DESIGN AND VERIFICATION OF THE MULTI-SLIT SOLAR EXPLORER CAMERA FIELD PROGRAMMABLE GATE ARRAYS

by

Jordan M. Johnson

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

Jacob Gunther, Ph.D. Major Professor	Chris Winstead, Ph.D. Committee Member

Don Cripps, Ph.D.
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2026

ABSTRACT

Design and Verification of the Multi-slit Solar Explorer Camera Field Programmable Gate Arrays

by

Jordan M. Johnson, Master of Science

Utah State University, 2026

Major Professor: Jacob Gunther, Ph.D.
Department: Electrical and Computer Engineering

MUSE (Multi-slit Solar Explorer) is a NASA program with a mission to take high-resolution images of the Sun's corona. The MUSE instrument consists of two cameras: one Spectrograph (SG), and one Context Imager (CI). Each camera is controlled by digital logic embedded in Field Programmable Gate Arrays (FPGAs). These FPGAs are primarily responsible for acquiring, processing, storing, and transmitting pixel data from the CCDs. The FPGAs are also responsible for various auxiliary functions to maintain camera operation, such as tracking hardware temperature and monitoring for single event effects (SEEs) caused by radiation.

This project is the design and verification of a specific portion of the digital logic in the MUSE cameras. This scope includes the majority of the image data path and select auxiliary modules. Specifically, this project includes modules that rearrange pixel data in raster-scan order, stream pixel data across a clock domain boundary, provide a clock-domain-crossing Wishbone bridge, integrate an image compression IP core, monitor pixel values and circuit current draw to check for SEEs, monitor camera temperature, calculate a histogram for every image, transform pixel data via a programmable lookup table, and provide a means for controlling heater power remotely. Each module has particular requirements that complicate

the design. For example, the images produced by the CI camera are too large for the image compression IP core to compress. The FPGA design therefore must divide the image and compress each piece separately. This complication, as well as other complications, is discussed in greater detail for each module.

Three select modules are formally verified: the module that places pixels in raster-scan order, the module that calculates a histogram, and the module that implements a clock-domain-crossing Wishbone bridge. The design for these modules is intricate; formal verification will provide the necessary confidence these modules work correctly in all scenarios. Properties for these modules are written in the Property Specification Language (PSL) and are proven via k-induction using open-source tools.

(110 pages)

PUBLIC ABSTRACT

Design and Verification of the Multi-slit Solar Explorer Camera Field Programmable Gate Arrays

Jordan M. Johnson

The Multi-Slit Solar Explorer, or MUSE, is a NASA mission that will take images of the Sun to study solar flares and the solar corona. The mission will provide insight into the mechanisms behind space weather. The mission consists of two cameras: the Spectrograph (SG), and the Context Imager (CI). The Utah State University Space Dynamics Laboratory is providing both cameras for the mission.

This report describes a part of the design and verification process for a central component on these cameras known as the Field Programmable Gate Arrays (FPGAs). These FPGAs are programmed to acquire, handle, and send images captured from the camera's sensors.

These FPGAs have additional functions. For example, the FPGA controls a heater to ensure the camera does not get too cold. Additionally, the FPGA continually monitors a critical component in the camera to make sure it continues working even in the presence of the radiation of space.

Pieces of the design are checked using tools that mathematically prove correct operation. This report describes how this process, known as formal verification, was applied in the MUSE FPGA design.

ACKNOWLEDGMENTS

I am indebted to Benjamin Oborn for taking a chance on me and allowing me to work on this project. I am grateful to him for guiding me through these years of development as my supervisor and for having patience as I learn.

I am grateful to my major professor, Dr. Gunther, for his willingness to support me even in the busiest times and for his excellent guidance.

I am grateful to my committee members, Dr. Winstead and Dr. Cripps, for their willingness to support this project, for igniting a passion in me for digital system design, and for their sage advice.

I am grateful to Angel Gifford, Kevin Spencer, Thane Winward, Hannah Brailsford, and Kelsie Johnson for their work to make publishing this report possible.

I am grateful to all of the MUSE team for their cooperation on this project.

I am grateful to Eric Allred for introducing me to the amazing world of electronics, for giving me confidence I could become an engineer, and for the invaluable hands-on experience his years of teaching has provided. Without Eric's influence, it is unlikely I would have pursued this field.

Jordan M. Johnson

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

ADC	Analog-to-Digital Converter
AXI	Advanced Extensible Interface
BMC	Bounded Model Check
CCD	Charge-Coupled Device
CCSDS	Consultative Committee for Space Data Systems
CDC	Clock-Domain Crossing
C&DH	Command and Data Handling
CI	Context Imager
DDR	Double Data Rate
DMA	Direct Memory Access
DWT	Discrete Wavelet Transform
ECC	Error Correction Code
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GHDL	G Hardware Description Language
HDL	Hardware Description Language
IP	Intellectual Property
LRB	Line-Rasterizing Buffer
LSRAM	Large Static Random-Access Memory
MCP	Multi-Cycle Path
MTBF	Mean Time Between Failures
MUSE	Multi-slit Solar Explorer
PCB	Printed Circuit Board
PLL	Phase-Locked Loop
PLUT	Programmable Lookup Table
PSL	Property Specification Language
PWM	Pulse Width Modulation
RTG4	Radiation-Tolerant Generation 4

Continued on next page

RX	Receive
SEE	Single Event Effect
SET	Single Event Transient
SG	Spectrograph
SoC	System-on-Chip
SPI	Serial Peripheral Interface
TX	Transmit
VHDL	Very High Speed Integrated Circuit Hardware Description Language

CHAPTER 1

INTRODUCTION

MUSE, the Multi-Slit Solar Explorer, is a NASA mission for which the Utah State University Space Dynamics Laboratory is building the cameras. The MUSE instrument contains a multi-slit spectrometer. It will take images of the Sun and will advance our knowledge of the Sun's corona. The MUSE instrument contains two cameras: a Spectrograph (SG) camera, and a Context Imager (CI) camera.

The SG is a multi-slit spectrometer and has three separate charge-coupled devices (CCDs) for imaging. Each of these CCDs has an active image area of about 1024 x 1024. The CI is a monochrome camera and has one CCD for imaging. The active image area for this CCD as used in the MUSE design is 4096 x 2048. The two cameras are completely separate. There is no direct communication between the cameras. Instead, the spacecraft electronics control each camera as needed.

The cameras contain Field Programmable Gate Arrays (FPGAs) to orchestrate control of the camera. In response to commands sent over SpaceWire, the FPGAs drive camera hardware to acquire image data and transform, buffer, compress, and transmit that image data. The FPGAs carefully coordinate these operations to ensure that the camera acquires frames at a precise framerate, that changes in the camera's configuration are applied at the appropriate time, and that image loss is minimized. This report describes the design and verification process of the camera's FPGA design.

Various factors make this FPGA design tricky. For example, the cameras are responsible for reading from multiple image sensors in parallel. The camera is required to be reconfigurable even while it is currently acquiring images. The configuration update is required to apply to the next set of images. Pipelining is required to achieve the required framerate, but this pipelining complicates correctly applying an updated camera configuration.

The MUSE cameras use specifically the Microchip RTG4 FPGA. These radiation-tolerant FPGAs are designed to withstand harsh radiation environments and are appropriate for the MUSE mission. The MUSE camera is required to use this specific FPGA due to flight heritage.

The MUSE cameras are also required to use a specific IP core to perform image compression according to the CCSDS 122.0-B-1 standard. The RTG4 FPGA has limited memory. The bulk of memory comes from 209 LSRAM blocks, or "Large SRAM" blocks, spread throughout the device. Each LSRAM can store up to 24,576 bits. This IP core takes up a large portion of this memory, and the remaining memory is insufficient for the rest of the design. See Appendix B for information on resource utilization. As a result, each camera has two FPGAs. The limited connectivity between the two FPGAs introduces another set of design challenges.

This particular IP core introduces additional challenges on the CI camera. This camera produces images that are too large for this IP core to compress. The CI camera therefore splits the image into four divisions and compresses each piece individually.

Most functionality is handled by the primary FPGA named the Command and Data Handling (C&DH) FPGA. The secondary FPGA, named the Compression FPGA, handles image compression and contains the aforementioned IP core. Both FPGAs exist on the same PCB and are connected via a number of traces on the circuit boards.

The scope of this project is a specific portion of the FPGA design for the cameras on the MUSE instrument. It is not the entire design, but the project contains the majority of the image data path in addition to auxiliary components. For a list of modules within the scope of this project, see Figure 2.1.

1.1 Requirements

Many program requirements are levied on the MUSE cameras which drive the design of the FPGA. Multiple engineers are responsible for the development of this FPGA, so the scope of the project is a specific portion of the design. The requirements in the scope of this project are the following:

- rearrange pixel data in raster scan order with enough throughput to avoid being the bottleneck
- buffer pixel data in memory to allow higher framerates
- stream pixel data between the two FPGAs across a clock domain boundary
- control the Compression FPGA via a custom clock-domain-crossing Wishbone interface
- integrate an image compression IP core per program requirements
- monitor current draw on the ADC power rails and pixel values to detect a Single Event Effect (SEE) and power down the ADC
- monitor other measurements important to the health of the camera, including camera temperature, through a separate housekeeping ADC
- calculate a histogram for every image
- transform pixel data via a reprogrammable lookup table
- allow bypassing the programmable lookup table for the image output and the histogram independently
- allow configuring heater power through the Wishbone bus

1.2 SpaceWire Interfaces

The camera has two SpaceWire interfaces, one per FPGA. Commands are sent to the camera via the C&DH SpaceWire interface, and the camera sends reply packets through the same interface. The camera sends image data out from the Compression SpaceWire interface.

1.3 The CCDs

The purpose of this section is to give background information on the image sensors. This information is relevant to the design of the MUSE FPGA.

The MUSE program uses CCDs for imaging. A CCD is an image sensing device consisting of a 2D grid of capacitors. As light reaches the sensor, the photons are converted to electrons, and the capacitor charges. The controlling device - the FPGA in the case of MUSE - controls the clock signals to shift charge between adjacent capacitors. The CCD Interface module is responsible for controlling these clock signals. This module is outside the scope of this report, but its operation is highly relevant to the Rasterizer's operation.

During readout, the controller first shifts the entire image down one row in the CCD. This action moves the charge in the bottommost row down into a special row called the register. The controller then shifts the data in the register to either the left or the right by one pixel. With an amplifier, the outermost single pixel value is presented on an output of the CCD as an analog voltage. This output is referred to as a tap. This process is repeated until the entire image has been read out of the tap.

To speed up readout of the CCD, the MUSE camera reads out of two taps of each CCD simultaneously. To accomplish this, the left half of the register is shifted towards the left, and the right half is shifted towards the right. The ADC, which has multiple channels, measures the value of both taps simultaneously. As a result, one shift of pixels in the register results in two separate pixel values.

This technique increases readout speed at the cost of pixel data being split into two separate streams. It adds onto the FPGA the responsibility of merging the two pixel streams into one serial pixel stream. The primary purpose of the Rasterizer is to perform this task and merge these streams into one raster scan stream of pixel data.

1.3.1 The SG CCDs

The SG camera will take images in three spectral passbands and requires three CCDs. The CCDs used on the SG are the Teledyne e2v CCD47-20.

The CCD47-20 has a separate image area and storage area. The image area is exposed to light, and the storage area is covered. The image area has 1024 columns and 1024 rows,

and the storage area has 1033 columns and 1024 rows. Additionally, there are 16 dark reference rows on the left and right sides of the image and storage areas, and there are 3 dark reference rows at the top of the image area. An additional 8 blank pixels are added per tap for each row read out of the CCD [2]. See Figure 1.1.

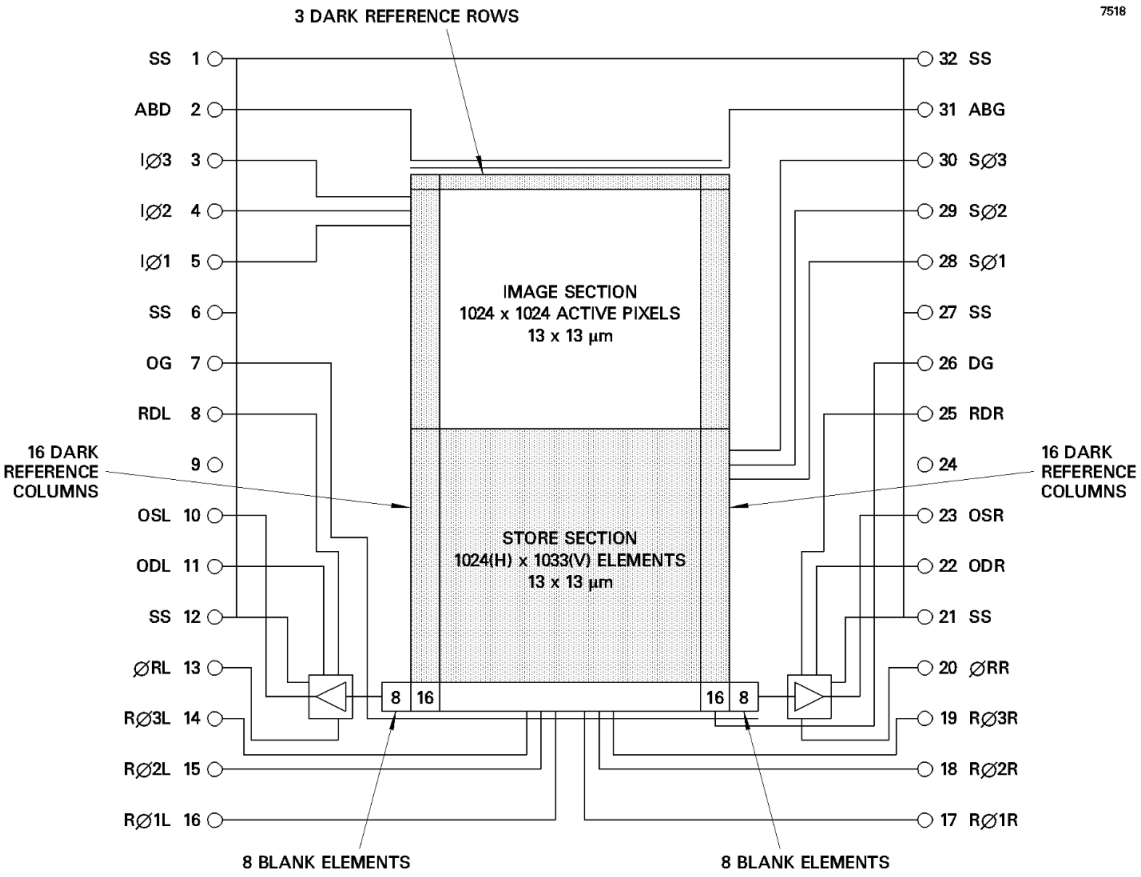


Fig. 1.1: The structure of the CCD47-20, the CCD used on the SG. This is an excerpt from the part's datasheet [2].

1.3.2 The CI CCD

The CI camera is monochromatic and requires one CCD. The CCD used on the CI is the Teledyne e2v CCD203-82.

The CCD203-82 is structured differently than the CCD47-20. The CCD203-82 is vertically symmetric. There are two image areas; image area D is on the top half of the CCD,

and image area A is on the bottom half. Pixel data in image area D may be read out through the two taps on the top labeled G and H, and image area A may be read out through the two taps on the bottom labeled E and F. See Figure 1.2.

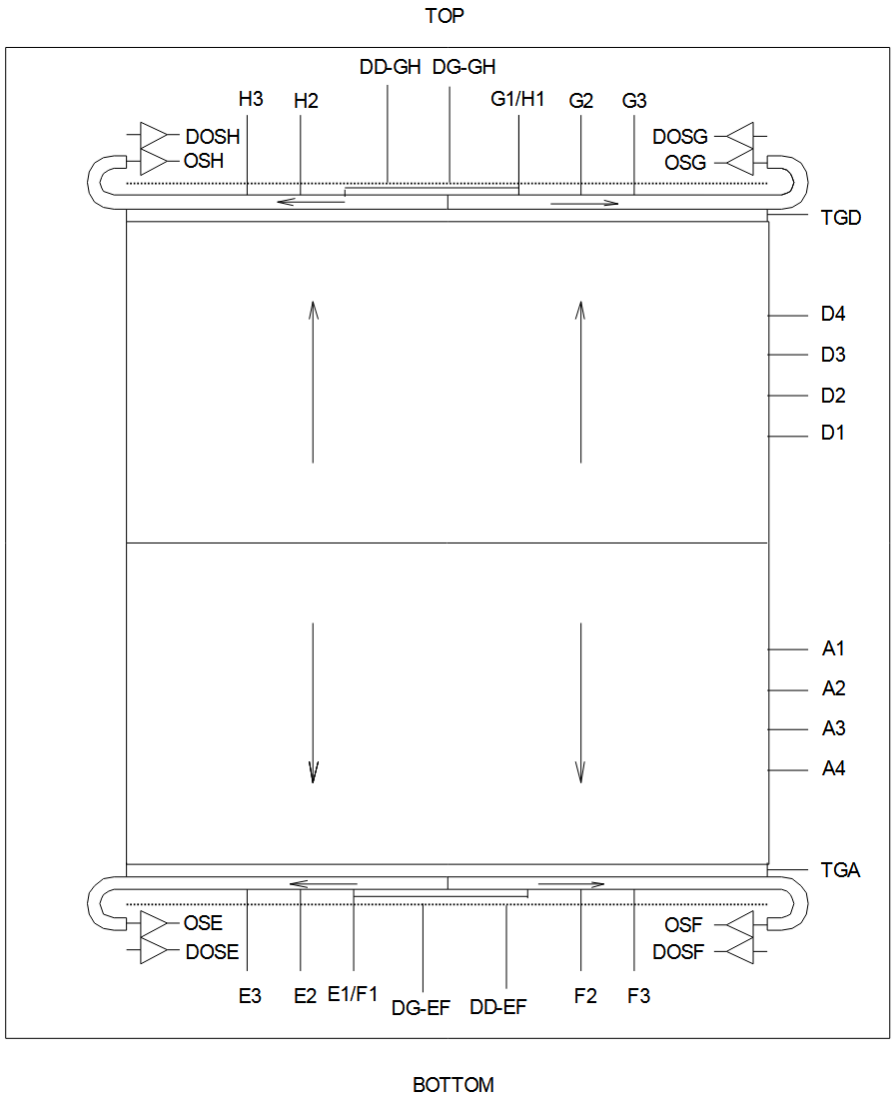


Fig. 1.2: The structure of the CCD203-82, the CCD used on the CI. This is an excerpt from the part's datasheet [3].

Both image areas have 4096 columns and 2068 rows. An additional 50 blank pixels are added per tap for each row read out of the CCD [3].

In the MUSE camera system, image area D is exposed to light and is used for imaging,

and image area A is covered and is used for storage during readout. The taps G and H are left unused in the MUSE system. Only taps E and F are used.

In this report, image area D is referred to as the CI CCD's "imaging area", and image area A is referred to as the CI CCD's "storage area". Additionally, tap E is referred to as the CCD's left tap, and tap F is referred to as the CCD's right tap. This vocabulary is used to discuss the camera design in a way that is generic over both the SG and the CI CCDs.

1.3.3 Report Structure

Chapter 2 of this report provides a broad overview of the design architecture. The chapter discusses how functionality is broken up into modules, the memory bus, data flow, control flow, and clock domains. The broad details provided in this chapter motivate the design choices discussed in the following chapters.

Chapter 3 of this report describes the design of modules involved in the camera's data path. These modules are responsible for storing and transforming pixel data. Detailed descriptions and block diagrams are provided.

Chapter 4 of this report describes the design of modules not involved in the camera's data path. These modules handle other tasks; for example, the Heater PWM module allows for controlling a heater inside the camera.

Chapter 5 of this report describes how the design was implemented and verified. This chapter discusses the three methods used to verify the design: formal verification, simulation, and integration tests.

Chapter 6 reports the results of the project. This chapter includes images and describes issues that were encountered.

Chapter 7 discusses the results. This chapter discusses which aspects of the design and verification worked well and which aspects could be improved.

CHAPTER 2

ARCHITECTURE

The appropriate point to begin discussion on the MUSE design is the overall architecture. This chapter discusses the various modules the design is composed of, how data moves through these modules, and how the state of the camera advances over time. The information in this chapter motivates many design details of each module.

2.1 System Overview

The camera design is divided into modules. Each module throughout the camera is responsible for a different function.

Some modules lie in the path of pixel data. These modules are responsible for acquiring, transforming, storing, and outputting pixel data. Other modules lie outside the path of pixel data. These modules are responsible for other tasks necessary to keep the camera functioning, such as controlling a heater.

The camera does not have software or a microprocessor. The functionality of the camera is implemented entirely with digital circuitry.

Figure 2.1 contains a block diagram of the modules composing the FPGA design. Only the modules highlighted green are within the scope of this report. Some modules are instantiated multiple times as discussed in Chapters 3 and 4.

2.2 Memory Bus

The camera has a large number of parameters, including window boundaries, integration time, compression parameters, and many others. These parameters must be configurable through the C&DH SpaceWire interface.

