

Virtual Pan-Tilt-Zoom for a Wide-Area-Video Surveillance System

by
Richard Sinn

S.B., EECS M.I.T., 2007

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the


MASSACHUSETTS INSTITUTE OF TECHNOLOGY

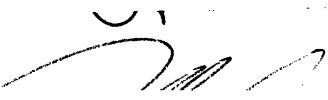
September 2008

© Richard Sinn, MMVIII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author
Department of Electrical Engineering and Computer Science
August 22, 2008

Certified by

Dr. Pablo I. Hopman
MIT Lincoln Laboratory
Thesis Supervisor

Certified by

Dr. Christopher J. Terman
Senior Lecturer
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

This report is based on studies performed at Lincoln Laboratory, a center of research operated by the Massachusetts Institute of Technology. This work was sponsored by the Secretary of the Air Force/Rapid Capabilities Office under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and not necessarily endorsed by the United States Government.

ARCHIV

ARCHIVES

Virtual Pan-Tilt-Zoom for a Wide-Area-Video Surveillance System

by

Richard Sinn

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Advancements in the CMOS Image Sensor have enabled very high-performance, high-resolution imaging systems to be built at relatively low cost. The availability of high-pixel count video imaging systems that can cover a wide field-of-view enables a surveillance technique called Virtual Pan-Tilt-Zoom. Virtual Pan-Tilt-Zoom provides the same functional properties as a mechanical pan-tilt-zoom setup, but it does not suffer from the physical limitations presented by a mechanical setup. A video system using Virtual Pan-Tilt-Zoom would have immediate continuous access to a high-pixel-count image representing a wide coverage area, and it would enable a user to "virtually" pan, tilt, and zoom around the coverage area by reading out only the relevant image data associated with a Region of Interest that is dynamically defined by the user. This paper will examine the various camera electronics readout architectures that are possible to support the Virtual Pan-Tilt-Zoom function. Then, this project will examine and implement a specific implementation of the readout architecture for a high-resolution video camera system developed at MIT Lincoln Laboratory. The Multi-Aperture Sparse Imager Video System (MASIV) developed at MIT Lincoln Laboratory incorporates CMOS imagers to create an 880 Megapixel image, and was used as the platform to implement the camera electronics for Virtual Pan-Tilt-Zoom functionality.

Thesis Supervisor: Dr. Pablo I. Hopman
Title: MIT Lincoln Laboratory

Thesis Supervisor: Dr. Christopher J. Terman
Title: Senior Lecturer

Acknowledgments

This project was possible because of the support, help, and guidance I have received from many people. I acknowledge a few below, but it is by no means a complete list of all the people who have supported and inspired me along the way.

First of all, thank you to my thesis supervisors, Pablo Hopman and Chris Terman, for their honest advice and guidance throughout the project. I would also like to thank the members of the Advanced Space Systems and Concepts group at MIT Lincoln Laboratory for giving me the opportunity to pursue a project with the group. A special thank you to Tom Karolyshyn for his patience, guidance and support in helping me with the project. Also, thank you to Larry Candell, James Glettler, Daniel Chuang, Fred Knight, Pete LaFauci, Mike Mattei and Bobby Ren for all their help and support.

I would like to thank my friends for making my time at MIT a truly educational experience. Thank you to Bobby Ren for hosting me in his house during the last couple weeks so that I could finish my thesis, and of course for being my role model and friend. Thank you to Dan Chuang, Nathan Hanagami, John Ho and Conor Madigan for being my mentors, role models and friends. And last but not least, thank you to Christian Deonier, Rich Lean, Tri Ngo and JinHock Ong for being the best of friends.

Finally, I would like to thank my family for giving me the opportunity to pursue my goals and dreams. Thank you Mom and Dad for supporting me and always believing in me. Thank you for teaching me to always try hard and never give up. Thank you Victor, my wonderful brother, for always inspiring me and bringing out the best in me.

