on every section of the report, either by writing an initial draft or by editing others' drafts
- Final Report
    - All collaborated on every section of the report, either by writing an initial draft or by editing others' drafts
- Webpage
    - Each took lead on a particular page of the site:
        - Matt - Results and Background (top level)
        - Izabella - Introduction, Background, and Methods
        - Megan - Abstract, Overview

9. References

[1] “An Upgrade to the Instrumentation of VERITAS.” [Online]. Available: https://veritas.sao.arizona.edu/~benbow/old/NSF_upgrade.pdf

[2] Holder, J. et al; “The First VERITAS Telescope,” *Elsevier Science, Astroparticle Physics,* vol. 25, no. 6, p. 391-401, 2006. [Online]. Available: https://arxiv.org/pdf/astro-ph/0604119.pdf.

[3] Griffin, Sean, “The VERITAS Bright Moonlight Program,” Ph.D. dissertation, Physics Dept., McGill University, Montreal, Quebec, Canada, 2015. [Online]. Available: https://veritas.sao.arizona.edu/documents/Theses/Griffin_PhD_thesis_final.pdf.

[4] Daniel, M. K., “The VERITAS standard data analysis” in *30th International Cosmic Ray Conference, ICRC 2007, Merida, Yucatan, Mexico, July 3-11, 2007*. [Online]. Available: https://arxiv.org/pdf/0709.4006.pdf.

[5] Abeysekara, A. U. et al, “VERITAS and Fermi-LAT Observations of TeV gamma-ray sources discovered by HAWC in the 2HWC catalog.” [Online]. Available: https://arxiv.org/pdf/1808.10423.pdf.

[6] Private correspondence with acting chairman of VERITAS executive committee Dr. J. H. Buckley.

[7] Tweddale, Jordan, “Selecting Island Pixels with a Likelihood Ratio,” California Polytechnic State University, Dept. of Physics, 2013. [Online]. Available: https://pdfs.semanticscholar.org/40ce/22aed7d81bced28c2ccff05218d05fe85c3c.pdf.

[8] Krennrich, Frank, “Gamma ray astronomy with atmospheric Cherenkov telescopes: the future,” *IOP Publishing and Deutsche Physikalische Gesellschaft, New Journal of Physics*, vol. 11, November 2009. [Online]. Available: https://iopscience.iop.org/article/10.1088/1367-2630/11/11/115008#nj321646bib5.

[9] Private correspondence with VERITAS engineer Richard Bose

[10] Tessier, Russell and Heather Giza, “Balancing Logic Utilization and Area Efficiency in FPGAs,” in *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications (FPL '00)*, Reiner W. Hartenstein and Herbert Grünbacher, Eds. Berlin: Springer-Verlag, 2000. pp. 535-544.

[11] Doran, R W, “Gray Code Fundamentals,” University of Auckland. [Online]. Available: http://rvk.tripod.com/index-2.html.

[12] Avins, Jerry, “Converting Between Binary and Gray Code,” dspguru.com, 2000. [Online]. Available: https://dspguru.com/dsp/tricks/gray-code-conversion/.

[13] Microsemi, “ProASIC3 Flash Family FPGAs with Optional Soft ARM Support,” *Microsemi Corporate Headquarters,* 2016. [Online]. Available: https://www.microsemi.com/document-portal/doc_download/130704-ds0097-proasic3-family-flash-fpgas-datasheet.