There are many possible solutions to this problem. One solution is to define special packets for different parameters. A module would receive from the C&DH SpaceWire

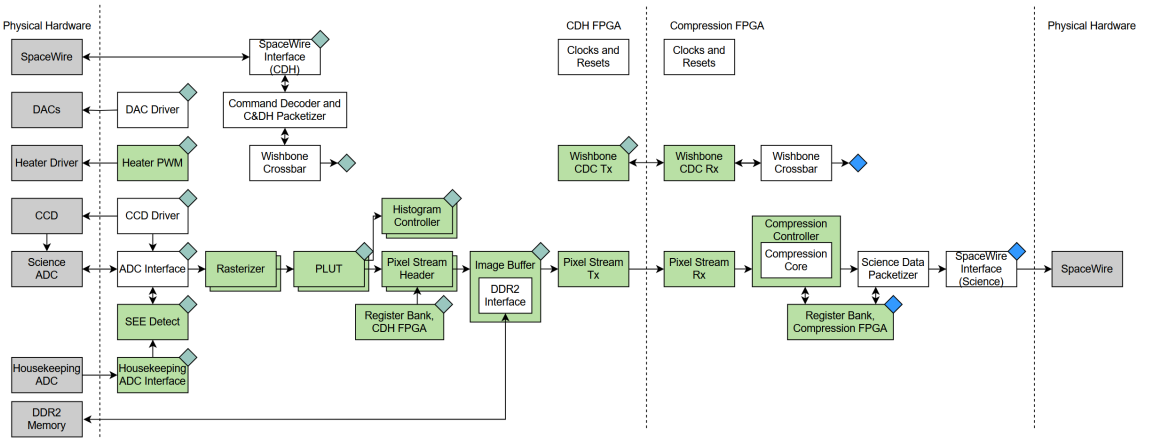


Fig. 2.1: A simplified block diagram of the MUSE FPGA design. Items in green are the scope of this project. This scope includes the entire original design and verification process. The two exceptions are the Image Buffer and Housekeeping ADC modules where the scope includes substantial contributions to existing designs. The blue diamonds represent modules with a connection to the Wishbone bus.

interface and would then output different values to the camera's other modules in response. The large amount of configuration parameters makes this ad-hoc approach impractical.

The camera design instead allows for reconfiguration via a memory bus. The camera configuration is stored in registers in each module, and each module connects to the memory bus to provide read and write access to these registers.

The Command Decoder and C&DH Packetizer modules respond to commands sent via the C&DH SpaceWire interface by performing read and write accesses on this memory bus to configure the camera. The design of these modules are outside the scope of this report. Each module is allocated 128 words on the bus for its configuration registers. The memory map is shown in Appendix A.

The memory bus is a Wishbone bus. Wishbone, a royalty-free standard, was chosen because its simplicity reduces the logic required in each module. Wishbone's simplicity also makes interfaces easier to formally verify than other standards like AXI. Unusually, the Wishbone standard implements word addressing, where each address corresponds to one word of data. This is in contrast to byte-addressing as used by other standards like AXI.

The word length and address length are both 16 bits. As a result, registers storing

unsigned integers may contain values in the range of 0 to 65,535. This range is too small for some parameters, so modules concatenate multiple 16-bit registers to form wider registers as needed.

The C&DH FPGA contains a Wishbone crossbar. Every module with configuration registers connects to this crossbar via a point-to-point connection. Due to the limited number of signals between the two FPGAs, the Compression FPGA contains its own Wishbone crossbar. This crossbar is a peripheral to the main crossbar in the C&DH FPGA. Notably, the interface between these two crossbars passes through a clock-domain-crossing Wishbone bridge. The design of this bridge is described in Section 4.4.

The camera contains a 6-gigabit DDR2 memory used solely to buffer image data. Unusually, this memory is *not* directly mapped to the Wishbone bus. Only the Image Buffer module interacts with this memory.

2.3 Data Flow

The primary purpose of the MUSE camera's FPGA is to handle image data. In broad terms, the FPGA must move pixel data between memories, transforming the pixel values along the way.

The pixel values in each image pass through multiple modules, each of which stores and transforms pixel values differently. This section gives a broad overview of this data path and the shape of image data at each stage.

Throughout the camera, pixel values are grouped by image. Grouping pixels into images allows the camera to maintain an unchanging configuration for each image, to gracefully drop images when buffers are full, and to associate metadata, like a serial number, with each image.

In the camera, modules send image data to each other serially, one pixel at a time. This serial stream of pixel data is augmented with extra information to denote the start and end of each image. In some cases, this extra information is an end-of-image flag sent as one extra bit in parallel to the data. In other cases, this extra information is in a header at

the beginning of the stream of data denoting the length of the following data. The details for each interface are described below.

2.3.1 Data out of the CCD Interface

As the CCD Interface acquires an image, it outputs *two* streams of pixel data. This is because, as described in Section 1.3, data is read out through two taps of the CCD to speed up image readout.

The packets in this stream are full images delineated with the packet terminating flag approach.

As the CCD Interface module reads out the last column of each row, it asserts the last-column flag. And, as the CCD Interface module reads out the last row of image data, it asserts the last-row flag. The end of the image, therefore, is indicated by both of these signals being high simultaneously. This structure is seen in Figure 2.2.

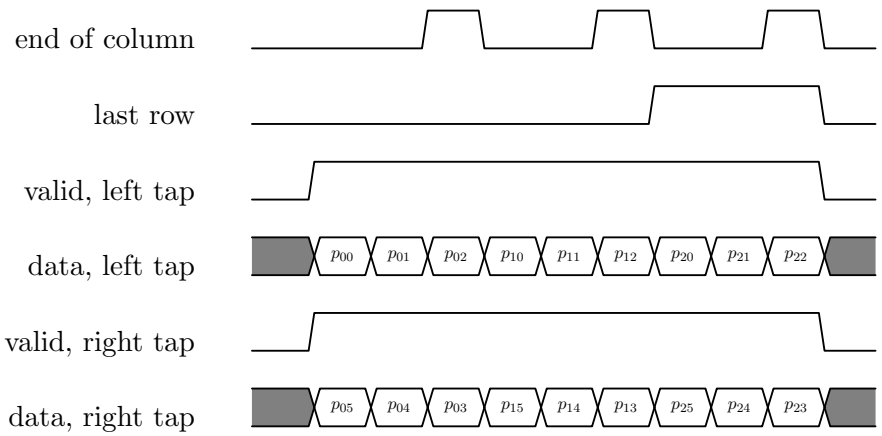


Fig. 2.2: Example stream of data between the CCD Interface module and the Rasterizer module. In reality, there are many more pixel values, and pixel values are separated by multiple clock cycles.

In Figure 2.2, data labeled p_{xy} corresponds to a pixel at coordinates (x, y) on the sensor.

Note the camera is configurable to read out of only one of the two taps. In this case, all of the pixel values will be in either the left or the right tap.

2.3.2 Data out of the Rasterizer

The Rasterizer transforms data from the CCD Interface's output by first combining the two streams into one stream of data in raster-scan order. Additionally, for the benefit of other modules, the Rasterizer outputs information about each pixel, including the pixel's coordinates on the sensor (x, y), whether the pixel is within the window, and, on the CI, which division a pixel is inside of. See Figure 2.3.

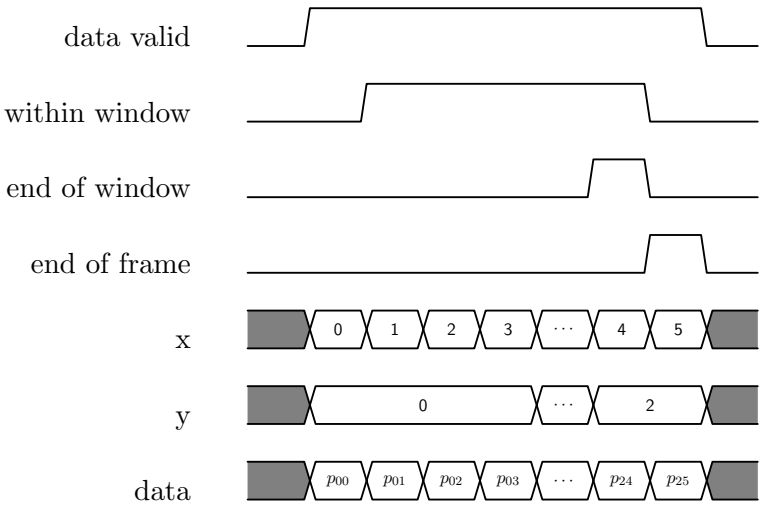


Fig. 2.3: Structure of data output from the Rasterizer module. Not shown are the within-division and end-of-division flags present only on the CI camera.

2.3.3 Data out of the PLUT

The Programmable Lookup Table modifies each pixel value output from the Rasterizer. Otherwise, the PLUT outputs data in a structure identical to the Rasterizer's output.

2.3.4 Data out of the Pixel Stream Header Prepender

The Pixel Stream Header Prepender module prepends the stream of pixel values with a header. This header is written to memory and is later used by the Compression Controller module. Header data includes the image size, a unique serial number for the image, and compression parameters. Most of these values are given to the Pixel Stream Header

Prepender module by the C&DH Register Bank. See Figure 2.4.

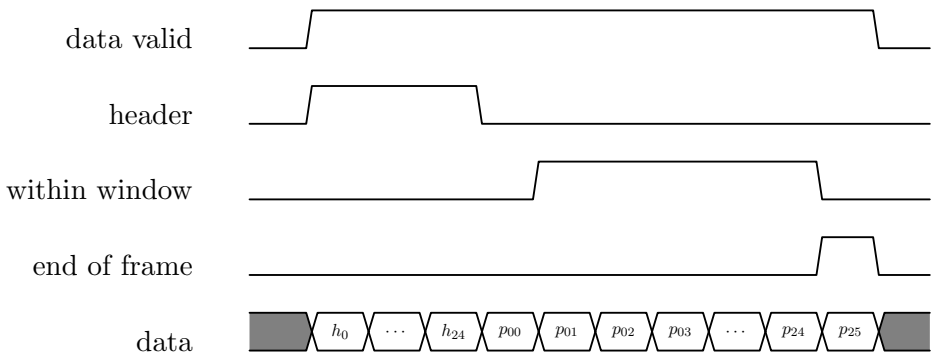


Fig. 2.4: Structure of data output from the Pixel Stream Header module.

2.3.5 Data out of the Image Buffer buffer module

The structure of data between the Image Buffer module and the Compression Controller module is an exception in the data flow path. Packets are delineated with a start-of-image flag instead of an end-of-image flag.

Additionally, the Image Buffer module outputs data in multiples of 128 words. If the total amount of data in an image is not a multiple of 128 words, the Image Buffer will output extra meaningless data until a multiple of 128 words is reached.

The Compression Controller module must ignore this extra data. To accomplish this, that module reads two fields in the header data, `IMAGE_X` and `IMAGE_Y`. The controller knows the total amount of image data read into the module is `IMAGE_X` $\times$ `IMAGE_Y` + 25 words, since the header contains 25 words. See Figure 2.5.

2.3.6 Data out of the Compression Controller

The interface between the Compression Controller and the Science Data Packetizer is simply image packets terminated by an end-of-image flag. The data values may be either 16-bit uncompressed values or 32-bit compressed values.

Finally, the interface between the Science Packetizer and the SpaceWire is of packets with end-of-packet terminators. This interface is outside the scope of this report.

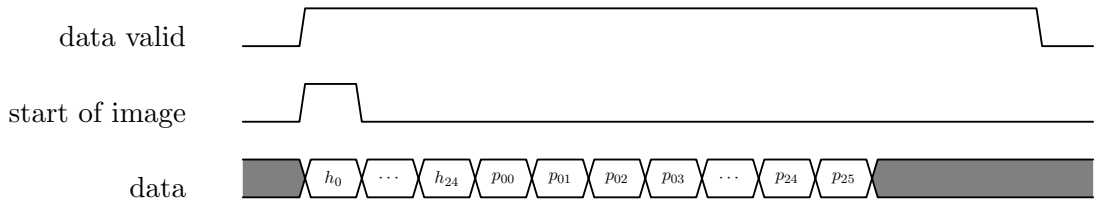


Fig. 2.5: Structure of data output from the Image Buffer module. Notice some extra words at the end of the packet are delivered to the Compression Controller; the Compression Controller therefore must rely on the image size fields to process the correct number of words from each packet. In reality, the number of words is a multiple of 128.

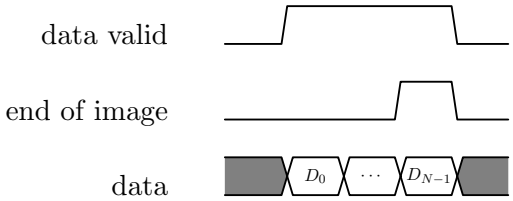


Fig. 2.6: Structure of data output from the Compression Controller module.

2.4 Control Flow

This section describes *control flow*, or how the state of the camera progresses through time. Control flow describes when and how the various transformations described in the previous section occur.

2.4.1 Command Period

An important requirement for the MUSE camera is that the camera's integration time should be precisely controllable. To accomplish this, the design has the concept of a command period.

Relevant to this discussion are three registers:

- The length of the command period, T_c
- The frame repetitions register
- The image control register

Setting the first bit in the image control register instructs the camera to begin a command period. At the start of this first command period, all image sensors are flushed by

actuating the sensor's clock and dump signals. After the flush is the integration time. The integration time is the time the image sensor captures light that will be eventually be read out. The integration time ends once the command period has completed. The pixels are quickly transferred from the image section of the CCD to the storage section and are then read out through the ADCs. See Figure 2.7.

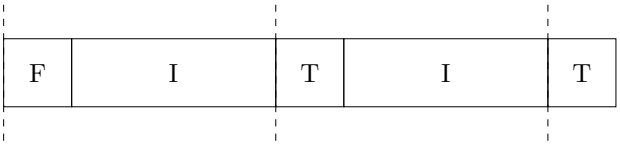


Fig. 2.7: Diagram of the command period. F represents a flush, where the residual charge of the CCD is removed. I represents image integration. T represents image transfer, after which the sensor is read out. The dashed line represents the start and end of the command periods.

The camera is pipelined. The stages of the camera may be working on different images at any given time.

2.4.2 Initial Design Iteration

The initial design of the camera was that of modules connecting to each other with ad-hoc interfaces. Modules would decide themselves when to begin and end processing images based off of the end-of-image flag present in the data stream. This design worked as a starting point for the camera's design, but it became clear this approach had significant problems.

The problem of dropping images

The first signs of problems were encountered during development when work began on gracefully handling the dropping of images. Dropping images is required when there is not enough room in the image buffer to store another image.

One approach to solving this problem would be to design the camera such that dropping image data is never required. Unfortunately, such a solution is not feasible. As Dr.

Gisselquist explains in his article "AXI Stream Is Broken", every boundary between a fixed-rate data source and a variable-rate data sink is a point where data loss is an unavoidable possibility [4].

Such a boundary exists in the MUSE FPGA on the input to the image buffer in DDR memory. Data loss will occur if the compression IP core or the Science SpaceWire output processes pixels too slowly and the image buffer becomes full. The camera must handle this situation gracefully.

With the initial iteration, however, handling the situation gracefully would be difficult. As each stage in the pipeline could be possibly working on data for a different image, complex, error-prone logic would have been required to drop only the correct image data and ensure every module returns to a valid state for the next image.

The problem of updating configuration

The second major problem encountered with this initial iteration was that of maintaining a consistent configuration for each image. Various modules throughout the camera rely on parameters, including the window size, compression parameters, and integration time.

In the initial design, updates to the camera configuration would apply immediately, even if the camera was currently processing image data. Parameters would change from underneath modules currently processing image data, and the camera would malfunction as a result.

One approach to solving this problem is to stage the camera's desired configuration and only update the true configuration between images. Unfortunately, as the camera is pipelined, pieces of the camera may be processing different images at a given time, and so updating the configuration would require flushing the entire image pipeline. Because the camera is required to be reconfigurable on-the-fly, and because pipelining is necessary to achieve the required framerate, this simple solution does not work.

There is a clear need to stage the camera's configuration and apply pieces of the configuration at the right time. The initial iteration greatly complicated this task.

The solution

It gradually became clear the initial iteration made these tasks difficult because the state of the camera was very fine-grained. Every module in the camera could possibly be in a different state, handling possibly a different image, at any given time.

The overall camera state was complicated as a result. The overall camera state at a given moment could be one of a vast combination of states of the individual modules. This made operations like dropping images and updating the camera's configuration difficult because the operations had to account for all of these possible states the camera could be in. This led to complicated, error-prone designs.

The solution was to reduce the set of possible camera states by forming groups of modules based off the tasks the modules perform.

2.4.3 Second Design Iteration

The second iteration of the camera's control flow groups the operation of the camera into three tasks: integration, readout, and compression. This approach still allows for pipelining since the camera may be performing these tasks to different images in parallel. This approach is shown in Figure 2.8.

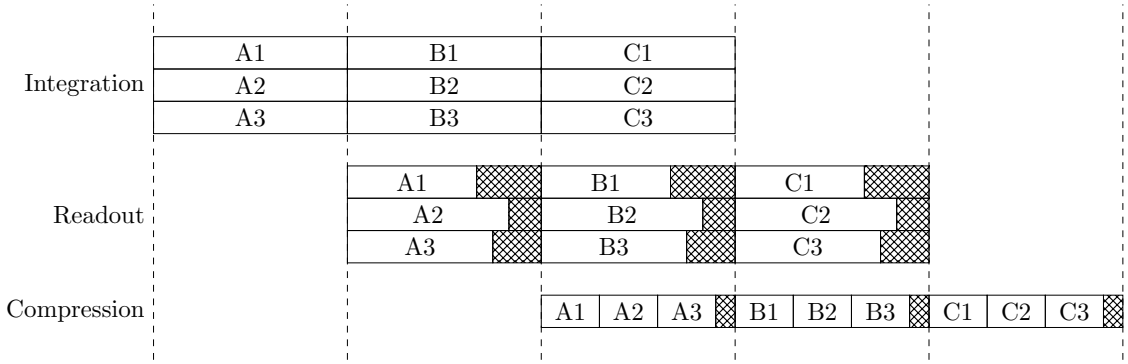


Fig. 2.8: A depiction of the second control flow iteration on the SG camera. In this depiction, the camera captures nine images, three at a time. The dashed lines represent the start and end of command periods.

In the second iteration, each task begins at the start of a command period and ends

at the very end of that command period. In a sense, the second iteration groups modules to form a coarsely-grained pipeline of three stages.

With this approach, dropping images becomes simple. If a command period ends and a task has not yet completed, the image the task is working on is dropped.

Correctly staging and applying camera configuration is also simplified. It is not as simple as applying the configuration to each module at the start of a command period. This direct approach leaves open the possibility of updated parameters applying inconsistently to different images in the pipeline.

For example, say the camera configuration was updated in the second command period of Figure 2.8 and is therefore applied at the very start of the third command period. The intention is for the updated configuration to only affect the images labeled C1, C2, and C3. However, any updated parameters that affect the readout task will also affect images B1, B2, and B3, and any images that affect the compression task will affect all images currently in the pipeline. The result is portions of the updated configuration applying to images that should not have been affected.

To fix this problem, the camera configuration is delayed through a pipeline of registers. This pipeline advances at the end of each command period. Modules in each task would read from the appropriate point in the pipeline. For example, modules in the Compression task would read the camera configuration at the end of this pipeline, after it has been copied three times. This approach is equivalent to each stage providing a copy of the camera's configuration for the following stage to use.

While this iteration simplifies the design greatly, it also adds an unacceptable amount of latency. The readout stage always takes T_c to complete, even if the entire image is written to memory faster. If the command period is long and the window size is small, most of the time in the readout task is spent waiting for the command period to complete, and image compression cannot begin until the readout task is complete.

The third design iteration solves this problem.

2.4.4 Third Design Iteration

The third iteration of the camera is shown in Figure 2.9. In this iteration, the readout and compression tasks run as soon as data is available rather than at the beginning of the next command period. The command period no longer dictates the flow of the image pipeline except that the camera begins taking a new image at the start of each command period.

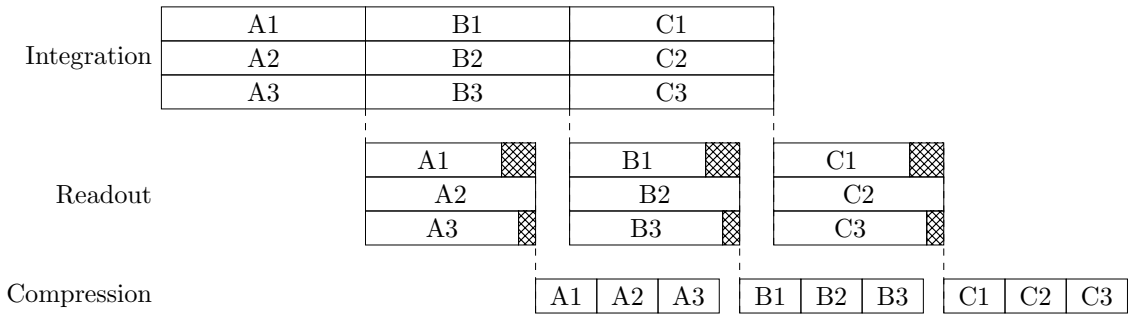


Fig. 2.9: The third iteration of camera control flow for the SG.

Dropping images remains simple. Once a task begins, an image is dropped if the buffer is full. For example, if the readout task begins but the image buffer is full, the readout task will drop the image.

This design does slightly complicate the problem of applying configuration parameters at the right time. The configuration is still delayed through a pipeline of registers, though the stages of this pipeline no longer advance all at the same time. The first stage advances when integration begins, the second stage advances when readout begins, and the third stage advances when compression begins. This approach works so long as any given stage only advances once the next stage has had an opportunity to read the current stage's configuration.

This solution works, but it is still flawed. Implicit in this and previous iterations is an assumption the image buffer in memory is a ping-pong buffer.

During operation, the camera will often be reconfigured from taking large images at a slow rate to taking small images at a higher rate. When this occurs, the last large image

may not be read out from the ping-pong buffer in time for the new small images, resulting in data loss. In addition, the Science SpaceWire interface may output image data at a variable rate. If this rate varies too much, the buffer may on occasion not empty in time, resulting in dropped images. For this reason, it is valuable to have a large image buffer that can store many images at once. The fourth and final design iteration implements this buffer.