Contents

1	Introduction	13
1.1	Overview and Purpose	13
1.2	Virtual Pan-Tilt-Zoom Readout Architecture on the MASIV Camera	16
1.3	Thesis Organization	17
2	The MASIV Overview	19
2.1	MASIV System Description	20
2.2	Hardware Components of the MASIV System	21
2.2.1	CMOS Digital Image Sensor	22
2.2.2	Field-Programmable Gate Arrays and Camera Readout Electronics	25
2.2.3	RocketIO Multi-Gigabit Transceivers	26
2.3	Component Connections in the MASIV System	27
2.3.1	Quadrant Layout	28
2.3.2	Aperture Connections	28
2.4	Camera Electronics Operation and Specifications	30
3	Design and Performance Considerations	33
3.1	VPTZ Specifications	34
3.1.1	Data Decimation	34
3.1.2	Data Flow	35
3.1.3	Self-Imposed Specifications and Digital Zoom	36
3.2	Performance Metrics of the Readout Architectures	37

3.3	Different Readout Architectures	41
3.3.1	Display Level VPTZ Decimation	41
3.3.2	FPGA Level VPTZ Decimation	43
3.3.3	Imager Level VPTZ Decimation	47
3.4	Summary of the Differences between the Readout Architectures . . .	53
4	Specific Implementation of a Camera Readout Architecture for VPTZ	55
4.1	Existing MASIV Firmware Overview	55
4.1.1	Imager Control Module	56
4.1.2	Logic Modules for Interfacing with RocketIO MGT	57
4.1.3	PowerPC and Register File	57
4.2	VPTZ Parameters	58
4.2.1	Coordinate Spaces and Region of Interest Start Point	58
4.2.2	VPTZ Subsampling and Number of Valid Pixel Columns and Rows	60
4.2.3	Row Offsetting	62
4.3	Logic Implementation of the Readout Architecture	64
4.3.1	Main Decimation Logic	66
4.3.2	Timeslot Sorting of the Pixel Data before Buffering	69
4.3.3	Row by Row Buffering using BRAM	72
4.3.4	Logic to Read Pixel Data from the Buffers	77
4.3.5	Quadrant Selector Module	80
4.4	Testing and Debugging	82
4.4.1	Simulating the Main Decimation Logic	83
4.4.2	Simulating the Top Module	84
4.4.3	Simulating the Reading of the Buffer	87
5	Conclusion	91
5.1	Future Work	91
5.2	Summary	92

A	VHDL Source Code	97
A.1	foveation_top.vhd	97
A.2	rectangles.vhd	111
A.3	fov_buf.vhd	117
A.4	read_fov_buf.vhd	126
A.5	quad_selector.vhd	133
B	VHDL Test Benches	139
B.1	tb_rectangles_new.vhd	139
B.2	tb_foveation_top.vhd	143
B.3	tb_read_fov_buf.vhd	146

List of Figures

1-1	Traditional Mechanical Pan-Tilt-Zoom Cameras	14
1-2	Virtual Pan-Tilt-Zoom with Multiple Users and their Region of Interests	15
2-1	The Current MASIV System and the External Hard Disks.	19
2-2	MASIV Concept, Four Lenses, Four Sparse Arrays, Digital Image Stitching [1].	20
2-3	Sparsely Populated Focal Plane of 44 CMOS Imagers for One Aperture [1].	21
2-4	One Complete Aperture with a Medium-Format Lens [4].	21
2-5	Pixel Array Structure of a CMOS Imager [7]	24
2-6	Pixel readout for a normal Bayer pattern	25
2-7	Hardware Layout for One Aperture [4]	27
2-8	Quadrant Layout of the CMOS Imagers on Focal Plane Mosaic	29
2-9	The RocketIO Connections of the Four Apertures of the Current MA- SIV Implementation	30
3-1	A simplified high level view of the data flow.	36
3-2	Sixteen External Cables Connections vs. One External Cable Con- nection for Image Data Transfer.	40
3-3	Block Diagram of Readout Architecture for Display Level Decimation.	42
3-4	Block Diagram of Readout Architecture for FPGA Level VPTZ Deci- mation	44
3-5	Block Diagram of Readout Architecture for VPTZ Decimation in the Imagers.	48