2.4.5 Fourth Design Iteration

The fourth design iteration changes the Image Buffer module to work as a virtual packet FIFO buffer. A packet FIFO works on entire packets instead of individual elements, and a virtual FIFO is a FIFO with a large, external backing memory.

This iteration solves the problem described above, but it complicates staging and applying configuration. In this design iteration, all camera configuration parameters used by modules in the compression stage are stored in memory alongside each image. The Pixel Stream Header Prepender module prepends this image data to be written to memory.

This iteration is the final iteration of the MUSE camera design. The design is the result of balancing design simplicity and robustness against performance requirements.

2.5 The two FPGAs

As described in Chapter 1, as a consequence of the requirements levied on the cameras, the MUSE camera contains two FPGAs: the C&DH FPGA, and the Compression FPGA. This split design introduces multiple challenges.

The first challenge is interconnectivity. Inside one FPGA, it is trivial to introduce new signals connecting two modules. There are a limited number of traces connecting the two FPGAs, however, so such connections cannot be freely made across this split. In the MUSE FPGA design, two interfaces exist: a Wishbone interface for configuration, and an interface for streaming pixel data.

The second challenge is related to timing. The logic in each FPGA lives in a separate clock domain, so logic must cross this clock domain boundary.

2.6 Clock domains

The camera design consists almost entirely of synchronous logic. The design is split into multiple clock domains. Table 2.1 contains every clock domain in the design.

Table 2.1: Clock domains

Clock Domain	Frequency (MHz)	Source
Input system clock (C&DH)	40	Camera hardware
Input system clock (Compression)	40	Camera hardware
Core clock (C&DH)	40	PLL
Core clock (Compression)	40	PLL
PLL auto-reset logic (C&DH)	50	Internal RC oscillator
PLL auto-reset logic (Compression)	50	Internal RC oscillator
Inter-FPGA pixel stream	10	PLL
Science ADC interface	280	Clock from AD9257
DDR2 interface	160	PLL
C&DH SpaceWire link RX input	33.3	Recovered from SpaceWire
C&DH SpaceWire link TX output	80	PLL
Science SpaceWire link RX input	33.3	Recovered from SpaceWire
Science SpaceWire link TX output	40	PLL
Science SpaceWire link TX logic	80	PLL

The number of clock domains is large, but the vast majority of logic lies within the two 40 MHz core clock domains.

The two 40 MHz core clock domains are considered separate domains. This is because the PLLs used in the RTG4 FPGA are configured to use triple-mode redundancy. When configured in this manner, the PLLs do not guarantee a phase relationship between the reference clock and the output clocks.

2.6.1 Interfaces between domains

Special care must be taken when designing interfaces that cross clock domains. If timing constraints are not met, circuit elements may become metastable. Even in the absence of metastability, care must be taken to correctly send data across a clock domain boundary.

Two of these clock domain crossing interfaces are in the scope of this report. The first interface is the Inter-FPGA pixel stream described in Section 3.5, which uses asynchronous

FIFOs to send data between clock domains. The second interface is the Wishbone CDC module pair described in Section 4.4, which uses a more involved clock domain crossing technique.

CHAPTER 3

IMAGE DATA PATH COMPONENTS

This chapter discusses the design of the various FPGA components involved in the data path.

3.1 Rasterizer

Modules in the MUSE FPGA design - particularly the Compression Controller and Science Data Packetizer modules - expect to receive image data in one serial stream of pixel values in raster-scan order.

However, as discussed in the introduction, pixel values are read out from two taps of the CCD. This means data enters the FPGA through *two* serial streams of pixel data, not one. Additionally, one stream receives data in reverse order from the other. Pixel data exits the left tap in order from left to right across the sensor, while pixel data exits the right tap in order from right to left across the sensor. The Rasterizer's primary responsibility is to correctly merge each of these two streams of pixel data into one stream of pixels in raster-scan order. See Figure 3.1.

The stream of pixel data from the left tap is termed the *forward* stream, and the stream of pixel data from the right tap is termed the *reverse* stream.

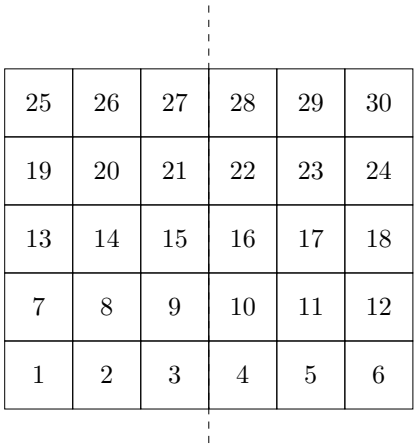
Because each Rasterizer reorders the pixel data from one CCD, there is one Rasterizer per CCD. As such, the Rasterizer is instantiated three times on the SG and once on the CI.

An additional responsibility of the Rasterizer is to determine the x-coordinate or the column, and the y-coordinate, or row, each pixel occupies within the image.

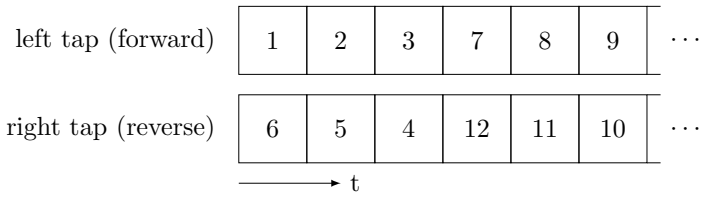
These coordinates are required for the following purposes:

1. Determining whether each pixel is within the window

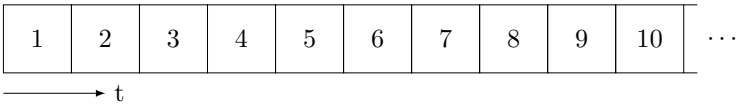
2. Signaling to modules when a pixel is the last pixel in the window



(a) Example CCD layout.



(b) Output of ADC Interface across time.



(c) Output of Rasterizer across time.

Fig. 3.1: An example CCD layout with each pixel labeled with a number (a). When both taps are enabled, pixels enter the Rasterizer in two streams (b). The Rasterizer must output pixels in raster-scan order (c).

3. Splitting the image into up to four divisions in the Context Imager

4. Separating pixels into subregions to calculate different histograms for different portions of the image

Items 1, 2, and 3 are the responsibility of the Rasterizer, while item 4 is the responsibility of the Histogram Controller modules.

The Rasterizer design is a hierarchy of modules. The top-level Rasterizer module instantiates a Ping-Pong buffer, and each side of the Ping-Pong buffer contains a Line-Rasterizing Buffer. The design of each module is described in the following sections.

3.1.1 Line-Rasterizing Buffer

The Line Rasterizing Buffer (LRB) lies at the bottom of the Rasterizer hierarchy. It is the part of the Rasterizer responsible for merging the forward and reverse streams into one ordered output stream.

Pixels from the forward stream must be output from the LRB in the same order the pixels were received. Likewise, pixels from the reverse stream must be output from the LRB in the reverse order the pixels were received.

It was recognized that a FIFO (First-In, First-Out) buffer could store data from the forward stream and output the same data in order. Similarly, a stack could store data from the reverse stream and output the same data in the opposite order it entered the Rasterizer.

Therefore, the initial design of the Rasterizer instantiated one FIFO buffer and one stack buffer on each side of the Ping-Pong buffer. The FIFO would accept pixel data from the forward stream, and the stack would accept pixel data from the reverse stream. Then, for each row, the Ping-Pong buffer would read out all data from the FIFO first, then all data from the stack. See Figure 3.2.

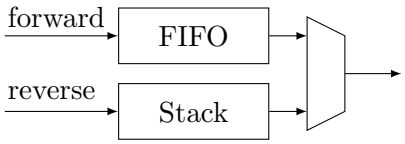


Fig. 3.2: A high-level block diagram of the initial LRB design.

This design worked, but it was wasteful with memory resources. Let N_C be the number of columns in the image, or the number of pixels in each row. For the SG, $N_C = 1072$, and for the CI, $N_C = 4196$; for an explanation, see Section 1.3.

Under normal circumstances, image data will be read out from both taps of the CCD. In this case, the forward and reverse streams will each contain $\frac{1}{2}N_C$ pixels per row. Therefore, the FIFO and stack need only enough room to contain $\frac{1}{2}N_C$ words each.

However, the cameras may be configured to read out of only one of the two taps. Because of this, it is possible for all N_C pixels in the row to come from the forward stream

if only the left tap is enabled, or for all N_C pixels to come from the reverse stream if only the right tap is enabled. While the total number of pixels in the row, N_C, is always the same, the distribution of these N_C pixels between the two streams may vary. Because of this, this design of the Rasterizer would only work if the FIFO and stack each are able to contain up to N_C pixels. As a result, the total memory required across the FIFO and stack with this design is $2N_C$.

To reduce resource utilization, the Rasterizer was redesigned to replace the FIFO-stack pair with one buffer, the LRB. The buffer acts similar to the FIFO and stack combination shown in Figure 3.2.

The LRB is a buffer with *two* write ports and one read port. The LRB makes use of the RTG4's dual-port LSRAM memory, which allows hardware to write memory at two separate addresses simultaneously. The dual-port LSRAM also permits reading from two separate addresses simultaneously, but the LRB does not make use of this feature.

The LRB is designed to be used in two distinct phases: the *fill* phase, and the *drain* phase. During the fill phase, the module interfacing with the LRB writes pixel values through the forward and reverse streams. The buffer fills up until it contains a full row of pixels, after which the interfacing module switches to the drain phase. During the drain phase, the module interfacing with the LRB reads data from the LRB by asserting the `r_req` signal. Once the LRB is empty, the module may return to the fill phase by writing the next row of pixels.

The LRB works by storing three pointers to memory: `fw`, `fr`, and `s`. The LRB uses `fw` and `fr` to implement FIFO-like behavior. The LRB uses `s` to implement a full descending stack in the same memory.

The LRB is empty at the start of the fill phase. When empty, `fw` and `fr` both point to the first address, and `s` points one past the last address in the memory.

As each pixel from the forward stream is sent to the LRB, the LRB places the pixel value at address `fw` and increments `fw` by one. As each pixel enters from the reverse stream, the LRB places the pixel value at address $\texttt{s} - 1$ and decrements `s` by one. As a result, data

from the forward stream grows in memory from the left, and data from the reverse stream grows in memory from the right. See Figure 3.3.

The LRB has at least N_C spots in memory. Because there is always N_C pixels in each row, the two sides of the buffer never overwrite each-other.

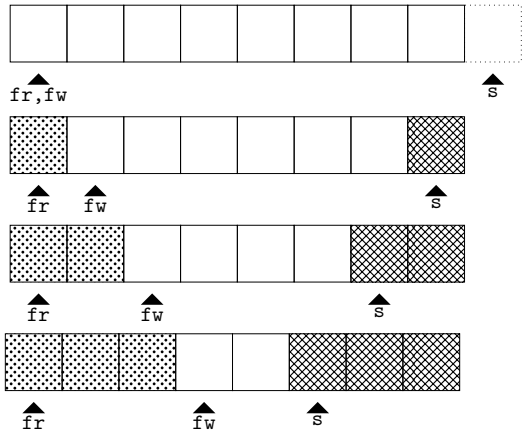


Fig. 3.3: The LRB across time as pixels enter from each stream. The dotted pattern represents pixels from the forward stream, and the crosshatch pattern represents pixels from the reverse stream. This example shows writing six pixels to an LRB configured for $N_C \leq 8$.

The drain phase is split into two parts. First, data is read out from the left of the buffer from left to right at address `fr`. With each value read, `fr` increments by one. Once all data has been read out from the left of the buffer, `fw` and `fr` both reset to point to the leftmost element. This is important to prepare the LRB for the next image, and it is a significant difference between the forward-side of the LRB and a traditional FIFO buffer.

Then, data is read out from the right side of the buffer at address `s`. With each value read, `s` increments by one. See Figure 3.4.

To support dropping data, the LRB has a drop input that causes the pointers to all reset to their initial state. The later control flow designs described in Section 2.4 make this functionality unnecessary.

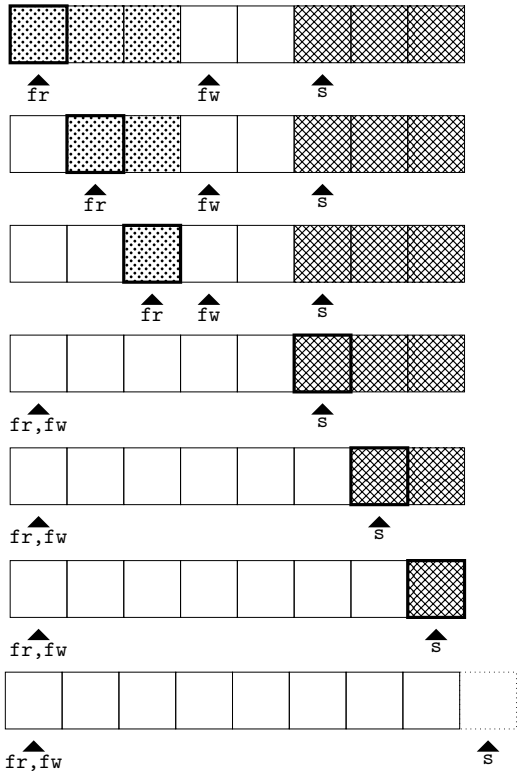


Fig. 3.4: The LRB across time as pixels exit the buffer. The dotted pattern represents pixels from the forward stream, and the crosshatch pattern represents pixels from the reverse stream. The highlighted element is the element read out of the LRB at each point in time.

3.1.2 Ping-Pong Buffer

The LRB has an important restriction in how it may be used. When the LRB is in its drain phase, there must be no pixels written to the LRB until it has been completely emptied. Furthermore, the Rasterizer expects to handle a full row of pixels at a time, so no pixels may be read out of the LRB until a full row of pixels has been written.

Unfortunately, the ADC Interface sends pixels to the Rasterizer fast enough there is not enough time to fully read out one row before the next row arrives.

Ping-pong buffers are a common technique used in situations where continuous data streaming is required but a buffer exists in the data flow path that cannot be written to and read from simultaneously. The ping-pong buffer wraps two copies of the buffer and provides write access to one and read access to the other at any point in time.

In the MUSE system, the Rasterizer contains the Ping-Pong module, where each side

contains an LRB. The Ping-Pong module allows for the Rasterizer to handle a continuous stream of pixel data.

At any point in time, one side of the Ping-Pong buffer contains an LRB in its fill phase, and the other side of the Ping-Pong buffer contains an LRB in its drain phase. The buffers swap at the end of each row, causing the now-full LRB to enter the drain phase and the now-empty LRB to enter the fill stage.

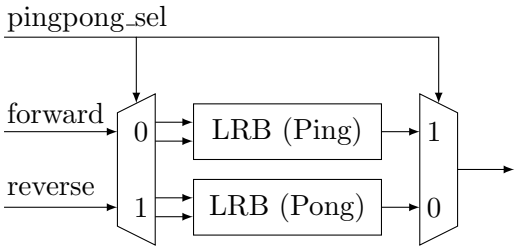


Fig. 3.5: High-level block diagram of the ping-pong buffer; individual signals are not shown. Note well the indices on the multiplexers; when data is written to one LRB, data is read from the other LRB.

The ADC Interface module provides a last-row indicator along its pixel stream. The Ping-Pong must correctly delay this signal such that the Rasterizer sees the last-row indicator as the Ping-Pong buffer outputs the corresponding row of data.

To accomplish this, the Ping-Pong buffer sets an internal flag high when it receives the last-row indicator from the ADC Interface module. Then, when the buffers swap, the internal flag resets, and if the flag was high, the Ping-Pong asserts its own last-row indicator on its output until the buffers swap again.

When the Ping-Pong buffer swaps, it will drop the data within the LRB that is becoming the write buffer. That LRB should already be empty. The top-level Rasterizer module checks this is true; if it is not, it signals an error.

The Ping-Pong buffer also has a `drop` input. This input is simply forwarded to both of the LRB's `drop` inputs.

3.1.3 Rasterizer Top-Level

The Rasterizer top-level instantiates the Ping-Pong buffer to implement row rasterization. The top-level then augments the row-rasterized data with metadata, including:

- `data_valid`: an indicator for whether the Rasterizer is currently outputting valid pixel data and metadata
- `x` and `y`: the x- and y-coordinates of each pixel within the image
- `within_window`: an indicator for whether the pixel is within a user-defined window
- `end_of_window`: an indicator for whether the pixel is the last pixel within the window
- `end_of_row`: an indicator for whether the pixel is the last pixel within each row
- `end_of_frame`: an indicator for whether the pixel is the last pixel within the entire frame
- `within_division`: four indicators, one per division, for whether the pixel is in a division; only used on the CI
- `end_of_division`: four indicators, one per division, for whether the pixel is the last pixel within the division; only used on the CI

The Ping-Pong buffer and the LRB buffers are only concerned about each row in the image. The Rasterizer top-level is concerned about the entire image, so more detail on the data entering the Rasterizer from the ADC Interface module is required.

An important detail in the design of the Rasterizer top-level is that the window size is configurable. The main reason for allowing a configurable window size is to speed up image acquisition. The speedup comes from two factors. First, with a smaller window, less data needs to be transmitted. Second, the CCD Interface does not need to serially shift out pixels in rows that are not in the window.

Because of this second point, the ADC Interface module will not send the Rasterizer any rows outside the window. The ADC Interface module still sends all pixels on any row

that's within the window, including pixels in columns that are outside the window. So, the Rasterizer should expect to receive a static number of pixels per row, N_C, but a variable number of rows from the ADC Interface.

For an example, suppose the CCD has an image size of 8×8 pixels. Also suppose the user-configured window ranges from columns $x = 2$ through $x = 5$ and rows $y = 3$ through $y = 4$. No pixels on any row $y \in \{0, 1, 2, 5, 6, 7\}$ will show up at the input to the Rasterizer, but all pixels on rows $y \in \{3, 4\}$ will, regardless of the x-coordinate of these pixels. So, the Rasterizer will receive $8 \times 2 = 16$ pixels for this image. The Rasterizer will then filter out the pixels outside of the window in each row and output $4 \times 2 = 8$ pixels for this image.

Determining x and y

The coordinates of a pixel, (x, y), correspond directly to the physical bin in the CCD the pixel came from. The Rasterizer must determine which physical bin on the CCD each pixel came from by counting the pixels output from the Ping-Pong module.

The Rasterizer determines the x-coordinate of each pixel within the image by incrementing a counter with every pixel read out of the Ping-Pong buffer. This counter is reset at the end of each row.

Determining the y-coordinate is more involved. A simple row count is insufficient, because the user-configurable window size may begin at a row other than $y = 0$, and rows outside the window are not sent to the Rasterizer. Additionally, the CCD Interface module has binning and skip-row options. When these options are enabled, the rows sent to the Rasterizer are not contiguous.

To determine the y-coordinate, the Rasterizer first maintains a row count r. This counter increments at the end of each row and resets to zero at the end of the image.

The Rasterizer then determines a scaling parameter s and offset b, where

$$(s,b) = \begin{cases} (2,3) & \text{if both binning and skip row are enabled} \\ (1,1) & \text{if either binning or skip row is enabled, but not both} \\ (0,0) & \text{if neither binning nor skip row are enabled} \end{cases} \tag{3.1}$$

Then, the Rasterizer finds y through

$$y = 2^s r + w_{br} + b \tag{3.2}$$

where w_{br} is the bottom row of the window. The term $2^s r$ is found trivially in hardware by left-shifting r by s bits. The addition by b is only required because when skip-row is enabled, the CCD Interface module skips the lower row. For example, if the window is configured to include rows 0 through 3 and if skip-row is enabled, the the CCD Interface module would skip row 0, send row 1, skip row 2, and send row 3.

Determining indicator signals

The indicator signals the Rasterizer outputs along each pixel are determined by the (x, y) coordinates of each pixel as determined above.

Let w_{tr} be the top row of the window, w_{br} be the bottom row, w_{lc} be the leftmost column, and w_{rc} be the rightmost column.

The `within_window` indicator is asserted when $w_{lc} \leq x \leq w_{rc}$. Note a check on y is not required, since the Rasterizer does not receive any row outside the window.

The `end_of_window` indicator is asserted when $x = w_{rc}$ and when the Ping-Pong is asserting `last_row`.

The `end_of_row` indicator is asserted when $x = N_c - 1$.

The `end_of_frame` indicator is asserted when `end_of_row` is asserted and when the Ping-Pong is asserting `last_row`.

Determining CI-specific indicator signals

The Rasterizer on the CI produces additional signals relating to the image divisions.

The image divisions are configurable through three division boundary registers. Let d_i be the value of the ith division boundary register with $i \in \{1, 2, 3\}$. Each division boundary register contains the rightmost boundary of each division, inclusive.

To be specific, Division 1 spans $w_{lc} \leq x \leq d_1$, Division 2 spans $d_1 < x \leq d_2$, Division 3 spans $d_2 < x \leq d_3$, and Division 4 spans $d_3 < x \leq w_{rc}$.

The user is constrained to only set these registers such that

$$w_{lc} \leq d_1 \leq d_2 \leq d_3 \leq w_{rc} \tag{3.3}$$

.

This constraint prevents divisions of negative width, but it allows for divisions of zero width for divisions 2, 3, and 4. For example, if $d_3 = w_{rc}$, then division 4 will be of zero width. Configuring a division to be of zero width effectively disables the division.

The Rasterizer internally determines which divisions are enabled by determining whether the right inclusive boundary of the division is greater than the left exclusive boundary of the division. The first division is an exception: as the left boundary is inclusive, the minimum valid width for the first division is 1, so the first division is always enabled.

The Rasterizer outputs a four-bit vector `within_divison` along each pixel. The Rasterizer asserts the ith bit in this vector by comparing the x-coordinate of each pixel against the left and right boundaries of the ith division. For example, the Rasterizer asserts the fourth bit in `within_division` if $d_3 < x \leq w_{rc}$.

The Rasterizer also outputs a four-bit vector `end_of_division` along each pixel. The Rasterizer asserts the ith bit in this vector if x equals the ith division's right inclusive boundary and if the ith division is enabled. For example, the Rasterizer asserts the third bit in `end_of_division` when $x = d_3$ and the width of the third division is nonzero.

3.2 Programmable Lookup Table

One requirement of the camera is to provide programmable lookup tables. The Programmable Lookup Table (PLUT) is a module that allows mapping every pixel value to another pixel value. Per design requirements, only the bottom 14 bits of each pixel is considered, so the PLUT memory contains 2^{14} elements. The width of the memory is 16 bits, so each pixel may be mapped to a 16-bit value.

At the heart of the PLUT is a memory. When transforming pixel values, each pixel is treated as an address into the memory. The value stored in the memory at that address is used as the new pixel value. The specific memory used is the RTG4's LSRAM.

The LSRAM provides two useful features used in the PLUT. The first feature is the memory's dual-port mode, where two separate read/write ports can be used simultaneously to access the memory. In the PLUT, Port A is used to provide read and write access to the Wishbone bus, and Port B is used during image acquisition to transform the value of each image. Without dual-port mode, the design would have been complicated by the need to arbitrate read access to the memory.

The second useful LSRAM feature used in the PLUT is ECC functionality. Unlike other memories throughout the design, the values stored inside the PLUT may persist for a long time. The PLUT is therefore more susceptible to soft errors caused by radiation and requires error correction and detection.

The LSRAM memory blocks inside the RTG4 implement ECC through shortened hamming codes. The available codes are limited. When configured to store 16-bit values, the code used is (36, 29) [5].

As values are read from each port, the LSRAM signals if a one-bit error was corrected or a two-bit error was detected. When an error is captured, the address of the problematic value is stored in memory, and an error is sent to the Error Handler module.

Reading values from the LSRAM with ECC functionality enabled incurs a delay of three clock cycles. The PLUT must account for this delay in two locations.

First, the PLUT must delay data associated with each pixel value, such as the pixel's

(x, y) coordinates, by three clock cycles to match the data. This is done with a simple pipeline of registers.

Second, the PLUT must ensure data read from the Wishbone bus is not out of date. When the Wishbone bus attempts to read from the PLUT memory, logic in the PLUT module stalls the bus if the memory was recently written to. This prevents incorrect data from being read from the PLUT's memory.

3.2.1 Bypassing

Bypassing the PLUT is possible. The PLUT outputs both the transformed and untransformed pixel values. The untransformed pixel values are simply the pixel values received at the PLUT's input delayed by three clock cycles.

The pixel values exiting the PLUT go to both the Histogram and the Pixel Stream Header Prepender modules. For maximal flexibility, the camera has two independent configuration options: one sets whether to transform pixel values going to the Histogram, and the other sets whether to transform pixel values going to the Pixel Stream Header Prepender.

3.2.2 Initialization

The initial design had a flaw; after reset, the PLUT would constantly report ECC errors. The reason for this was that the memory was uninitialized. The solution was to add initialization logic. After reset, this logic takes over Port A and writes to the entire memory. Only after this initialization sequence are ECC errors reported.

3.3 Pixel Stream Header Prepender

As described in Section 2.4.5, to allow for the Compression FPGA to consistently read the camera's configuration as it was when the image was first taken, it is required to write the configuration to memory alongside the data. This is the purpose of the Pixel Stream Header Prepender.

The module works by delaying the pixel stream long enough to insert this information at the beginning of the stream of pixel data. If the length of the header is S, the data is delayed through a pipeline of S registers. See Figure 3.6. This design is not optimal with respect to resource usage, but as used in the MUSE camera, $S = 25$, so the waste is minimal.

In idle, `data_sel` = `header_valid` = `header_idx` = 0. When the module receives the first pixel, it asserts `header_valid` and increments `header_idx` once per clock cycle. When `header_idx` = $S - 1$, the entire header has been output, and the module sets `data_sel` = 1 to allow for the pixel data to flow through.

When the last pixel exits the module, the module sets the three signals back to 0 in preparation for the next image.

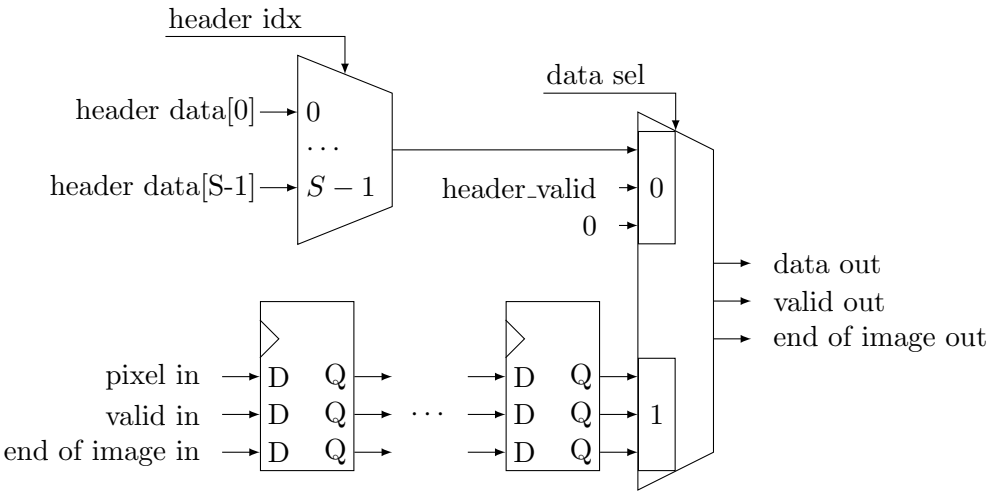


Fig. 3.6: Circuit diagram for the Pixel Stream Header Prepender.

The module is generic over exactly what data belongs in the header; it is the C&DH Register Bank's purpose to create this data.

3.4 Image Buffer

The Image Buffer module is responsible for storing pixel values in a buffer. This module lies between the camera's front-end, including the CCD Interface, Rasterizer, PLUT, and

Pixel Stream Header Prepender, and the camera's back-end, including the Compression Controller, the Science Data Packetizer, and the SpaceWire Interface.

The portion of this module's design that is within the scope of this report is the descriptor FIFO design. Originally, the Image Buffer was outside the scope of this report, but the introduction of framerate requirements required substantial changes to the original design and the introduction of this descriptor FIFO.

A large buffer is required for multiple reasons. The first reason is that a buffer is required between the front-end and the back-end because the back-end applies backpressure but the front-end produces pixels at a fixed rate and does not accept backpressure. At such a boundary, a buffer is required to prevent constant data loss. Because the CCD is read out at a fixed rate, and because the SpaceWire interface may apply backpressure arbitrarily, eliminating this boundary by design is not possible.

The second reason is that the SG camera reads out from three CCDs in parallel, but as the design only has room for one compression IP core, only one image may be compressed at a time. The image buffer allows for serializing the image data.

The Image Buffer module is the one module in the design that interfaces with the camera's large DDR2 memory. This radiation-tolerant 768 MiB DDR2 memory module contains far more memory than available in the RTG4 FPGA. The DDR2 memory interface is nontrivial but is outside the scope of this report.

The Image Module instantiates an IP core implementing Reed-Solomon error correction coding. For every 32 bits of data, the core adds 16 check bits. Even with this large overhead, the available memory exceeds the needs of the camera with a total of 512 MiB of data. One MiB is $2^{20} = 1,048,576$ bytes.

3.4.1 Descriptors

As described in Section 2.4, the control flow of the camera underwent multiple iterations and so did the operation of the image buffer. The final design is similar to common Direct Memory Access (DMA) module designs.

In the final design, the memory is allocated according to Figures A.2 and A.3. On the SG, the buffer has room for a total of 81 batches of 3 images, one from each CCD. On the CI, the buffer has room for a total of 6 images.

As described in 2.3, a 50-byte header is stored with each image. Each slot in memory is large enough to store this header and one full image.

Access to the memory is controlled via a set of FIFOs containing descriptors. A descriptor is a data structure containing two values: a pointer to an address in memory, and a length. When the camera front-end finishes writing to memory, the Image Buffer module writes a descriptor to a FIFO. The presence of a descriptor in a FIFO therefore represents valid image data being present in memory. The descriptor is only taken out of the FIFO once the back-end finishes reading the image data, signaling new data may be written to the slot in memory now.

Each slot in the FIFO corresponds to one slot in the memory. Since the FIFO contains as many slots as there slots in memory, the FIFO's full flag indicates when the image buffer is full. If the front-end begins reading an image but the full flag is asserted, the entire image is dropped.

Determining whether to drop the image this way is a significant simplification over previous design iterations. Previously, the front-end of the camera was required to support dropping an image in the middle of readout if the image buffer became full. Now, the full flag being low indicates there is room for an entire image, and therefore the hardware will be able to write the entire image to memory without concern of the buffer becoming full.

3.5 Inter-FPGA Pixel Stream Interface

The Inter-FPGA Pixel Stream Interface is a pair of modules responsible for transmitting data between the FPGAs. As described in Section 2.6, the phase relationship between the core clocks in the two FPGAs is unknown. Sending pixel values between FPGAs requires crossing a clock domain boundary.

Multiple solutions are possible; see further discussion on clock domain crossing in Section 4.4. The solution used in the MUSE camera is to use clock-domain-crossing FIFOs.

The C&DH FPGA sends pixel values to the Compression FPGA via a source-synchronous interface by sending a clock along with the data. The Compression FPGA accepts data from this interface using the forwarded clock and then sends this data to a clock-domain-crossing FIFO.

Backpressure is required. The Compression FPGA sends a signal to the C&DH FPGA to indicate its FIFO is full. Closing timing with this signal required reducing the clock frequency. This frequency was reduced to 10 MHz. As a result, another clock domain boundary exists on the C&DH FPGA, for which another clock-domain crossing FIFO is used.

Since one pixel is transferred per clock cycle, the limit through this interface is 160 Mbit/s. This limit is significantly faster than other bottlenecks in the system; in particular, the theoretical limit of how quickly the SG may read pixel data from the three sensors is 96 Mbit/s, and the limit for the CI camera is 64 Mbit/s. Therefore, this frequency is sufficient. Section 6.1 describes throughput bottlenecks in more detail.

If 10 MHz were not fast enough to meet requirements, other solutions to this problem are possible. For example, the Compression FPGA could assert a signal when its FIFO is within some N elements of being full. The C&DH FPGA could treat this signal as coming from a different clock domain and re-synchronize it to its own core clock. This design would work so long as N is large enough to account for the synchronization delay.

3.6 Compression Controller

It is a requirement of the MUSE camera system that it incorporates a specific IP core to perform image compression. The IP core implements the CCSDS 122.0-B-1 image compression standard. This compression standard uses the Discrete Wavelet Transform (DWT) in a manner similar to JPEG compression and supports both lossless and lossy compression. The Compression Controller instantiates this IP core and is responsible for interfacing with it.

3.6.1 Configuring the Compression Core

The IP core contains configuration registers that determine how it functions. For example, these registers control the image size and whether the compression is performed in a lossy or lossless manner. Complicating matters is the requirement the SG camera supports three different compression configurations, one per CCD, despite there only being one compression core as discussed in Section 2.5. The only solution is therefore to continually re-configure the compression core between each image.

The Compression Controller receives data in 17-bit words. As described in Section 2.3, the topmost bit is a start-of-image indicator, and the other 16 bits form a stream of data starting with the image header followed by the image data. Importantly, all the compression parameters are stored in the image header.

The state machine in the Compression Controller therefore begins by consuming words from the input stream and discards words until it sees the start-of-image flag. Then, the state machine accepts and stores words from the input stream until it has received an entire image header. At this point, the Compression Controller now has enough information to configure the compression core, so the state machine writes to the compression core registers. After a short delay to ensure the parameters have had time to propagate into the core, the state machine finally allows the pixel data to flow from the input stream into the compression core.

As described in Section 2.3, streams of data read from memory are padded to extend the number of words to a multiple of 128. It is important these padding words are not delivered to the compression core, especially since the core was found to become stuck in an inoperable state if it received more pixels than the image size as indicated by the core's configuration registers. The Compression Controller therefore counts the number of pixels entering the compression core and stops the flow of data once the correct number of pixels have been delivered.

This design means the input stream must be diverted into two directions: first, towards a small set of registers to store the image header, and second, towards the compression core.

3.6.2 Allowing for Uncompressed Images

The camera has an additional requirement that the camera supports taking uncompressed images. To implement this, a compression-enable flag is present in the image header. After receiving the image header, the Compression Controller checks this flag. If the flag is low, the state machine instead diverts the stream of pixel data directly out from the Compression Controller.

This means the input stream must, in fact, be diverted in three directions, the third direction being directly out from the Compression Controller.

3.6.3 Test Images

The camera is required to be capable of outputting test image data in various locations. One of those locations is immediately before the science data packetizer. The Compression Controller accesses a Wishbone-accessible register from the Compression Register Bank that, when set, causes the module to continually send a test pixel on every clock cycle to the Science Data Packetizer. The value of the test pixel is also configurable via Wishbone.

3.6.4 Direct Access to the Compression Core Registers

The design of the MUSE camera places the main compression configuration registers in the C&DH Register Bank. These registers can be written to and read from using the Wishbone bus. These values are added to the image header and sent to the Compression Controller, which then writes the registers to the compression core. Therefore, under normal operation, the Compression Controller directly writes to the compression core's registers.

The camera has an additional requirement, however, that the core's registers should be directly accessible. The Compression Controller therefore has additional logic to directly access these registers.

CHAPTER 4

AUXILIARY COMPONENTS

This chapter discusses the proposed design of certain FPGA components that indirectly aid in the task of collecting image data.

4.1 Histogram

The MUSE camera is required to output a histogram for each image. This histogram gives information required to quickly react to solar flares.

It is required each histogram contains 2^{10} bins. Additionally, the camera design is required to divide the image into subregions: two subregions per CCD on the SG camera and three subregions on the CI camera. Histograms are calculated independently for each subregion. The SG camera therefore produces six histograms per batch of three images, and the CI camera produces three histograms per image. The subregion boundaries must be reconfigurable.

The Histogram Core and Histogram Controller modules fulfill these requirements. These modules accept image data as it is read in, bins each pixel value according to its top 10 most significant bits, increments one counter per pixel, and provides a FIFO-like interface for the C&DH Packetizer module to read out the data.

The Histogram Core contains one 2^{10}-deep memory and is responsible for calculating one histogram. The Histogram Controller instantiates multiple copies of the Histogram Core, one per subregion, and dispatches each pixel value to the correct Histogram Core according to the pixel's (x, y) coordinates.

The Histogram Core module is the same in both cameras, but the Histogram Controller modules are slightly different. The SG camera instantiates the Histogram Controller module three times, once per CCD; each controller module instantiates two Histogram Core

modules, one per subregion. The CI camera instantiates only one Histogram Controller module; that module instantiates three Histogram Core modules.

The Histogram Controller provides a Wishbone interface to reconfigure the subregion boundaries.

4.1.1 Data Forwarding

The Histogram Core module is implemented by storing counters in memory. Each pixel is read and interpreted as an address into the memory. The circuitry reads the value at that address, increments it by one, and then writes back to the memory. The result is a histogram. The core circuit is shown in Figure 4.1.

A design challenge with the Histogram Core is that there is a small delay involved in accessing the memory. Every read and write takes one clock cycle to take affect.

Data enters the Histogram Core at a rate of one pixel per clock cycle. To keep up with this rate, the Histogram Core module uses both the read port and the write port of an RTG4 LSRAM memory to perform reads and writes in parallel. Unfortunately, this pipelining leads to incorrect results when two pixels of the same value are presented in sequence. The updated count value is not written to memory by the time the second value causes the module to read from the memory. The result is the count is one less than the correct value.

Rather than slowing down the Rasterizer output, the Histogram Core design overcomes this issue through the technique of data forwarding. This technique is a common optimization in pipelines, particularly in pipelined CPUs, to forward data directly to later pipeline stages as needed.

As shown in Figure 4.2, an additional register exists next to the memory whose value is accessible without delay. When two consecutive pixels have the same value, the circuit asserts `forward`, and the register value is used instead of the one read from memory.

Data forwarding can result in subtle bugs, though the Histogram Core is a simple case. Nonetheless, to ensure the Histogram Core works under all scenarios, the module is formally verified as described in Section 5.5.4.

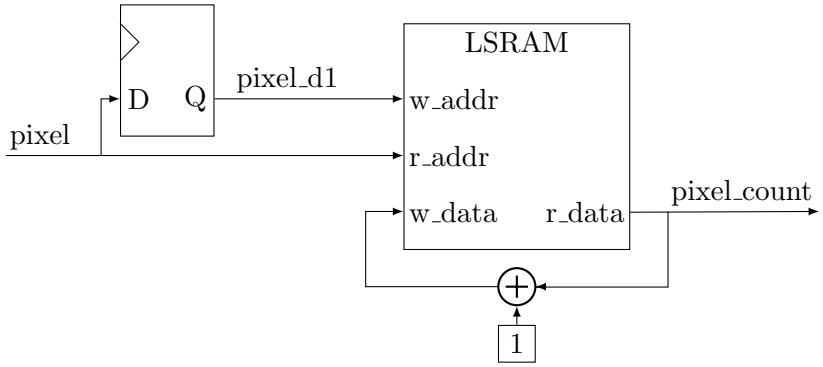


Fig. 4.1: Concept for the Histogram Core without data forwarding. Much logic is not shown, including the write and read-enable signals, initialization logic, and logic for reading out the histogram at the end of an image.

4.1.2 Initialization

The Histogram Core must also reset the entire contents of memory to zero between images. To implement this, the core contains a state machine that writes zero to every address in memory, one address at a time. This state machine clears the memory both after reset and after an image's histogram is entirely read out.

4.1.3 Readout

The Histogram Controller provides access to the histogram data via Wishbone. One address per Histogram Controller is reserved for reading out histogram data. When a histogram is complete, the packetizer reads from that Wishbone address $2^{10} \cdot S$ times, where S is the number of subregions; two on the SG camera, and three on the CI.

First, histogram values are read out from the first subregion in increasing order. Then, values are read out from the second subregion, and so forth.

The counters in the histogram memory must be large enough such that, even in the case every pixel in an image is of the same value, the count does not overflow. On the SG, counters must be $\lceil \log_2 (1072 \times 1027) \rceil = 21$ bits wide. On the CI, counters must be $\lceil \log_2 (4196 \times 2068) \rceil = 24$ bits wide. The memory width is rounded up to 32 bits. The module splits each count into two words as they are read out onto the 16-bit Wishbone bus.

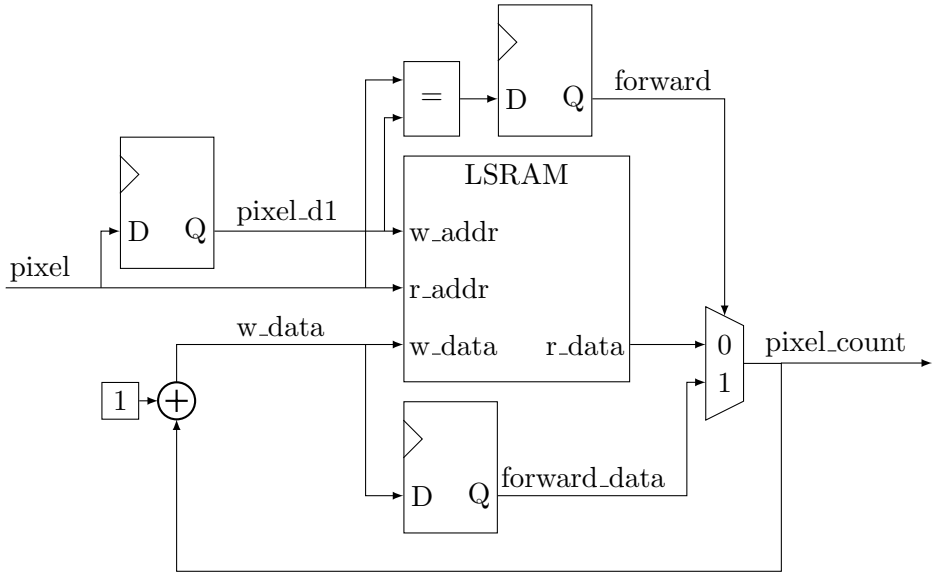


Fig. 4.2: Histogram Core forwarding logic. The `forward` signal indicates when `forward_data` should be read instead of the data read from memory. The logic behaves identically to that in Figure 4.1 except that it correctly handles consecutive pixels of the same value.

4.1.4 Cumulative Histogram

The camera is required to provide specifically a *cumulative* histogram. The value of each bin for such a histogram is the number of pixels in an image that fall in that bin *and all lower bins*. A cumulative histogram in this way is similar to cumulative distribution functions in probability theory.

The most direct solution to making the histogram design cumulative is to store and constantly update the cumulative histogram in memory. This is infeasible in practice; in the worst case where every pixel in an image is of value 0, the counters of all 2^{10} bins would have to be updated every clock cycle.

The solution is to instead store a non-cumulative histogram as before and instead calculate the cumulative histogram on the fly as it is read out from memory. The circuit performing this function is shown in Figure 4.3. The circuit keeps a running count as pixel counts are read out from the memory. This running total is added to every non-cumulative histogram value.

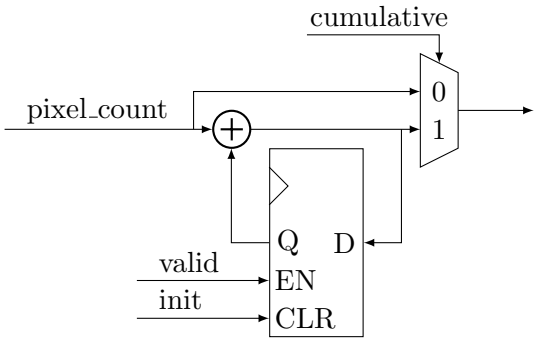


Fig. 4.3: The circuit to convert a non-cumulative histogram to a cumulative histogram. The controlling state machine asserts the `init` signal between images.