3-6	An example of a Region of Interest spanning multiple imagers.	50
3-7	Multiple Regions of Interests scenarios with same zoom mode within one CMOS imager.	50
4-1	Pixel Dimensions of the Consolidated Coordinate Space of Entire Cov- erage Area.	59
4-2	Row Offsetting During Vertical Subsampling in One Quadrant	63
4-3	High-Level Block Diagram of Readout Architecture	65
4-4	Block Diagram for the Part of the Architecture that Handles the Burst Pixel Data	66
4-5	Block Diagram of <i>Rectangles</i> Module	67
4-6	State Transition Diagram for Skipping Decimation	69
4-7	Block Diagram Showing How Single Pixel Data Stream is Created from <i>Active Rectangles</i>	70
4-8	Timing Diagram for Sorting Pixel Data into Timeslots of a Single Data Stream	71
4-9	State Transition Diagram for Sorting Pixel Data into Timeslots of a Single Data Stream	72
4-10	<i>Buffer</i> Module Block Diagram to Buffer Pixel Data into BRAMs	73
4-11	Summary of the Bits in a Word	74
4-12	State Transition Diagram for FSM in the <i>Buffer</i> Module	76
4-13	Block Diagram for the <i>Read Buffer</i> module	78
4-14	State Transition Diagram for the FSM in <i>Read Buffer</i> Module	79
4-15	Block Diagram for the <i>Quadrant Selector</i> Module	81
4-16	State Transition Diagram for FSM in <i>Quad Selector</i>	82
4-17	Simulation of the <i>Rectangles</i> Module	84
4-18	Simulation of the Top Module	86
4-19	Simulation of the <i>Read Buffer</i> Module	88
4-20	Simulation of the <i>Read Buffer</i> Module Showing the Last Pixel of the Row	89

List of Tables

2.1	Parameters and Specifications of the CMOS Imager [8]	23
2.2	Timing Specifications of the MASIV System	30
3.1	Calculation of Performance Metrics	38
3.2	Qualitative Comparison of the Different Architectures	53
4.1	Skip Modes and # of Skipped Pixels	61

Chapter 1

Introduction

1.1 Overview and Purpose

Traditional video surveillance systems often incorporate mechanical pan-tilt-zoom (PTZ) schemes to achieve a wide coverage area. The ability to physically pan and tilt the video camera increases the overall coverage area of the system because it increases the number of points in the physical environment that can be observed by the camera. As with any system involving moving parts, these mechanical PTZ schemes often require maintenance, and can become complex and costly for higher performance surveillance applications.

A video surveillance system with a mechanical panning and tilting setup is constrained by the inherent physical limitations of its mechanical structure. The speed at which a user can scan around the coverage area is limited to how fast the camera can physically slew around. Furthermore, multiple regions of the overall coverage area cannot be viewed simultaneously in a mechanical PTZ setup because the camera can only point toward one region at a time. Likewise, mechanical zooming is also bounded by certain physical constraints; the speed at which the user can zoom in and out of the coverage area is limited to how fast the camera's optics can vary its focal lengths, and multiple regions cannot be viewed at different zoom modes simultaneously. Figure 1-1 shows some examples of video surveillance systems with a traditional PTZ setup.



Figure 1-1: Traditional Mechanical Pan-Tilt-Zoom Cameras

A Virtual Pan-Tilt-Zoom (VPTZ) system provides the user with something that is functionally equivalent to a mechanical PTZ system but without the physical constraints. The fundamental basis behind the VPTZ concept is having a wide field-of-view with many pixels. Rather than outputting the entire pixel data for the complete visual field, the video system using VPTZ would read out only the relevant pixel data associated with a Region of Interest that is defined by the user. The user would be able to dynamically change his or her Region of Interest to view different sections of the system's visual field all the while the camera remains stationary. Hence, the user is "virtually" panning and tilting around the area with his or her Region of Interest. The user would also have the ability to virtually zoom in and out of the visual field by downsampling the Region of Interest at various different digital zoom modes.