4.2 C&DH Register Bank

The Pixel Stream Header Prepender takes header data as input. This input is provided by the C&DH Register Bank. The C&DH Register Bank is very simple; it simply hosts a number of Wishbone-accessible registers.

These registers are then delayed by a configuration pipeline. The first stage of this pipeline advances when integration begins, and the second stage advances when readout begins. See Chapter 2.4 for the camera's overall strategy for latching in configuration at the correct times.

The output of this configuration pipeline is then combined with settings stored in other modules - specifically, the window size and frame serial number - to form an image header that is then sent to the Pixel Stream Header Prepender module.

4.3 Compression Register Bank

The Compression Register Bank provides Wishbone-accessible registers used by the Packetizer and the Compression Controller modules in the Compression FPGA. Originally, the Compression FPGA did not have a Wishbone crossbar, and all configuration used by the Compression FPGA was stored in the Compression Register Bank. Later design revisions added a crossbar, allowing new modules in the FPGA to contain their own configuration registers.

As this change came late in the design process, the Compression Register Bank was kept to reduce the number of changes required.

4.4 Wishbone CDC Bridge

As described in Section 2.6, the core clock domains in the two FPGAs have an unknown phase relationship and must be treated as separate clock domains.

The Wishbone CDC Bridge module is responsible for connecting the Compression FPGA's Wishbone memory bus to the C&DH FPGA's memory bus. This module implements a custom clock-domain-crossing solution. This module is formally verified as described in Section 5.5.2.

The Wishbone CDC Bridge is composed of two modules, one on each side of the clock boundary. See Figure 4.4. The module on the C&DH FPGA is termed the transmit (TX) module, and the module on the Compression FPGA is termed the receive (RX) module. This naming reflects control flow; Wishbone commands are sent from the transmitter to the receiver. The receiver then sends acknowledgements back to the transmitter.

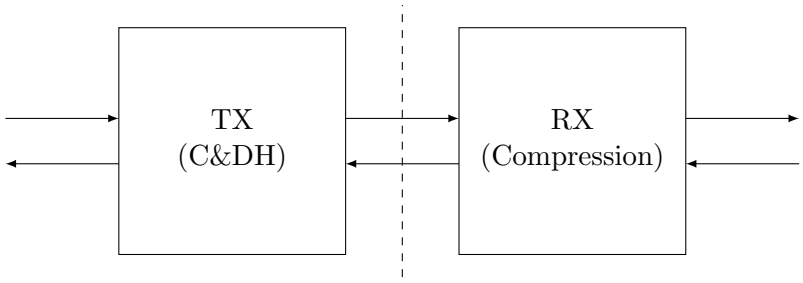


Fig. 4.4: The pair of modules composing the Wishbone CDC Bridge.

4.4.1 Synchronizers

Multiple categories of clock boundaries exist. These categories are summarized in Table 4.1.

Multiple techniques exist for synchronizing signals across a clock domain boundary. To synchronize a signal into a clock domain is to provide samples of the signal aligned to a

Table 4.1: Categories of clock domain boundaries [1, p. 474]

Category	Periodic clocks	Frequency relationship	Phase relationship
Synchronous	Yes	Identical	Identical
Mesochronous	Yes	Identical	Different
Plesiochronous	Yes	Similar	Slowly changing
Periodic	Yes	Different	Rapidly changing
Asynchronous	No	-	-

local clock. Synchronizing signals avoids metastability that could otherwise lead to circuit failure.

Which techniques apply to a problem depend on which category in Table 4.1 the clock boundary belongs to. In *Digital Systems Engineering*, Dally and Poulton provide a variety of synchronization techniques for these different scenarios, including the delay-line, two-register, and FIFO synchronizers.

Synchronizers that work for a broad category will also work for a more narrow category. For example, the synchronization method of chaining two flip-flops works for the asynchronous boundary and therefore works for all other boundaries, too.

4.4.2 Multi-Cycle Path Formulation

As the phase relationship between the two FPGAs are unknown, the clock boundary at the interface between the two FPGAs is at best mesochronous. This is because, as configured, the PLLs in each FPGA only guarantee phase relationships between its output clocks, not between output clocks and the input reference clock. It is unknown whether this phase may slowly shift over time, so it is safest to assume the clock boundary is plesiochronous. Therefore, any technique that works for plesiochronous, periodic, or asynchronous clock-domain-crossing boundaries would work for this application.

In a 2008 paper, *Clock Domain Crossing (CDC) Design & Verification Techniques using SystemVerilog*, Clifford describes a very simple clock-domain-crossing technique [6]. The approach, named the Multi-Cycle Path (MCP) Formulation, works for asynchronous boundaries.

The approach was chosen due to its simplicity compared to other synchronization methods. The approach has two important downsides discussed below.

Tradeoff: extra latency

The MCP formulation is not optimal with regard to latency. A technique such as Dally and Poulton's three-element FIFO synchronizer would reduce latency by one clock cycle. As discussed below, another register stage may be optionally added with the MCP formulation, increasing the additional cost to two clock cycles.

This additional latency of one or two clock cycles is not a concern for this Wishbone interface. This interface is only used for configuring the camera and is already extremely fast; a delay of two clock cycles is inconsequential.

Tradeoff: probability of failure

An additional issue with the MCP formulation is that it suffers the same issue all synchronizers across asynchronous boundaries suffer: a nonzero probability of failure.

In all such devices, an N-stage synchronizer consisting of N flip-flops in series, where $N \geq 2$, is responsible for sampling the signal and synchronizing it with the receiving domain. A two-stage synchronizer is shown in Figure 4.5. The first flip-flop may enter a metastable state, but that state usually resolves before the next flip-flop samples the signal at the following clock edge.

The failure occurs when the first flip-flop remains in its metastable state long enough for the second flip-flop to sample the metastable signal. Dally and Poulton provide an analysis to derive the probability of this failure [1]. This analysis is applied to the MUSE design below.

The time a flip-flop takes to exit from a metastable state is referred to as the decision time t_d. This time is exponential [1]:

$$t_d = -\tau_s \log\left(\Delta V_1\right) \tag{4.1}$$

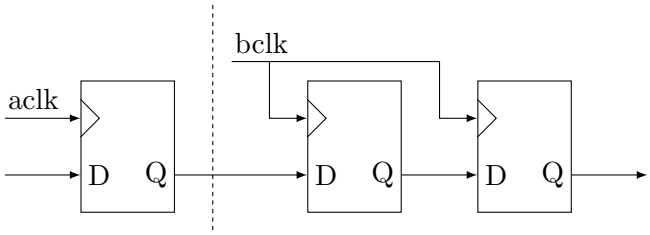


Fig. 4.5: A two-stage synchronizer. The dashed line represents the clock domain boundary.

In Equation 4.1, τ_s is a circuit-dependent constant named the regeneration time, and ΔV_1 is the voltage at the output of the NAND-latch inside the flip-flop the instant it registered the signal.

Assuming the received signal has a perfectly linear rise time, then if the receiving clock edge happens to fire as the sampled signal is transitioning, ΔV_1 is uniformly distributed. For the sake of simplicity, this analysis assumes ΔV_1 is uniformly distributed between 0 and 1.

If a synchronization failure occurs when the decision time exceeds some time t_w, then the probability of a synchronization failure is:

$$P_{sf} = P_b P\left(t_d \geq t_w\right) \tag{4.2}$$

$$= P_b P\left(-\tau_s \log\left(\Delta V_1\right) \geq t_w\right) \tag{4.3}$$

$$= P_b P\left(\Delta V_1 \leq \exp\left(-\frac{t_w}{\tau_s}\right)\right) \tag{4.4}$$

$$= P_b \exp\left(\frac{-t_w}{\tau_s}\right) \tag{4.5}$$

In Equation 4.2, P_b is the probability the signal transition lies in the aperture of the receiving latch. The aperture time t_a is the sum of the setup and hold times of the latch. In the worst case for an RTG4 FPGA with speed grade -1 and with SET mitigation enabled, the setup time is 1.523 ns and the hold time is 0.104 ns, resulting in $t_a = 1.627$ ns [7]. Let T_a be the period of the transmitting register's clock; in the case of MUSE, $T_a = 25$ ns. Then, assuming the edges of one clock is uniformly distributed across one cycle of another clock, $P_b = \frac{t_a}{T_a} = 0.065$.

The Mean Time Between Failures (MTBF) is used often to determine whether the risk of metastability is acceptable. In this case,

$$\text{MTBF} = \frac{1}{f_b P_{sf}} \tag{4.6}$$

where f_b is the frequency of the receiving register's clock. For MUSE, $f_b = 40$ MHz.

The regeneration time τ_s is not provided in the device datasheet, but a typical value for FPGAs fabricated in a 65 nm process like the RTG4 are 100 ps [8]. As a conservative estimate, $\tau_s = 200$ ps is used.

With a two-stage synchronizer, the signal must stabilize before the second flip-flop latches in the value. The clock period is $\frac{1}{f_b} = 25$ ns. To account for propagation delay and to add margin for other factors, let $t_w = 20$ ns. Then, MTBF $\approx 10^{29}$ years.

Note this analysis is flawed. It was incorrectly assumed the edges of one clock are uniformly distributed in the cycle of the other clock. As described in Section 2.6, the two clocks have a fixed but unknown phase relationship; this relationship might vary every time the device is powered on, but the probability distribution is unknown. Applying this standard MTBF estimation technique under this false assumption can produce overly optimistic results [9]. Therefore, a third flip-flop stage is added to be safe.

Description of the MCP Formulation

The MCP formulation builds on the N-stage synchronizer shown in Figure 4.5 to build a *pulse transmitter* and *pulse receiver* pair. This circuitry allows a one-clock-cycle pulse in one clock domain to result in a one-clock-cycle pulse in another domain, regardless of the frequency or phase relationship of the two clocks.

During operation, pulses in the first clock domain cause the CDC signal to toggle. This CDC signal is synchronized into the receiving domain, and a dual edge detector sees the signal toggle and outputs a pulse in the second clock domain. The circuit is shown in Figure 4.6.

In the MCP formulation, a pulse in the sending clock domain causes a register to latch

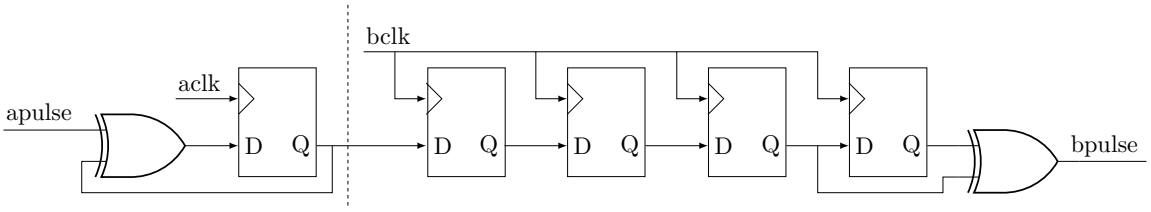


Fig. 4.6: Pulse transmitter/receiver pair with a three-stage synchronizer. A one-clock-cycle pulse on *apulse* results in a one-clock-cycle pulse on *bpulse*.

in the data to be sent. This register's output is the CDC signal. This pulse is then delivered to the receiver as described above. On receipt of the pulse, the register in the receiving clock domain latches in the CDC signal. See the circuit in Figure 4.7.

The pulse signal is delayed by at least $N - 1$ clock cycles. Therefore, the CDC data signal will certainly be stable by the time the receiving domain latches in the data, avoiding metastability.

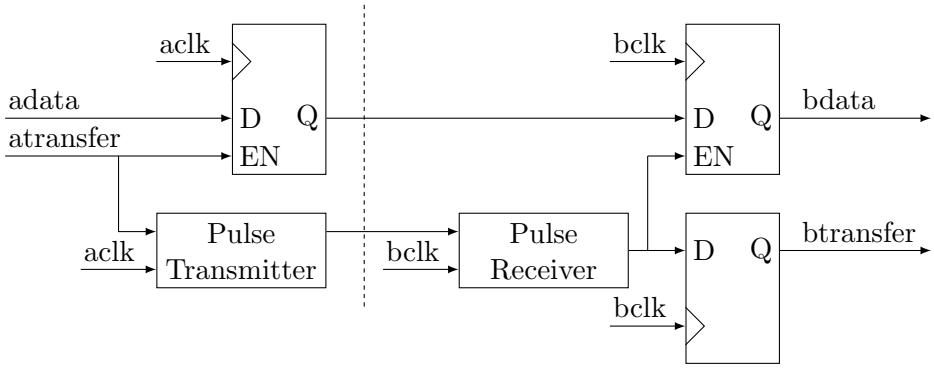


Fig. 4.7: The MCP formulation used to transmit data across a clock boundary.

4.4.3 Application of the MCP Formulation

The Wishbone CDC module contains two instances of the MCP formulation.

One instance, termed the command channel, allows the C&DH FPGA to issue a Wishbone request to write to or read from a register on the Compression FPGA. The data sent across this channel include the register address, the write-enable flag, and the write data.

The other instance, termed the acknowledge channel, allows the Compression FPGA to respond to commands. The read data is sent across this channel.

The entire design is shown in Figure 4.8.

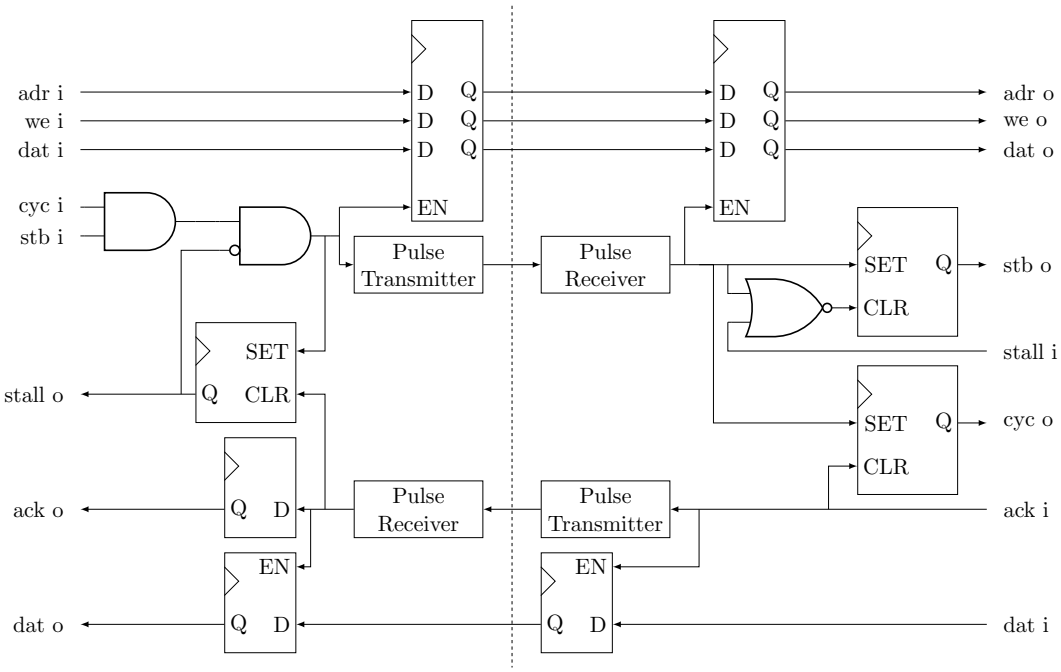


Fig. 4.8: The Wishbone CDC Bridge. The dashed line represents the clock boundary. The transmitter is on the left side of the boundary; the receiver is on the right. Clock signal connections to registers are implied.

The MCP formulation guarantees the receiving registers never latch in data until that data has been stable for multiple clock periods. The formal verification of this module pair proves this property is invariant. Nonetheless, one extra register was later added to each receiving register out of an abundance of caution. The extra delay caused by this register does not affect the module pair's behavior.

4.5 Single Event Detection

When electronics are flown in space, they are exposed to significantly higher levels of radiation. This radiation may cause the electronics to malfunction. There are a variety of mechanisms by which radiation can cause device failure; these effects are collectively referred

to as Single Event Effects (SEEs). A single-event effect can cause functional interrupts or hardware failure.

Special radiation-hardened and radiation-tolerant devices are typically used in space-flight electronics for this reason. These devices are less susceptible to the effects of radiation. For example, the FPGAs used in the MUSE design, the Microchip RTG4, are radiation-tolerant.

However, the MUSE system has a performance requirement that necessitates using a specific ADC, the Analog Devices AD9257. Due to concerns about this device's susceptibility to radiation, the Parts Control Board overseeing the MUSE mission requires the program to implement SEE mitigations for this device.

This ADC measures pixel values from the CCD. It is referred to as the science ADC to distinguish it from other ADCs on the camera, referred to as the housekeeping ADCs. The housekeeping ADCs monitor temperatures, voltages, and currents throughout the camera; unlike the science ADC, they are radiation-tolerant and do not need special mitigation.

The purpose of the Single Event Detection module is to implement SEE mitigation for the science ADC.

4.5.1 Monitoring for Single-Event Effects

The Single-Event Detection module uses two sources of information to detect when a harmful SEE occurs.

The first source of information is the current the ADC is consuming. As mentioned previously, auxiliary housekeeping ADCs exist in the circuitry. Two of the values the housekeeping ADCs measure are the currents on the ADC's digital and analog power rails. A harmful SEE may cause the ADC to draw more current than expected.

The second source of information is the pixel output of the ADC. With every image, there are points along the CCD output waveform that should always have a predictable value. The ADC reporting these pixels to be outside the expected range indicates a fault that may be an SEE.

The Single Event Detection module handles this information through a circuit called the event counter. The event counter is repeated eight times on the SG: twice for the power rails, and six times for the six taps. The event counter is repeated four times on the CI: twice for the power rails, and twice for the two taps. Each instantiation is referred to as a "channel".

First, the event counter determines whether a given sample is out of bounds. These bounds are set via Wishbone-accessible registers. If a sample is out of bounds, and if the camera is configured to monitor this channel, then the out-of-bounds counter for that channel increments by one. Once the out-of-bounds counter exceeds a user-defined threshold, the module determines an SEE has occurred.

Rare, spurious events are expected to occur over time. To prevent these rare events from slowly incrementing the out-of-bounds counter until it exceeds the count threshold, a watchdog timer occasionally clears the out-of-bounds counter. The watchdog timer counts up once each clock cycle, resetting only when an out-of-bounds sample was detected.

4.5.2 Handling Single-Event Effects

The Single Event Detection module performs a sequence of actions to power cycle the ADC. This sequence is implemented via the state machine in Figure 4.9.

First, the state machine remains idle in the wait state. Once an SEE is detected, the FSM enters the prepare state.

In the prepare state, the Single Event Detection module signals to the ADC Interface module to set its output to high-impedance. This step is taken as a safety measure to ensure the FPGA does not assert control signals to the ADC while the ADC is powered off.

Once the ADC Interface module indicates its outputs have been set to high-impedance, the FSM moves to the power-off state. In the power-off state, the FPGA signals to the power supply circuitry to remove power to the ADC. The FSM remains in this state for one second.

Once one second has passed, the FSM moves to the start-up state. During this state, the FPGA signals to the power supply circuitry to reapply power to the ADC. During

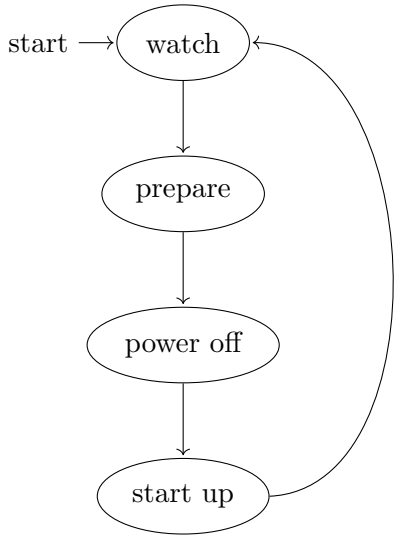


Fig. 4.9: A high-level diagram of the Single Event Detection module's state machine.

this state, the Single Event Detection module still instructs the ADC Interface module to maintain its outputs in a high-impedance state.

Finally, once another second has passed, the FSM re-enters the watch state. The Single Event Detection module allows the ADC Interface to continue communications with the ADC as normal, and the Single Event Detection module begins monitoring for SEEs once again.

4.6 Heater PWM

The Heater PWM outputs a pulse-width modulated (PWM) signal to control camera heaters.

The design is extraordinarily simple. The entire circuit, excluding the Wishbone bus module and bus-accessible configuration registers, is shown in Figure 4.10.

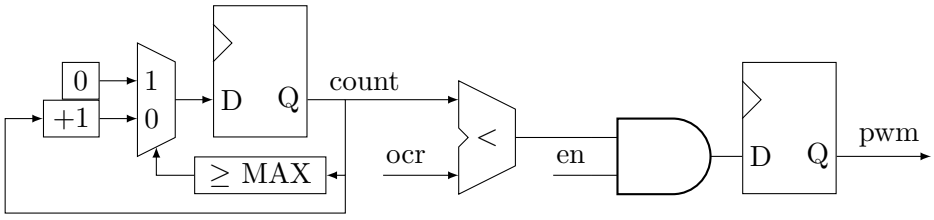


Fig. 4.10: The Heater PWM circuit.

There are two configuration registers: the enable register, and the output compare register (`ocr`).

The heater PWM module works via an incrementing counter. The counter begins at zero and counts up to and including MAX, after which it resets to zero. The PWM output is high when the count is less than `ocr`, and it is low otherwise. See Figure 4.11.

Valid values for the output compare register are $0 \leq \texttt{ocr} \leq \texttt{MAX} + 1$. The larger `ocr` is, the higher the duty cycle of the PWM output. When $\texttt{ocr} = 0$, the duty cycle is 0%. When $\texttt{ocr} = \texttt{MAX} + 1$, the duty cycle is 100%.

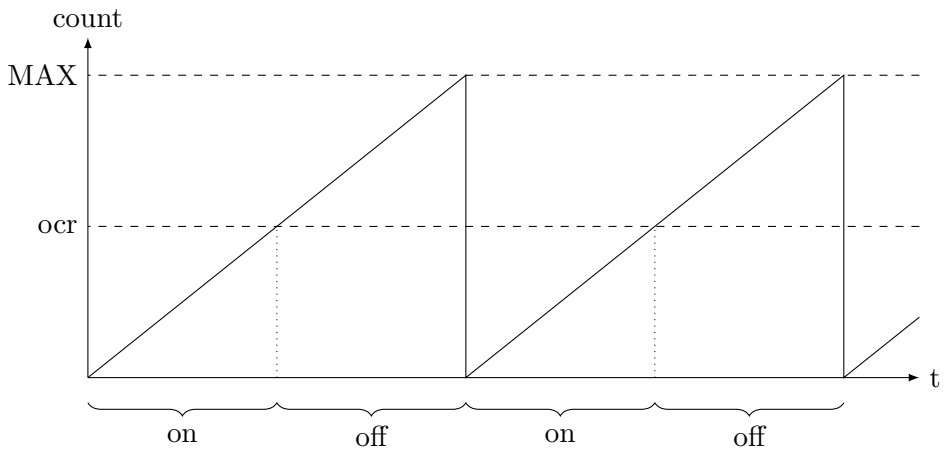


Fig. 4.11: Concept for the Heater PWM.

`MAX` is a design constant that determines f_{pwm}, the frequency of the PWM signal.

$$f_{pwm} = \frac{f_{clk}}{\texttt{MAX} + 1} \tag{4.7}$$

The circuitry was designed for a frequency $f_{pwm} = 500$ kHz, and the FPGA clock frequency is $f_{clk} = 40$ MHz. So, the constant was chosen to be $\texttt{MAX} = 79$.

4.7 Housekeeping ADC Interface

The MUSE camera is required to report various analog values. The specific analog

values differ between the two cameras but generally include the voltage and current of various power rails and temperatures at various points on the hardware. The Housekeeping ADC Interface module is responsible for reporting these analog values. It does this by interfacing with three ADCs in the camera hardware. These ADCs are the 8-channel, 12-bit ADC128S102 made by Texas Instruments.

Of particular interest are the current of the two power rails powering the science ADC, the AD9257. As discussed in Section 4.5, the mission is required to monitor the functionality of this ADC to be permitted to use it. This is the responsibility of the SEE Detection module. That module monitors these currents provided by the Housekeeping ADC module.

Similar to the Histogram modules, the Housekeeping ADC Interface consists of a core module and a controller module. The core module interfaces with one ADC, and the controller module instantiates three copies of the core module and provides access to the measured values via the Wishbone bus.

The core module is an iteration of an existing design. As mentioned previously, the scope of this project includes the modification of this module to implement polling. Other aspects of the core module's design were changed, too, including timing of the SPI interface and coadding behavior. The controller module is an original design; the scope of this project includes the design and verification of this module.

4.7.1 Interface with the ADC

The Housekeeping ADC Interface module uses an existing design to interface with the ADC128S102. This work is therefore outside the scope of the report and is only briefly mentioned here for context.

Data acquisition through the ADC128S102 occurs through a Serial Peripheral Interface (SPI). Each transaction takes 16 cycles of the SPI's clock signal. The ADC is clocked at 10 MHz, meaning one transaction requires 64 cycles of the camera's 40 MHz clock. An extra two cycles are added at the start and end of each transaction, resulting in a total of 68 cycles required for one transaction, or 1.7 µs.

During one SPI transaction, data is transferred both to and from the ADC. The data received from the ADC is one 12-bit sample, and the data sent to the ADC is the channel the ADC should measure for the next SPI transaction.

The module performs coadding to reduce noise and increase the effective resolution of measurements. The module measures each channel 32 times and sums all measurements. The sum is 17 bits; to fit in the camera's 16-bit word width, the module drops the least significant bit.

One measurement requires 33 SPI transactions. The first transaction is required to send the channel number to the ADC. The data acquired during this transaction is ignored. Then, the other 32 transactions each take a sample of the channel. One measurement takes a total of 56.1 µs.

4.7.2 Polling

As mentioned above, the final design builds upon an earlier design to interface with the ADC. The earlier design took measurements from the ADC on demand. However, the current on the rails powering the AD9257 must be continuously monitored. For this reason, the Housekeeping ADC Interface polls the housekeeping ADC periodically.

To measure all 8 channels takes 448.8 µs. This means the ADC can be polled no faster than 2.2 kHz. The ADC measuring these critical values is polled near this limit at a rate of 2 kHz. The other two ADCs are polled at only 4 Hz. A set of registers, one per channel, store the last measured value. The controller module provides read access to these registers on the Wishbone bus and forwards the AD9257 current measurements separately to the SEE Detection module.

CHAPTER 5

IMPLEMENTATION AND VERIFICATION

Previous chapters describe the design of the camera FPGA. This chapter discusses how the design was implemented and verified.

Development was a continuous, iterative process of writing VHDL source code, producing documentation, and verifying the design. This documentation includes circuit diagrams, waveforms, explanatory figures, and a memory map defining the address of each configuration register.

Initially, development effort focused mostly on development of new features. As the project progressed, effort was gradually focused on refining the design and resolving issues.

5.1 Tools

The design and verification of the MUSE FPGA design required multiple software tools. The source HDL was written in VHDL-2008. Microchip's Libero SoC software synthesizes, places, and routes the design for the RTG4 FPGA. See Figure 5.1.

Git, an open-source version control program, manages the source HDL and preserves development history. Throughout the project, Git simplified the team's workflow by automatically merging changes.

Yices, Yosys, and SymbiYosys perform formal verification. Performing formal verification with these tools on VHDL source required the open-source VHDL simulator GHDL and a plugin to connect GHDL and SymbiYosys.

Siemens QuestaSim performs simulation. For the modules within the scope of this report, testbenches were written in VHDL, the same language as the rest of the FPGA design. Some modules outside the scope of this report were tested using CocoTB, an open-source tool that allows testbenches to be written in Python.

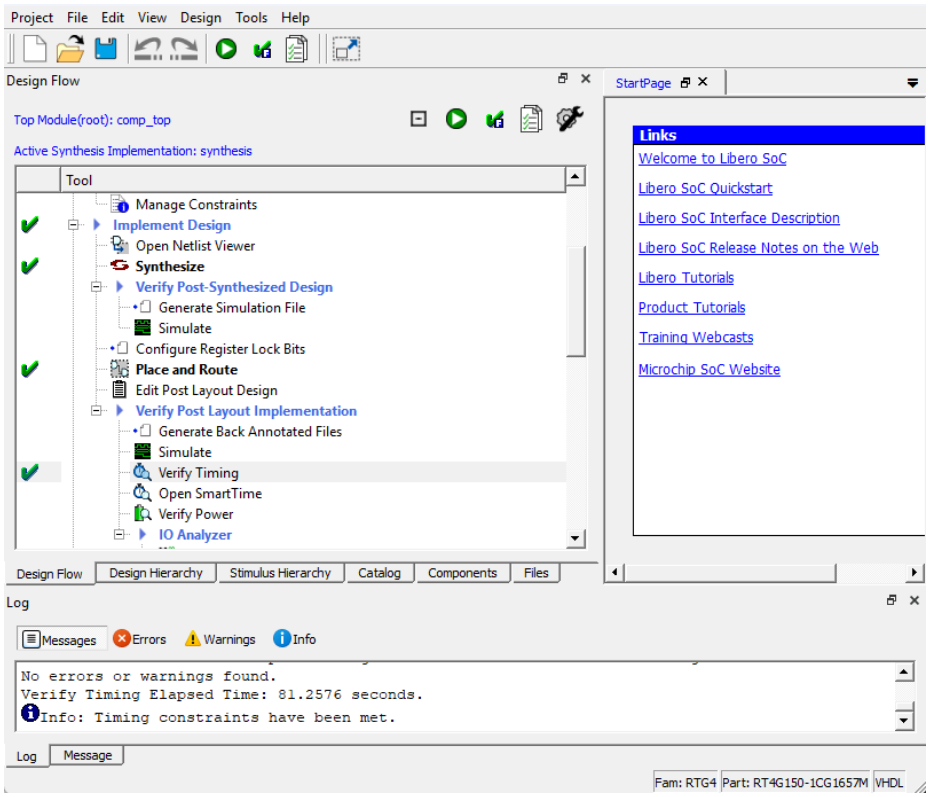


Fig. 5.1: The Microchip Libero SoC software.

Software was written to support the development of the MUSE FPGA design. including a calculator program shown in Figure 5.2, a packet parser program shown in Figure 5.3, and integration tests described in Section 5.7. These programs were written in Rust. The integration tests in particular require highly efficient software for which Rust is a good fit. Rust's features and open-source libraries made writing this software convenient.

5.2 Emulator

The design was continually tested during development. Much verification could be performed purely in software via simulation or formal methods. Some testing, including integration tests, required hardware to test on.

Camera hardware was not always available to test on as it was developed in parallel with the FPGA design. When camera hardware was unavailable, the FPGA design was tested on a Microchip RTG4 development kit shown in 5.4.

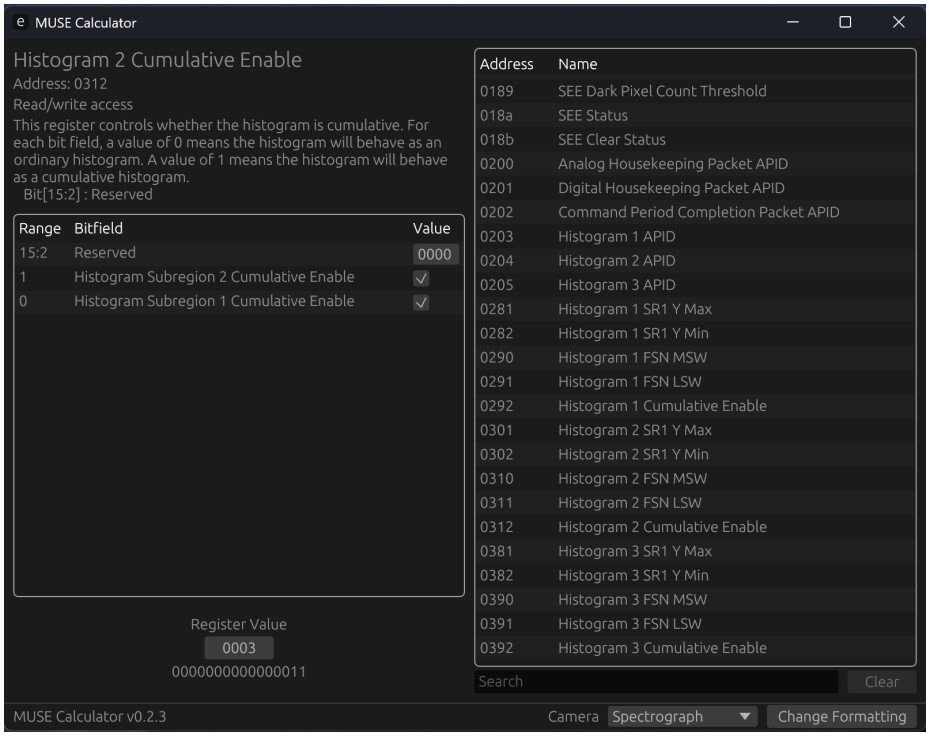


Fig. 5.2: The calculator program written to make writing register values more convenient.

The firmware developed for the camera hardware would not work with the development kit due to hardware differences. For example, the emulator only has one FPGA instead of two. A special version of the firmware, named the emulator, was developed for this testing. The emulator firmware instantiates the top-level modules of both the C&DH and Compression FPGAs. There are two variants of the emulator: the SG emulator, and the CI emulator.

To fit the entire design on one FPGA, the emulator excludes the compression core. As a consequence, the emulator can only produce uncompressed images. The camera hardware is still required to test image compression.

The development kit contains DDR3 memory modules in contrast to the camera hardware's DDR2 memory. Using this memory required additional development effort. While the development kit contains two DDR3 memory interfaces, the emulator only used one to match the camera hardware design.

As the development kit does not have the CCDs or ADCs present on the camera

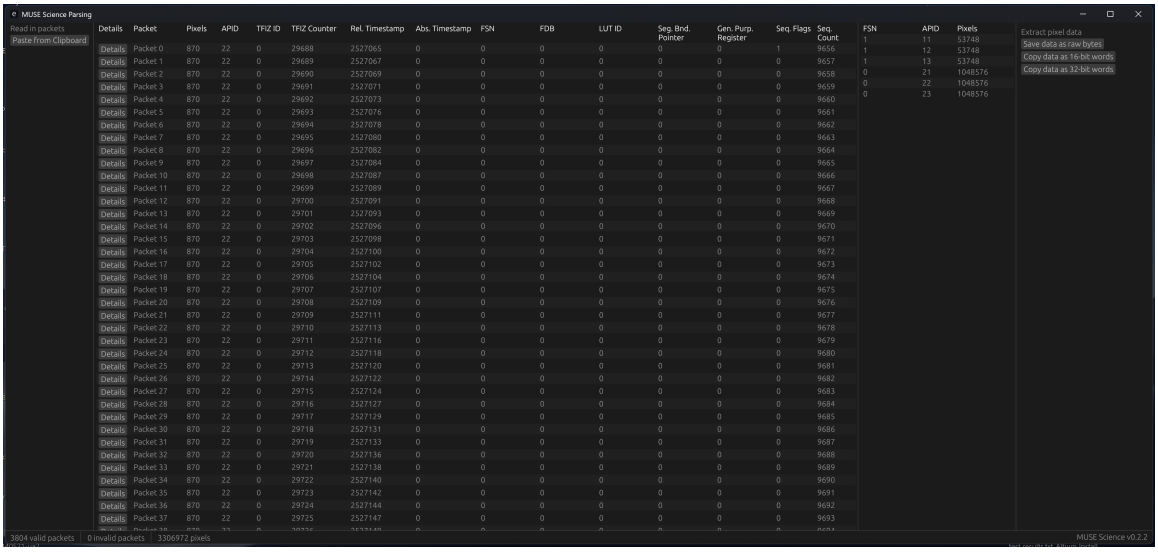


Fig. 5.3: The packet parser program written to inspect image packets produced by the camera.

hardware, all acquired images have pixel values of exactly zero. For testing the design, it is useful for pixels to be distinguishable. The Science ADC Interface module has a test mode that replaces each pixel value with a pattern where each pixel value is one greater than the previous pixel. This pattern of linearly increasing pixel values is named the ramp pattern and was used often to test the camera design on the emulator.

Figure 5.5 shows a screenshot of the ramp pattern. This image is displayed using software written by another engineer.

5.3 Source Control and Organization

One difficulty encountered throughout the design process was managing all of the targets. With the addition of the emulator designs, there are a total of six unique FPGA designs:

- SG C&DH FPGA
- SG Compression FPGA
- CI C&DH FPGA

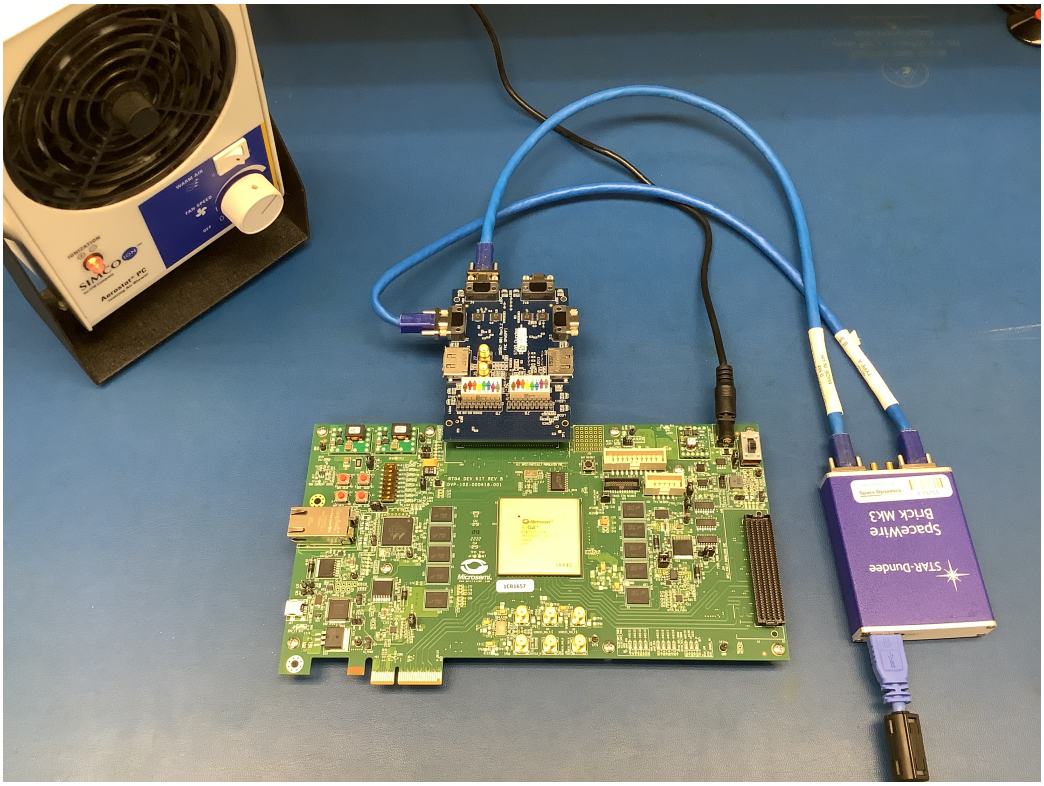


Fig. 5.4: The emulator, consisting of the RTG4 development kit and a SpaceWire interface board.

- CI Compression FPGA
- SG emulator
- CI emulator

Initially, the VHDL source was divided into multiple repositories: one for the C&DH FPGAs, one for the Compression FPGAs, one for modules common to both FPGAs, and one for the emulators. Git Submodules were used to share code between the repositories.

Later, all source code was combined into one Git repository. Keeping the design in one repository was found to reduce development friction, as it simplified the structure and removed the need to keep separate repositories correctly synchronized.

5.4 Approach to Verification

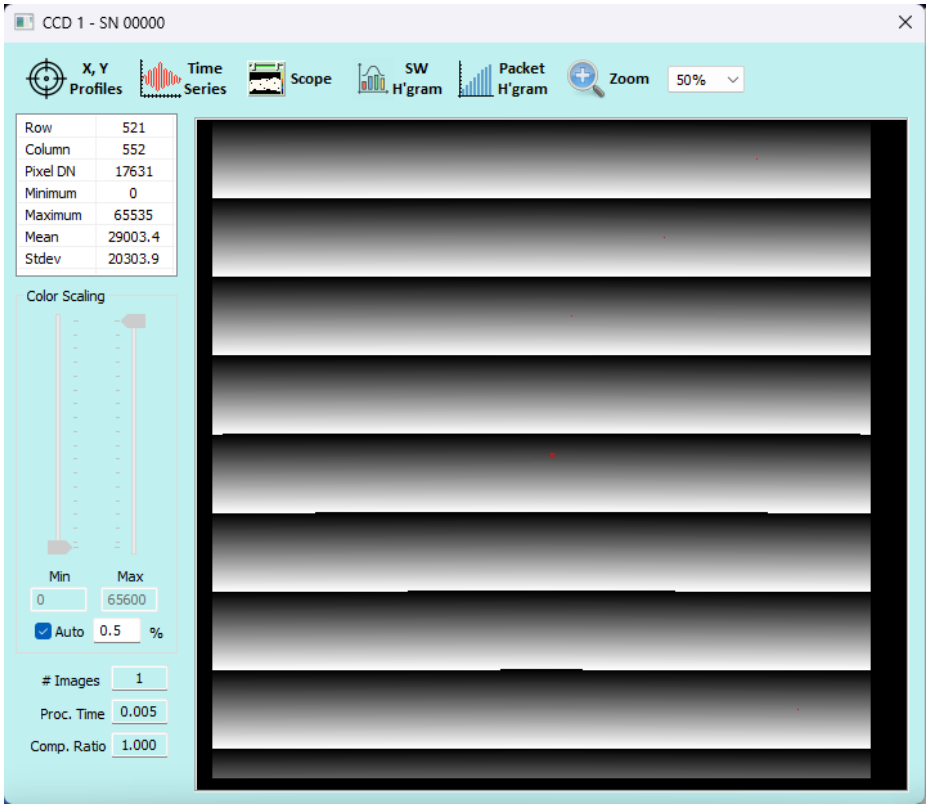


Fig. 5.5: The ramp pattern used for testing.

It is important every module in the camera works as expected and meets requirements. Verification of hardware designs is a major challenge projects must solve to ensure the design works as expected.

The MUSE camera design is verified using a combination of formal verification, simulation, and top-level integration testing. The chosen approach depends on the scope of the subsystem under test. Very small subsystems are verified through formal methods, larger subsystems are verified through simulation, and the system at large is verified via integration testing.

5.5 Formal Verification

Formal verification, when achieved, gives a level of confidence in the correct operation of a module unmatched by other forms of verification. Formal methods ensure there is no scenario in which the module enters a bad state. This is in contrast to simulation, which

only demonstrates the module remains in a good state under a small number of scenarios. A formally verified module can only fail if the written formal properties do not accurately describe correct behavior, if an assumption made during verification is incorrect, or if a bug in the tools cause a faulty design to pass.