The VPTZ application does not suffer from many physical limitations of a mechanical PTZ setup. The video system itself must be able to instantly produce a high resolution image for a wide coverage area from a stationary position, and then VPTZ can select and read out portions of the image data as needed for the user. The advantages of VPTZ exist because the camera has immediate continuous access to the pixel data of the entire coverage area. When a user changes his or her Region of Interest to view a different section of the coverage area, the change will happen

instantaneously because the VPTZ application has to just read out a different part of the data set from the video camera. The system is not restricted by the speed of the physical movements a normal camera would have to make in order to view a different section of the coverage area because the entire coverage area is already under the video camera's visual field. Thus, the VPTZ application can now support simultaneous viewings for multiple different sections of the coverage area because VPTZ has access to the data to support the simultaneous viewings. This can allow for multiple users to use the system, with each user controlling his or her own Region of Interest and getting a sense that they are each viewing and controlling a unique camera. Figure 1-2 gives a cursory view of multiple users with their Regions of Interest using VPTZ.

Virtual Pan/Tilt/Zoom Concept

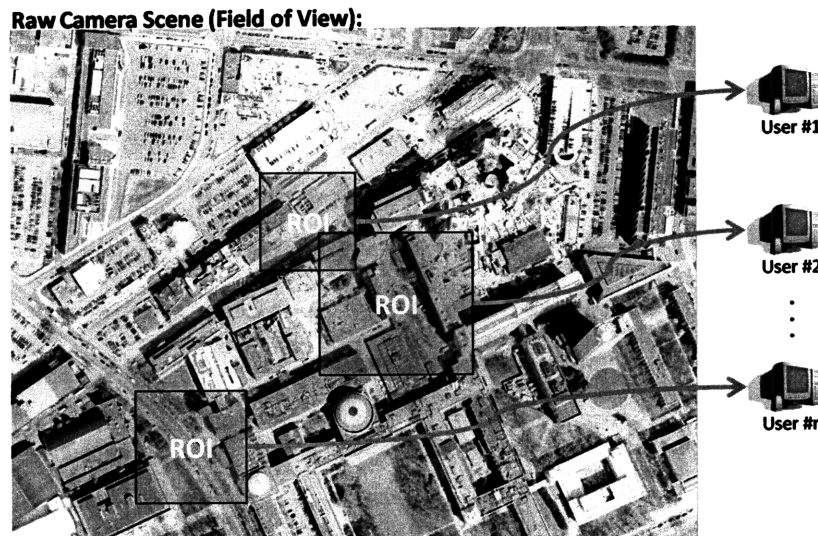


Figure 1-2: Virtual Pan-Tilt-Zoom with Multiple Users and their Region of Interests

Recent developments in digital image sensing technologies spurred by the rise in demand for digital imaging consumer electronics have made high-performance, high-pixel-count imaging systems more readily available. In particular, as a result of the demands in the commercial markets, the advancements of the Complementary Metal Oxide Semiconductor (CMOS) digital image sensor have made high-pixel-count video

imaging possible [2] [3]. Moderately sized video surveillance systems are now able to obtain image data at increasingly higher resolutions for wider coverage areas.

The Multi-Aperture Sparse Imager Video System (MASIV) camera is an 880 Megapixel video imaging system being developed at MIT Lincoln Laboratory [1]. The MASIV camera is the realization of the fact that high-pixel-count imaging systems are now relatively inexpensive to obtain; it uses a unique approach to achieve its 880 Megapixel resolution by using multiple off-the-shelf CMOS image sensors and other common electronic parts. Its original purpose is to be a monolithic staring sensor, but its high-performance specifications make the MASIV camera an ideal platform for the development and testing of the VPTZ concept.

1.2 Virtual Pan-Tilt-Zoom Readout Architecture on the MASIV Camera

When given a very large image data set that represents a wide coverage area, VPTZ is used to decimate the majority of the pixels generated by the cameras by windowing and downsampling the image data to the user's Region of Interest. The readout architecture for the camera electronics for VPTZ functionality essentially determines where in the data flow to decimate away the image data that is not a part of the end user's Region of Interest. In a digital video imaging system, the image data usually flows from the image sensors to the camera's electronics to the end user.