Unfortunately, formal verification can be difficult and time consuming. This cost increases exponentially as the verified subsystem grows in complexity. This difficulty arises largely due to a combinatorial explosion in the number of possible states a subsystem may be in. It is possible to create larger formally verified systems, but practical considerations limit this powerful verification approach to a few small, critical modules in the camera design. These modules include:

- Histogram Core
- Line Rasterizing Buffer
- Wishbone CDC Bridge

Two additional modules were formally verified, but these modules were removed from the design in later revisions.

A weaker form of verification, known as the bounded model check (BMC), ensures safety properties hold only for a finite number of clock cycles after reset. The following modules are verified via a BMC:

- Ping-Pong Buffer
- Rasterizer

Unfortunately, this setup results in limitations. The most significant limitation is that assertions that relate a signal in a module to a signal in a submodule are not possible. This limitation largely prevents formal verification of any modules that contain submodules. This is why the Rasterizer and Ping-Pong buffer are only verified via BMC.

5.5.1 Writing Properties

Formal verification using these open-source tools requires an understanding of how the tools work. Let P be the module's safety properties. The tools show P is invariant, meaning P holds for all reachable states, by proving two statements:

- P holds for the first k steps after reset, and
- if P holds for k consecutive steps, then P holds for $k+1$ steps.

This process is known as k-induction, a more general form of proof by induction. The two statements are known as the base case and the inductive step.

The safety properties are written in the Property Specification Language (PSL), a language designed for formal verification of hardware designs. PSL supports expressing properties, including properties that span time, and asserting or assuming the properties hold.

When these modules were formally verified, the first step was to write the properties that describe the module's desired behavior. For example, the formal proof of the Wishbone CDC module asserts if a Wishbone write transaction is presented on the transmitting side, the write transaction will be forwarded to the receiving side with the same address and data.

At this stage, induction usually fails. This is because the formal properties are not strict enough, allowing induction to start in an unreachable state. The process of adding assertions to cause the inductive step to pass is most often the bulk of the work required in formal verification [10].

For example, a formal verification of an N-deep FIFO will likely require an assertion the FIFO's internal counters never exceed N. While states where the counters exceed N may be impossible to reach from any valid state, if these states are not asserted as invalid, the inductive step will explore starting from these states and will quickly find states that violate properties.

Assumptions are also usually required. Even when not required, assumptions ease the verification process. Incorrect assumptions, however, can entirely void a proof. For

this reason, modules formally verified in the MUSE design only contain assumptions about inputs, never about internal state or outputs. One important assumption made in all the formal proofs is that the reset signal is asserted on the first formal timestep.

5.5.2 Formal Verification of the Wishbone CDC Bridge

The Wishbone CDC Bridge, being a clock-domain-crossing module, is an excellent candidate for formal verification, since clock domain crossing interfaces can be tricky to design correctly. The most important property proven in the formal verification is the CDC safety assertion. These assertions state data is always stable for long enough before the receiving flip-flops sample the data, avoiding metastability.

Multiple Clock Domains

The presence of multiple clock domains introduces unique challenges for the formal verification of this module.

The Yosys toolchain provides a `multiclock` setting to allow for this type of formal verification. When this setting is enabled, the tool provides a *formal timestep* as a global clock to access during verification. The behavior of the clocks can then be described relative to this formal timestep via assumptions. One approach, as demonstrated by Gisselquist, is to assume an arbitrary, finite period for each clock [11]. Such an assumption produces two periodic clocks with no frequency or phase relationship.

However, as described in the design of the Wishbone CDC Bridge, the MCP formulation should work even without periodicity. An even weaker assumption is possible. The only assumption made about the clocks in this verification is that they never have a span of more than ten formal timesteps without a clock edge. An example of the sort of clock waveforms allowed under these assumptions is shown in Figure 5.6.

Further assumptions are required that are not normally seen in the formal verification of modules. For example, it is required to assume the interfaces between each module and other circuitry within the same clock domain only change on the rising edge of that domain's clock.

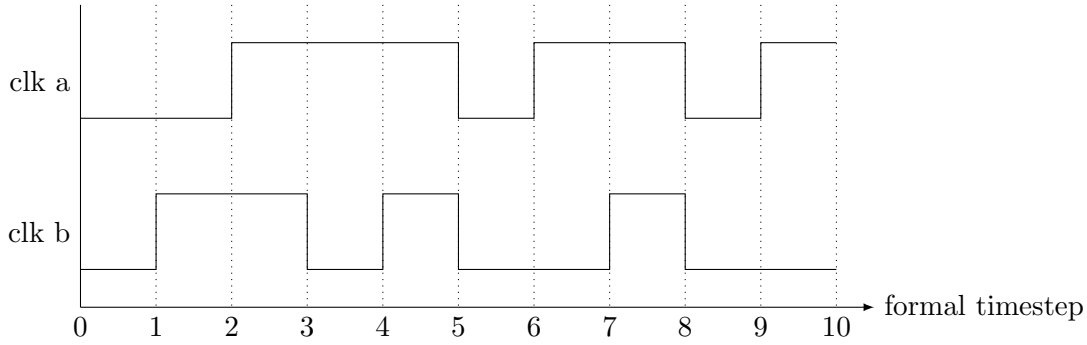


Fig. 5.6: An example of clock waveforms valid under under the given assumptions.

Formal State Machine Properties

The goal of this formal verification is to prove two categories of properties. The first category is CDC safety; specifically, the data signals crossing the clock domains are only latched in by the receiving clock domain once those signals have settled for some time. The second category is data integrity, or that the Wishbone interface works correctly and data and addresses are correctly delivered.

Many properties were made in the context of a state machine that exists only as part of the formal proof. Formal verification of these modules often includes these auxiliary state machines to describe the flow of state over time and to assert properties conditional on specific states. This state machine, shown in Figure 5.7, is unique in that some state transitions occur on the edges of one clock, while other transitions occur on edges of the other clock.

When in the reset state, this state machine transitions to the idle state on the first formal timestep the reset signals to both modules are low.

When in the idle state, this state machine transitions to the transmitting state on the rising edge of the *transmitter*'s clock when the transmitter's cycle and strobe inputs are both high and the stall output is low.

When in the transmitting state, this state machine transitions to the transmitted state on the rising edge of the *receiver*'s clock when the receiver's cycle and strobe outputs are both high and the stall signal is low.

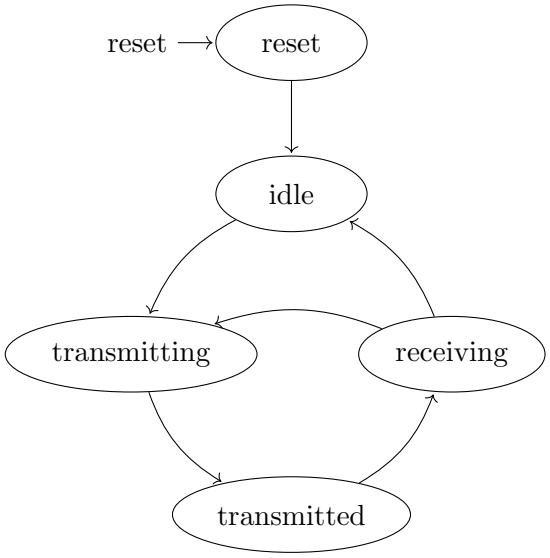


Fig. 5.7: The formal FSM used in the formal verification of the Wishbone CDC Bridge.

When in the transmitted state, this state machine transitions to the receiving state on the rising edge of the *receiver*'s clock when the receiver's ack input is high.

When in the receiving state, this state machine transitions on the rising edge of the *transmitter*'s clock when the transmitter's ack output is high and the stall output is low. If the transmitter's cycle and strobe inputs are both high, it transitions to the transmitting state; otherwise, it transitions to the idle state.

With the auxiliary state machine prepared, many assertions were made according to the current state to describe the behavior of the module pair over time.

To track Wishbone transactions, two values, T_a and T_b, track the number of in-flight Wishbone transactions on the transmitter and the receiver, respectively. These counts increment by one when a Wishbone transaction begins and decrements once the transaction ends through assertion of the Wishbone acknowledge signal.

To ensure data remains stable at the correct times in the MCP formulation, two values, ΔC_a and ΔC_b, track the number of value changes across the module's three-stage synchronizers. For example, if the registers contain the sequence 0, 1, 1, 0, then $\Delta C = 2$. Intuitively, if $\Delta C \neq 0$, then the synchronizer's edge detector will soon detect a signal transition and output a one clock-cycle pulse.

Similar to the auxiliary state machine, both T and ΔC are variables that exist only during formal verification. Properties assert the values of these variables according to the current formal state. These values are shown in Table 5.1.

Table 5.1: Asserted values for the Wishbone CDC module

State	T_a	T_b	ΔC_a	ΔC_b
Reset	0	0	0	0
Idle	0	0	0	0
Transmitting	1	0	1	0
Transmitted	1	1	0	0
Receiving	1	0	0	1

Clock-Domain Crossing Safety Properties

Define a Wishbone command to consist of three signals: the address, the write-enable flag, and the write data. Let C_{tx}, C_{cdc}, and C_{rx} be these signals at the input to the transmitter, at the clock boundary, and at the output of the receiver, respectively.

Similarly, define a Wishbone acknowledge to consist of one signal: the read data. Let A_{rx}, A_{cdc}, and A_{tx} be this signal at the input to the receiver, at the clock boundary, and at the output of the transmitter, respectively.

Let p_b and p_a be the outputs of the receiver's and the transmitter's synchronizers, respectively. These signals pulse high for one clock cycle.

In the following assertions, signals are indexed with n and m. Let n be a discrete time index that increments once per clock cycle of the transmitter. Similarly, let m be a discrete time index that increments once per clock cycle of the receiver.

The formal verification proves these properties are invariant:

- If $p_b(m)$ is high, then $C_{cdc}(m) = C_{cdc}(m-1) = C_{cdc}(m-2) = C_{cdc}(m-3)$.
- If $p_a(n)$ is high, then $A_{cdc}(n) = A_{cdc}(n-1) = A_{cdc}(n-2) = A_{cdc}(n-3)$.

Note that p_b being high is the condition for the receiver to latch in C_{cdc}, and p_a being high is the condition for the transmitter to latch in A_{cdc}. Therefore, the above two assertions

verify the important property that whenever either module latches in data, that data has had at least three clock cycles in the receiving module's clock for that data to settle.

This verification ignores signal propagation delays. If the clock-domain-crossing request or acknowledge signals arrive multiple clock cycles earlier than the data, then the results of this verification are invalid. Timing constraints ensure this never occurs.

Data Integrity Properties

To verify data is always correctly delivered from the transmitter to the receiver, the formal verification proves these properties are invariant:

- If $C_{tx} = x$ at the last time the cycle and strobe signals were asserted to the input to the transmitter without stall asserted, and if the formal FSM is in the transmitting state, then $C_{cdc} = x$. Furthermore, if the receiver is asserting the cycle and strobe signals, then $C_{rx} = x$.

- If $A_{rx} = y$ at the last time the acknowledge signal was asserted to the input to the receiver, and if the formal FSM is in the receiving state, then $A_{cdc} = y$. Furthermore, if the transmitter is asserting the ack signal, then $A_{tx} = y$.

5.5.3 Formal Verification of the Line Rasterizing Buffer

The LRB lies at the heart of the camera. Subtle bugs in the LRB would be difficult to notice and destructive to images. Formal verification of the LRB helps reduce this concern.

Formal verification of FIFO-like buffers are subtle. Verification of the LRB is easier than a FIFO, however, since its two-phase semantics are simpler and do not require any wrapping pointers as is often seen in FIFO designs. An auxiliary state machine shown in Figure 5.8 represents whether the LRB is in the fill or the drain phase. This FSM is only present during formal verification.

The two-phase semantics are defined with assumptions dependent on this state. The LRB enters the drain phase once data is first read from the buffer. During the drain phase, only reads from the LRB are permitted until the buffer is empty.

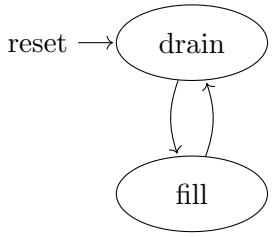


Fig. 5.8: The formal FSM used in the formal verification of the Line Rasterizing Buffer.

When the buffer is empty and data is written to the buffer, the LRB enters the fill phase. Both writes and reads are permitted during the fill phase. Once any read occurs, the LRB immediately transitions to the drain phase.

Data Integrity and Ordering Properties

The formal verification proves these properties are invariant:

- If a module writes A, then B, to the forward port, the LRB will read out A, then B, from the output before the LRB is empty.
- If a module writes A, then B, to the reverse port, the LRB will read out B, then A, from the output before the LRB is empty.
- If a module writes A to the forward port and B to the reverse port in any order, the LRB will read out A, then B, before the LRB is empty.

Only four assumptions were required:

- The reset signal is initially asserted.
- Once a module has started reading from the LRB, the module must read the entire contents of the LRB before writing any more data.
- No more than N_C values may be written during one fill phase, where N_C is defined in 3.1.1.
- If n values were written during the fill phase, then no more than n values may be read out during the corresponding drain phase.

Formally specifying the data integrity and ordering properties follows the same pattern as a typical FIFO formal verification. The Yosys toolchain provides an `anyconst` attribute for this scenario. This attribute can be used to form properties against arbitrary constants; for example, a property may state when an arbitrary value x is written to the LRB, then that same arbitrary value will be read out from the LRB before it is empty.

For the inductive step to pass, additional assertions were required to constrain the values of the three LRB pointers. For example, one simple assertion states that at all times, $\texttt{fr} \leq \texttt{fw} < \texttt{s}$.

5.5.4 Formal Verification of the Histogram Core

The Histogram Core module is formally verified because it contains a tricky piece of data-forwarding logic to allow the module to accept new data each clock cycle.

To formally verify the Histogram Core, an auxiliary state machine, shown in Figure 5.9, was made to track the state of the module over time.

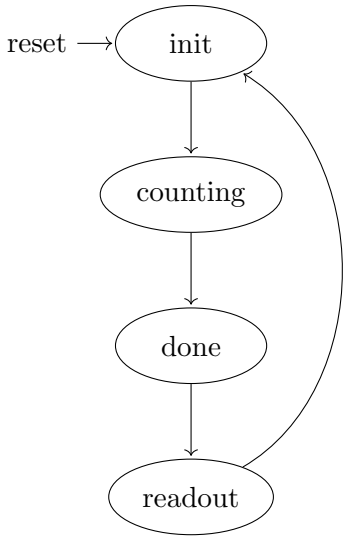


Fig. 5.9: The formal FSM used in the formal verification of the Histogram Core module.

The state machine moves from `init` to `counting` when the core has finished initializing itself. The state machine then moves to `done` when the core receives the end-of-frame flag.

After a short delay, the core enters `readout`, where it will stay until the memory has been fully read out.

The formal verification assumes the module only receives data and the end-of-frame flag when this state machine is in the `counting` state. Therefore, the module must be provided sufficient time to clear its memory, and the entire histogram data must be read out before collecting more histogram data.

The `anyconst` construct is again used for this formal verification. In the verification, p is an arbitrary pixel value where $0 \leq p < 2^{14}$, and n_p is the number of times a pixel of value p has entered the module after exiting the initialization state.

Pixel value counts exit the Histogram module in increasing order of pixel value. Therefore, to assert the Histogram Core works as expected, it is asserted if p pixel counts have been read out during the `readout` state, the next pixel count will be n_p. This assertion is complicated by the fact the module splits each 32-bit count into two 16-bit words to fit in the 16-bit Wishbone bus.

Multiple additional assertions were required for an inductive proof to pass. One assertion states that during the `counting` and `done` states, the value in the Histogram memory at address p is equal to the value of n_p as it was two clock cycles ago.

5.6 Simulation

As described above, formal methods quickly become impractical as the scope of the subsystem being verified grows. When formal methods are impractical, modules are verified using simulation instead.

Simulation is practical even for millions of clock cycles. Still, there is a practical limit to what may be simulated. For example, simulating the entire camera design as it acquires one image on the CI takes over an hour even with small window sizes. A simulation that has the camera acquire multiple full-sized images would take too long to be practical.

Simulation techniques are appropriate, therefore, for verifying subsystems that are too complex for formal verification but are still small enough in scope to make simulation

feasible. The majority of the MUSE design lies in this middle ground, so most of the camera is verified via simulation.

Verifying modules via simulation involves writing a testbench to exercise the module's inputs. In most cases, testbenches are self-checking in that they include assertions to programmatically check the output of the module is as expected. Self-checking testbenches avoid human error caused by manually reading simulation waveforms. Such testbenches also give immediate feedback on whether a module still works, allowing for quick iteration.

Many simulations read test input from a file. One custom file format used in many simulations throughout the design contains a list of Wishbone operations. An example is shown in Figure 5.10. The first column indicates whether the command is a read (`r`) or a write (`w`) command. The second column is the address to read from or write to. The third column is the value. In the case of a write command, the testbench writes the value to the address; in the case of a read command, the testbench reads from the address and asserts the read value matches the value in the file.

```
w 0100 0123
w 0101 4567
w 0102 89AB
w 0103 CDEF
r 0100 0123
r 0101 4567
r 0102 89AB
r 0103 CDEF
```

Fig. 5.10: Example Wishbone operations list used during simulation.

A similar file format exists for SpaceWire packets for top-level simulations.

5.7 Integration Tests

As much of the design is verified through formal methods and simulation as possible. Some functionality, however, is only practical to test in hardware. Simulation time is the largest factor; some aspects of the camera's functionality would take weeks to test in simulation.

Integration tests are used where even simulation is impractical. These tests are ran on a computer connected to the camera via SpaceWire. Each test sends a series of packets to the camera via the C&DH SpaceWire Interface and receives packets through both SpaceWire interfaces. The test then parses all packets and checks the value of each packet. The integration tests are written in the Rust programming language.

Integration tests as used here refer specifically to those automated tests performed on hardware that require no intervention. The overall process of testing the camera meets requirements involves many other tests outside the scope of this report.

Many integration tests rely on the ramp pattern feature of the Science ADC Interface described in Section 5.2. This feature creates a perfectly reproducible image, named the ramp image, on every run of the test. The tests have access to the golden ramp image, a known image produced ahead of time.

Below is the list of tests. Each test has a SG and a CI variant.

5.7.1 PLUT Ramp Test

This test writes random values to the PLUT. After enabling the test ramp, the test then takes one image with the PLUT bypassed and one image with the PLUT enabled. The test then checks the first image matches the gold ramp image. Finally, the test checks each pixel in the second image is equivalent to the corresponding pixel in the first image after applying the PLUT transformation.

5.7.2 Compression Test

This test verifies the camera compresses images correctly. The test relies on a bit-accurate model provided by the compression IP core vendor.

This test starts by enabling the test ramp and configuring the compression IP core for lossless compression. The test then takes one image with compression disabled and one image with compression enabled.

The test first verifies the uncompressed image matches the golden ramp image produced ahead of time. The test then applies the bit-accurate model to the ramp image to obtain

a golden compressed ramp image. The test finally checks the second image data matches that golden image exactly.

5.7.3 Lossy Compression Test

This test is identical to the above compression test except that the compression IP core is configured for lossy compression.

5.7.4 Framerate Test

This test ensures the camera is capable of acquiring and transmitting image data fast enough.

The test sets the frame repetitions register, a register in the CCD Interface module that causes the camera to take many pictures at a fixed rate. The test then sets the command period to the lowest command period the camera should be capable of handling.

After collecting the sequence of images, the test verifies the correct number of images was received and that each image matches the golden ramp image exactly.

5.7.5 Memory Test

This test verifies the entire DDR2 memory is accessible. The test first generates a random test pattern. Then, the test writes the pattern to the camera's memory, reads the entire memory back, and compares the read data against the test pattern. This test uses the Image Buffer's debug functionality to access the memory.

It is important the entire test pattern is written first before any of the pattern is read. If the pattern was written to memory and then read back in smaller pieces, memory aliasing errors could be missed.

CHAPTER 6

RESULTS

The firmware design of the MUSE cameras was successful. Both cameras work as expected. To demonstrate the cameras successfully acquire images from the CCDs, images were taken in the lab at room temperature. These images, shown in Figures 6.1 and 6.2, were displayed on a lab computer using image display software written by another engineer on the team.

Fig. 6.1: Image acquired from the Spectrograph during lab testing.

6.1 Framerate

One issue encountered during the development of the MUSE camera was that the camera was unable to take images at a fast enough rate.

The initial design only stored one image in the external DDR2 memory at a time. The

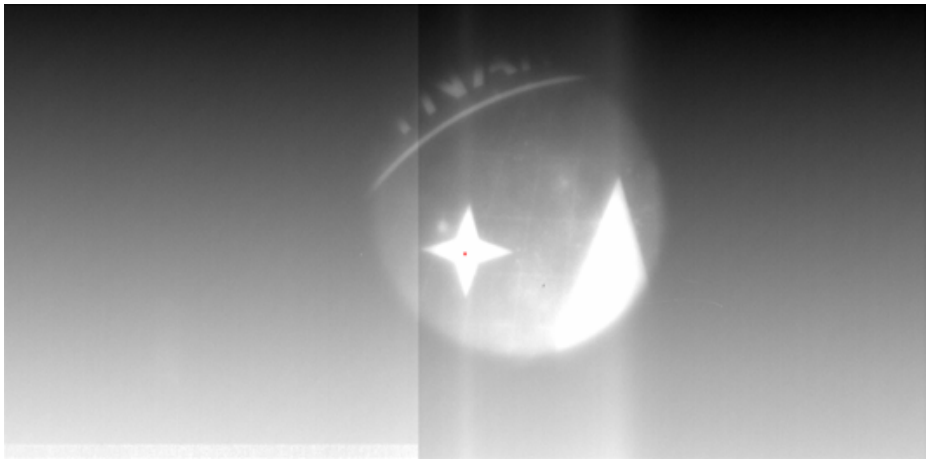


Fig. 6.2: Image acquired from the Context Imager during lab testing.

camera would handle one image in its entirety, including reading from the sensor, compressing, and outputting via SpaceWire, before handling the next image. While the approach was straightforward, it became clear the resulting lower framerate was unacceptable for the mission and the design was required to change.

As discussed in Section 2.4, the design underwent multiple revisions, with later revisions including a circular image buffer.

6.1.1 Analysis

After the revision, the new design underwent analysis and verification to ensure the camera was operating as expected.

There was no set framerate the camera is required to operate at. This is because the camera may operate at a higher framerate when a smaller window size is used. A smaller window size reduces the amount of data required to be read from the sensor and output via SpaceWire.

Initial analysis efforts attempted to characterize the performance of each module in the camera based on the command period and window size. It became clear a better approach was to analyze the camera based off the concepts of throughput and latency. Such analysis is valid so long as the buffers in the system are large enough to handle transient delays in modules. While both parameters are important, throughput is the critical parameter that

determines the maximum framerate for a given window size.

Per requirements, the camera design reads from each tap of the SG sensors at a rate of 1 MPix/s, or one million pixels per second. With three sensors, each with two taps, the total pixel rate is 6 MPix/s. Rounding up to 16 bits per pixel, the total data rate coming from the sensors is **96 Mbit/s**. The camera reads data from each tap of the CI sensors at 2 MPix/s, resulting in a total data rate from the one sensor of **64 Mbit/s**. These are worst-case estimates as they assume no dead time.

The Rasterizer, Programmable Lookup Table, and Pixel Stream Header Prepender modules are all capable of handling pixel data at a rate of one pixel per clock cycle. In the slowest case on the CI, all pixel data runs through one Rasterizer, resulting in a throughput of **640 Mbit/s**.

The DDR2 memory runs at 160 MHz. Two transfers occur each clock cycle. Each transfer moves 48 bits of data, of which the bottom 32 bits are non-ECC. The throughput of the memory is therefore around **10,240 Mbit/s**.

Pixel values then flow through the inter-FPGA pixel stream interface. This module pair operates at 10 MHz, resulting in a throughput of **160 Mbit/s**.

The compression IP core handles pixel values at a rate of one pixel per 1.4 clock cycles, resulting in a throughput of **400 Mbit/s**.

Finally, the pixel data is output through an 80 MHz SpaceWire interface. The SpaceWire standard has two bits of overhead per byte of data transmitted, resulting in an effective throughput of **64 Mbit/s**. Pixel values are wrapped in packets, but the overhead from the packet headers are negligible.

The analysis shows that on the CI, the camera is already capable of outputting even uncompressed image data faster than it can read data from the image sensor. On the SG, however, if the camera is reading from all three sensors as fast as possible, the camera can only output the data if the compression ratio is greater than 1.5.

To verify the analysis and ensure the camera is capable of displaying images at the required framerate, a script was written to acquire many images in sequence. This is the

framerate test described in Section 5.7.4.

CHAPTER 7

DISCUSSION

The discussion section of this report discusses which aspects of the design and verification process worked well and which did not work.

7.1 The Design Process

An initial architecture for the MUSE camera was developed at the start of the program. However, as development proceeded, this architecture changed many times as the full shape of the problem became clear. In general, it is difficult to correctly predict the optimal architecture of a complex system in precise detail ahead of time. When one architects a system, one should expect the architecture to change. The initial architecture should be viewed as a starting point from which the problem space can be explored through implementation. During implementation, issues will arise that motivate changes to the architecture.

There is balance to strike on how flexible the architecture should be to such changes during development. When the architecture is fixed, modules must work around flaws in the architecture, resulting in higher design complexity, lower productivity, and sometimes worse performance. On the other hand, if the architecture is seen as so flexible it undergoes large changes in response to every problem that arises during implementation, much design effort will be spent re-designing existing modules.

The MUSE design struck a reasonable balance on this flexibility. There were large changes made to the architecture, but these efforts resulted in significant improvements and were necessary. For example, earlier on in the design process, the architecture changed from ad-hoc behavior as described in Section 2.4.2 to more structured control flow as described in Section 2.4.3. If this change did not occur, a substantial amount of effort would have been required to work around the flaws of the initial architecture.

Designing from a place of more experience would have allowed for the initial architecture

to be closer to an optimal design. Experience would have also made changes towards the optimal design faster. The rest of this section discusses general knowledge revealed by the MUSE design that would be beneficial for future designs.

7.2 Control Flow

As discussed in Section 2.4, the design of the MUSE camera underwent significant changes to its control flow. These changes simplified the design of the camera, opening up opportunities to make the design more robust.

Later revisions of the control flow aimed to simplify handling backpressure. These later revisions decide at the beginning of the image whether to reject an entire image depending on whether the Image Buffer has enough room for one full image. This is different than a previous design iteration where an image could be dropped at any arbitrary point in time when the buffer became full, requiring every module to be designed to gracefully support dropping an image.

The more general pattern of this aspect of the design is this: where possible, engineers should place their efforts towards eliminating failure points by design instead of gracefully detecting and recovering from failures. A generally useful tactic to achieve this is to only begin an operation when the conditions are such that the operation is guaranteed to complete successfully.

Another useful tactic for simplifying a design's control flow is to centralize control in the design as much as possible. Initial iterations of the MUSE design spread control of the camera throughout multiple modules. Each module would determine when to act by reacting to signals or new data from surrounding modules. This design came from a fine-grained focus on the design of each module with insufficient consideration for the control flow of the entire camera as a whole.

As a consequence, modules had internal state that was fairly loosely connected to the state of surrounding modules. The number of states the camera could be in at a given time was very large due to the large combination of possible states each individual module in the design could be in. Controlling the camera reliably was difficult.

Later iterations of the MUSE design moved towards more centralized control. If the program schedule permitted, the design could benefit from moving even further along this direction.

One example of a major design change that could have benefitted the MUSE design is to use a soft-core processor. Processors are particularly suitable for control flow as implementing large amounts of conditional branches is trivial to perform with software. A processor also increases iteration speed; if a change only affects software, new designs can be tested immediately without having to synthesize a new circuit design. Slow iteration speed has been a major obstacle in the MUSE program.

If the design contained a processor, one aspect of the design in particular would see much benefit: handling commands. The design currently responds to commands using a very large state machine. This approach is costly to implement and error prone; multiple bugs have been discovered in this part of the design.

7.3 Memory Bus

The MUSE camera design implemented a memory bus. This bus is used almost entirely for access to configuration registers within each module. Having a consistent method for accessing the camera configuration was valuable as it simplified the process of adding extra configurable parameters to the camera. There were multiple requirements for the camera to be very flexible, so this aspect of the design was particularly helpful.

The MUSE camera design could be improved by making use of the memory bus even more. Various modules throughout the design contain memories, including the PLUT, the Histogram, and the Image Buffer. These modules all provide custom ad-hoc interfaces for reading and writing to these memories.

These memories could instead have been directly memory-mapped to the bus. Had this been done, configuring the PLUT would have required simply writing to a range of addresses in memory, reading the Histogram would have required simply reading to a range of addresses in memory, and the debug interface with the Image Buffer would have been simplified.

Access to the large DDR2 memory would also require a larger address width in the design. The current address width of 16 bits only allows access to 65,536 words, or 131,072 bytes; this is less than 0.03% of the DDR memory. If the width was increased to 32 bits, every address in the DDR memory could be accessed with much room to spare in the address space for configuration registers and other memories.

Special consideration should be given to which bus standard to use in future designs. The choice to use the Wishbone standard greatly simplified the design of each module. This simplicity also aided in formal verification. The standard has downsides, however. Wishbone is less commonly supported. For example, the Image Buffer module instantiates an IP core to implement Reed-Solomon error correction. This module only provides an AXI interface. Directly connecting this IP core to the memory bus would require the use of a Wishbone-to-AXI bridge.

7.4 Verification

As described in Section 5.4, verification effort was separated according to the scope of the subsystem into one of three categories: formal methods, simulation, or integration testing. This approach proved to be an effective way to allocate verification effort.

Formal verification, in particular, helped greatly during development. Formal verification gave confidence in particularly tricky designs that would have resulted in subtle bugs. The formally verified modules proved to be very robust.

One bug was discovered in the Wishbone CDC module. Register accesses in the Compression Core would not work immediately after powering on until after a soft reset. Similar bugs were discovered elsewhere in the design, but this bug is notable because it was found in a formally verified module. This discovery exposes the limitations of formal verification. Formal methods only guarantee correct behavior if all the listed assumptions are met and if the proven properties correctly describe expected behavior. In this case, the formal verification implicitly made incorrect assumptions about the behavior of the RTG4 FPGA's outputs as it powered on.

Simulation and integration tests were made to automatically check for correct behavior. For example, many integration tests check every single pixel of received images is as expected. This automation was very useful as it allowed for quickly checking changes. The automation boosted confidence in the design.

There are aspects of the verification approach that could be improved. While all modules were verified as they were completed, module designs have changed over time to adapt to new requirements and a changing architecture. On occasion, changes were made that introduced issues, but the affected modules were not re-verified. The MUSE program could have benefited greatly from continuous integration techniques. These techniques are common in software projects and involve having a server run a suite of tests on the design with each change. These techniques would have ensured every module is continually tested during development and would have exposed these bugs as soon as they were introduced.

7.5 Conclusion

This document described the design of FPGA modules created to meet the needs of the MUSE mission. Years of careful development and thorough verification have resulted in a robust, functional camera design that will help advance our understanding of space weather.

The modules described in Chapter 3 will directly handle pixel data from the high-resolution images of the Sun's corona. These modules order pixel values read from each CCD into raster-scan order, transform pixel values according to a lookup table, buffer images in memory, and compress each image. The overall architecture of the camera helps ensure changes to the camera's configuration apply to each section of the image pipeline at the right time.

The modules described in Chapter 4 will support other functions of the camera during the mission. These modules produce cumulative histograms for each image, detect and mitigate single-event effects, monitor voltage, current, and temperature measurements, control a heater, and provide configuration register access across a clock domain boundary.

These modules were verified using a combination of formal methods, simulation, and

integration testing. This approach proved to be an effective way to allocate verification effort and still achieve a reasonable level of confidence in the design. Three modules, the Rasterizer, Histogram Core, and Wishbone CDC Bridge, were formally verified. Formal verification provided great value, as it gave very high confidence in the trickiest part of the camera design.

There are known limitations of the design. As described in Section 7.4, a bug was bug found in the Wishbone CDC module that formal methods did not catch due to incorrect assumptions about the behavior of the RTG4 FPGA's outputs as it powers on. This bug was fixed, but the fix has not been formally verified. Additionally, interfaces to the various modules - the the various memories in particular - is inconsistent, and the design could benefit from a sweeping pass to make these interfaces more similar. Future work may map all of these memories into a common address space accessible via the memory bus as described in Section 7.3.

Future work may also consider implementing a soft-core processor to handle commands and configure the camera. A central controller would simplify camera control, and a processor would simplify decision-heavy operations such as handling packets as Section 7.2.

Future works would benefit from Continuous Integration techniques to automatically verify the entire design every time changes are made. These techniques would help maintain a higher level of confidence in the design and catch issues as soon as they are introduced.

While there is room for improvement, the project was successful. The camera successfully captures images from the CCDs and correctly handles updates to its register configuration. The modules described in this report will soon be operating in space as part of the MUSE mission and will handle high-resolution images of the Sun's corona.

REFERENCES

[1] W. Dally and J. Poulton, *Digital Systems Engineering*. Cambridge, U.K.: Cambridge University Press, 1998.

[2] Teledyne e2v, "Back illuminated AIMO frame-transfer high performance CCD sensor," CCD47-20 datasheet, Oct. 2017, version 4.

[3] ——, "Back illuminated 4096 x 4096 pixel scientific CCD sensor," CCD203-82 datasheet, Jan. 2007, issue 2.

[4] D. E. Gisselquist. (2022, Feb.) AXI stream is broken. Gisselquist Technology. [Online]. Available: https://zipcpu.com/blog/2022/02/23/axis-abort.html

[5] *RTG4 Fabric User Guide*, Microchip, Chandler, AZ, Nov. 2023, rev. B.

[6] C. E. Cummings, "Clock domain crossing (CDC) design & verification techniques using SystemVerilog," in *SNUG*, Boston, MA, 2008.

[7] *RTG4 FPGA Datasheet*, Microchip, Chandler, AZ, Feb. 2024, rev. D.

[8] S. Beer, R. Ginosar, M. Priel, R. Dobkin, and A. Kolodny, "The devolution of synchronizers," in *2010 IEEE Symposium on Asynchronous Circuits and Systems*, 2010.

[9] S. Beer, R. Ginosar, R. Dobkin, and Y. Weizman, "MTBF estimation in coherent clock domains," in *2013 IEEE 19th International Symposium on Asynchronous Circuits and Systems*, 2013, pp. 166–173.

[10] D. E. Gisselquist, "An introduction to formal methods," Class notes for An Introduction to Formal Methods, Gisselquist Technology. [Online]. Available: https://zipcpu.com/tutorial/formal.html

[11] ——. (2018, May) Formally verifying asynchronous components. Gisselquist Technology. [Online]. Available: https://zipcpu.com/formal/2018/05/31/clkswitch.html

APPENDICES

APPENDIX A

Memory Map

The memory map of the Wishbone bus is shown in Figure A.1. Word addressing is used per the Wishbone standard. Note `Histogram 2`, `Histogram 3`, `PLUT 2`, and `PLUT 3` are unused on the CI camera.

The memory map of the image buffer is significantly different between the SG and the CI. The SG image buffer is shown in Figure A.2, and the CI image buffer is shown in Figure A.3. Byte addressing is used for the image buffer memory.

Having separate memory maps and using different types of addressing in the same design is unusual. Chapter 7 discusses this aspect of the design.

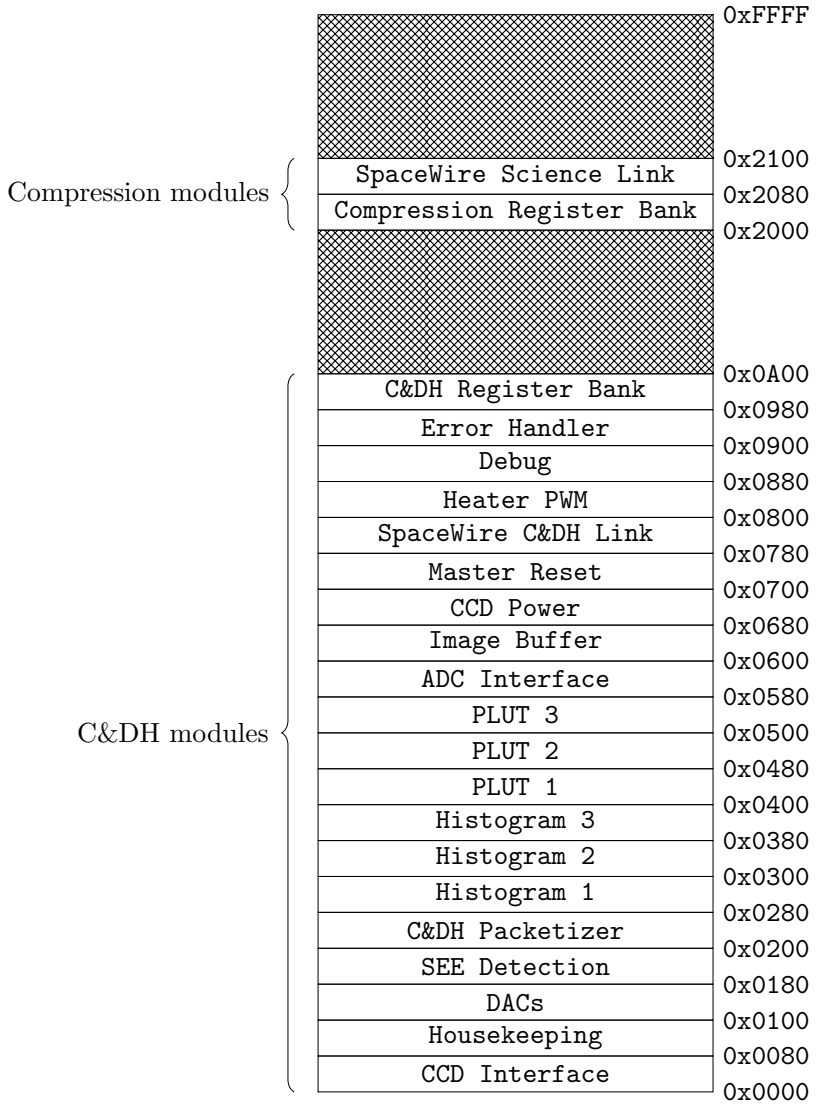


Fig. A.1: The memory map of the Wishbone bus (word-addressed).

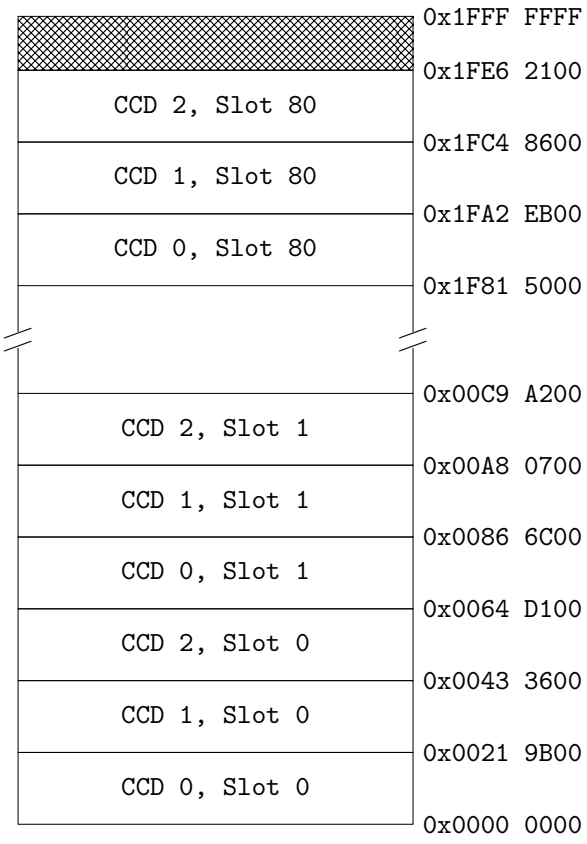


Fig. A.2: The memory map of the image buffer on the SG (byte-addressed).

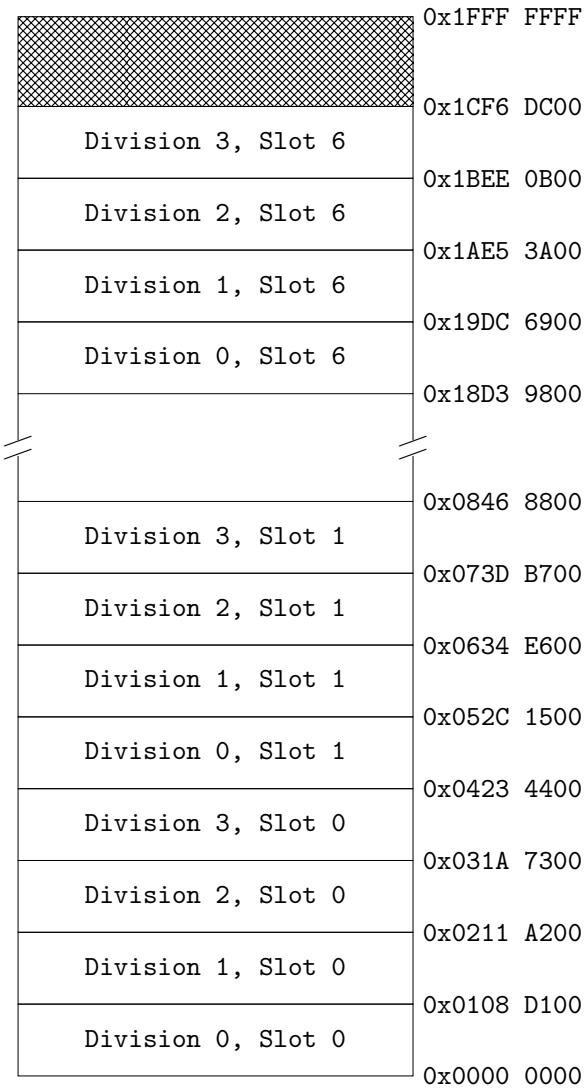


Fig. A.3: The memory map of the image buffer on the CI (byte-addressed).

APPENDIX B

Resource Utilization

Table B.1 contains the list of resources used throughout the FPGA as reported by the toolchain.

The RTG4 FPGA contains 151,824 logic elements. Each logic element contains one four-input lookup table and one flip-flop reported as `4LUT` and `DFF`, respectively. The lookup table and flip-flop can be used together or independently. The RTG4 also contains two forms of writeable memories: µSRAM and LSRAM, reported as `RAM64x18` and `RAM1K18` respectively [5].

Table B.1: Resource Utilization

FPGA	`4LUT`	`DFF`	`RAM64x18`	`RAM1K18`	`MACC`
SG C&DH	51,602 (34%)	33,088 (22%)	41 (20%)	141 (67%)	0 (0%)
CI C&DH	43,866 (29%)	28,845 (19%)	44 (21%)	115 (55%)	0 (0%)
SG Compression	46,859 (31%)	37,360 (25%)	45 (21%)	154 (74%)	36 (8%)
CI Compression	46,860 (31%)	37,360 (25%)	45 (21%)	154 (74%)	36 (8%)
SG Emulator	50,025 (33%)	33,790 (22%)	67 (32%)	125 (60%)	0 (0%)
CI Emulator	43,667 (29%)	30,517 (20%)	73 (35%)	99 (47%)	0 (0%)