In the MASIV system, the data can be decimated by the CMOS imagers, by the camera electronics, or by external devices that the camera system is interfaced to, such as a computer. The different VPTZ readout architectures determine which component or components would do the main image data decimation for VPTZ functionality. Certain system specifications and performance metrics such as the frame rate of the system can change depending on the different VPTZ readout architectures. Furthermore, the VPTZ readout architecture must be able to handle challenges such as overlapping Regions of Interests if there are multiple users using the system as

shown in Figure 1-2.

1.3 Thesis Organization

In this thesis, we will examine the various camera electronics architectures for implementing the VPTZ concept in the MASIV system. Chapter 2 provides a background for the current implementation of the MASIV concept to create a high resolution digital imaging video system. In Chapter 3, we will study the various readout architectures that are possible for implementing VPTZ and the tradeoffs associated with the different architectures. In Chapter 4, we will describe a specific implementation within the MASIV camera that demonstrates the VPTZ camera electronics architecture. In Chapter 5, we will provide a discussion for future work and a summary of the project.

Chapter 2

The MASIV Overview

The MASIV system is an airborne video sensor that was designed to observe a wide coverage area at a very high-resolution for persistent surveillance applications [1]. It was primarily constructed to save all the image data coming off of the cameras so that the data could later be used for surveillance and analysis. Figure 2-1 shows a photo of the current MASIV camera system alongside a chassis that contains the hard disk drives designed to store several Terabytes of data.

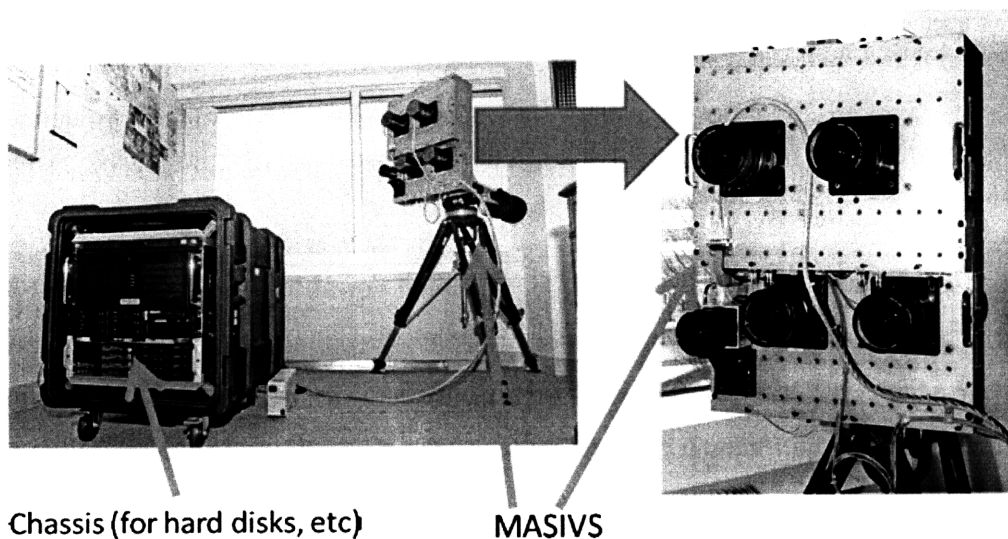


Figure 2-1: The Current MASIV System and the External Hard Disks.

2.1 MASIV System Description

The MASIV system is a very high-pixel-count video imaging system. It creates a contiguous image by stitching together four sparsely populated mosaics of smaller sensors [1]. Figure 2-2 demonstrates this concept behind the MASIV system. The small sensors that are populated on the mosaics of the MASIV are CMOS digital image sensors. The CMOS imagers' performance, small size and availability are what make the MASIV system's concept feasible.

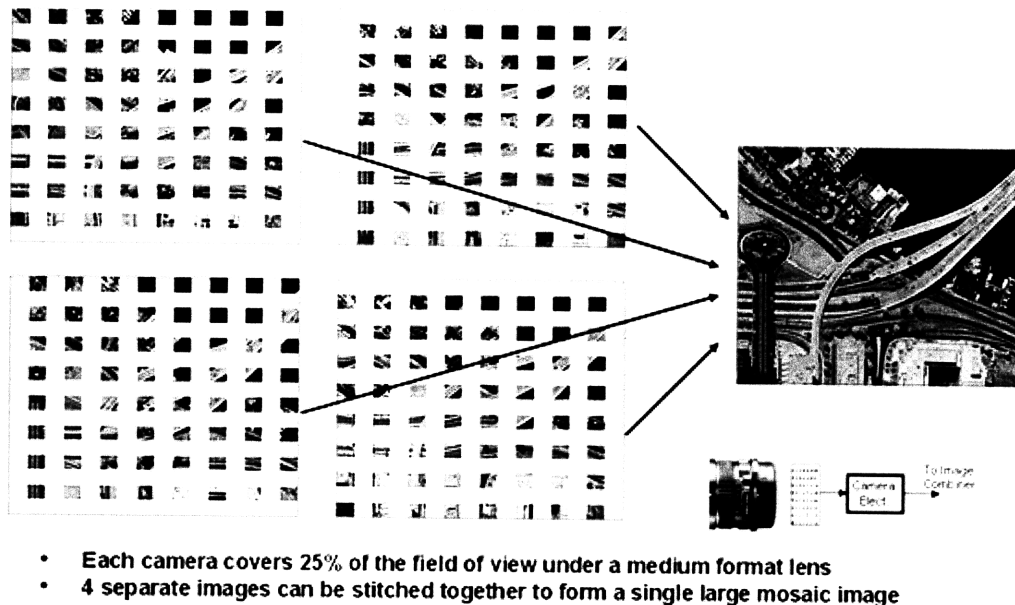


Figure 2-2: MASIV Concept, Four Lenses, Four Sparse Arrays, Digital Image Stitching [1].

Each of the mosaics corresponds to a focal plane of an aperture of the MASIV system. The high-pixel-count is achieved by sparsely populating each of the focal plane mosaics with 44 Five-Megapixel CMOS imagers in an array fashion. Figure 2-3 shows the sparsely populated focal plane mosaic for one aperture of the MASIV system. The gaps that occur between the CMOS imagers on the focal plane mosaic are filled in by the imagers from the other mosaics when the final contiguous image is stitched together. Notice, that the corners of the focal plane are not populated because the image quality of the optics in the corners are not usable. A single aperture

of the system with the lens attached to it is shown in Figure 2-4. There are a total of four apertures with these focal plane mosaics in the MASIV system. The MASIV system as shown in Figure 2-1 combines the four apertures together, thus employing a total of 176 Five-Megapixel CMOS imagers to create an 880 Megapixel image.

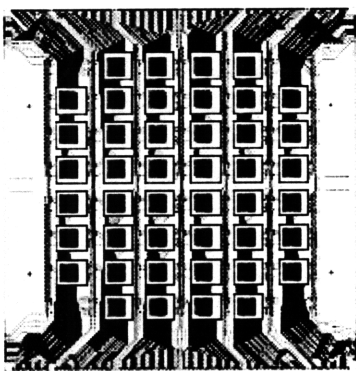


Figure 2-3: Sparsely Populated Focal Plane of 44 CMOS Imagers for One Aperture [1].

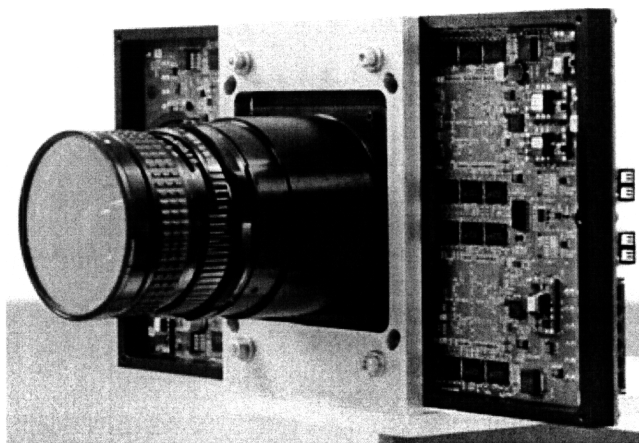


Figure 2-4: One Complete Aperture with a Medium-Format Lens [4].

2.2 Hardware Components of the MASIV System

The MASIV system is entirely made up of commonly available high-performance electronic parts. The architecture of the system is designed to ensure that the data readout from the 176 CMOS imagers can be supported. Because of the massive

amounts of pixel data that are generated, high-speed components and connections must be used in order to sustain the data rates and payload for the 880 Megapixel system.

The current implementation of the MASIV system is a fairly complex setup as it needs to buffer, process and store 880 Megapixels worth of raw image data at 2 frames per second. Components such as Double-Data-Rate Random Access Memory (DDR RAM) are used as data rate buffers for the image data coming off the CMOS imagers; JPEG processing cards are used to compress the image frames coming off the cameras. Though these components are critical to the current implementation of the MASIV system, they are not actually important to the discussion of the readout architecture for VPTZ functionality as described in this thesis. The VPTZ functionality provides an alternative way to read out the image data being generated by the MASIV cameras, and it is not reliant on some of the components that are used with the original implementation of the MASIV system. Therefore, only the major hardware components of the MASIV system that will also be relevant to the discussion of VPTZ will be discussed in detail.

2.2.1 CMOS Digital Image Sensor

The CMOS imager is what senses and collects the image information. The CMOS imagers' relatively low cost and ease of availability make it a popular option for both scientific and commercial high-performance imaging devices [6]. They are what make the near-Gigapixel video rate imagery of the MASIV system possible. The CMOS imager's resolution capabilities, small pixels, and compact form factor enable the MASIV system to achieve very high-resolution images over a wide coverage area. The CMOS imagers that are currently employed in the MASIV system are the Micron MT9P001 5-Megapixel CMOS digital image sensors. Table 2.1 gives a quick summary of the important specifications for this particular CMOS imager.

One of the key requirements needed for the MASIV system is that the image sensor must be small and compact. The pixel dimensions for a typical CCD sensor is on the order of about $10\mu\text{m} \times 10\mu\text{m}$, but these particular Micron CMOS imagers have

Table 2.1: Parameters and Specifications of the CMOS Imager [8]

Parameter/Specification	Value
Pixel Size	2.2 μm x 2.2 μm
Die Dimension	8.5mm x 7.95mm
Active imager size	5.70mm x 4.28mm
Total # of Active pixels	2,592H x 1,944V (5Mpixels)
Color filter array	RGB Bayer Pattern
Max Frame rate at full resolution	15 fps
ADC resolution	12-bits
Power	<317mW (15fps)

a pixel dimension of $2.2\mu\text{m} \times 2.2\mu\text{m}$. The small pixel size of these CMOS imagers make possible for a compact focal plane, relatively low power consumption, and good noise performance. The compact die form factor of these Micron CMOS imagers enables the imagers to be closely packed to each other on the focal plane mosaics so that there is a gap size of only one imager between the imagers. The gaps are filled in by the imagers from the other apertures when the multiple apertures of the system are combined together. In a four aperture setup like that of the MASIV system, the external die dimensions must be smaller than two times the area of the active imaging region of the CMOS imager chip. This is necessary in order to properly integrate the imagery from the four apertures to create a large contiguous image as demonstrated in Figure 2-2.

CMOS imagers inherently combine both the analog and digital components needed for common camera electronics into a single integrated circuit. As a result, the analog-to-digital-conversion (ADC) for each individual pixel data occurs on-chip, allowing for high-speed, digitized readouts of valid pixel data from the imager itself [5]. A CMOS imager has a photo-detecting circuit and amplifier for each pixel arranged in a two-dimensional array. Thus the CMOS imagers collect and read out the pixels of the image in an array fashion. Figure 2-5 shows the arrangement of the pixel array structure in a typical CMOS imager. The Micron CMOS imager also incorporates a Bayer color filter array on top of the pixel array to support red, green and blue color readouts of the pixels. Figure 2-6 shows the pixel array readout for a normal Bayer pattern; the raw pixels from the pixel array are ultimately delivered in quads of two

green pixels, one red pixel, and one blue pixel.

As shown in Table 2.1, the ADC resolution for a pixel is 12 bits, and so each pixel value that is read out of the imager is 12 bits wide. The 12 bit pixel values are sent in parallel via bond wires from the imager on the focal plane to the camera electronics. One imager package has 48 pins used to interface with the electronics board, and twelve of the pins are used for the pixel data [8]. Therefore in one aperture, there are 528 bond wires used to connect the pixel data pins from all 44 imagers to the camera electronics. Other important pinouts from the imagers are the frame valid and line valid output pins, and they are driven high during active pixel read outs. In total, there are over 2000 bond wires used to connect the 44 imagers to the focal plane board of one aperture.

The pixel data is only valid when the frame valid and line valid signals are both asserted high. During an active frame valid and line valid, a new pixel is read out at every rising edge of the pixel clock, and pixels are read out row by row from the pixel array, with each row reading one pixel at a time. The imagers' readout of the raw pixel data reflects the Bayer filter color pattern as shown in Figure 2-6.

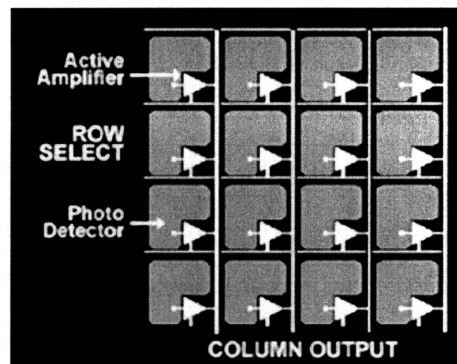


Figure 2-5: Pixel Array Structure of a CMOS Imager [7]

Bayer Pattern Color filter layout of pixels

R	G	R	G	Row 0
G	B	G	B	Row 1
R	G	R	G	Row 2
G	B	G	B	Row 3

Figure 2-6: Pixel readout for a normal Bayer pattern

2.2.2 Field-Programmable Gate Arrays and Camera Readout Electronics

The core MASIV camera electronics functionality is implemented with multiple Field-Programmable Gate Arrays (FPGA). Four Xilinx Virtex-II Pro FPGAs are used to manage the readout electronics of one aperture, for a total of sixteen FPGAs for the complete MASIV system. The reason for having four FPGAs per aperture is partly a result of the hardware layout of the aperture, and will be discussed in detail in Section 2.3.

The Xilinx FPGAs are high-performance programmable logic chips with large amounts of logic elements available in each device. The specific Xilinx FPGA devices used in the MASIV system are the XC2VP50 FPGAs. These Xilinx FPGAs have up to 88,192 internal registers/latches, 88,192 internal look-up tables (LUT) and two embedded IBM PowerPC processors within each FPGA [10]. In the MASIV system, the PowerPC is used as the interface for messaging and commanding between the aperture's electronics and the external world.

There are also up to 8 MB of Block RAM (BRAM) embedded in each Xilinx FPGA. In the MASIV system, these BRAMs are used as burst rate buffers to buffer the image data bursting off the CMOS imagers.

2.2.3 RocketIO Multi-Gigabit Transceivers

The RocketIO Multi-Gigabit Transceiver (MGT) is a Xilinx-defined transceiver used for high-speed serial data transfers. It is an embedded feature in the Xilinx FPGAs, and it supports a maximum data transfer rate of up to 3.125 Gigabits/second [9]. It is used in the MASIV system to transmit data from the camera electronics to the external devices. It is also used to receive commands and messages from the external devices to the camera electronics. There are also RocketIO MGT connections between the FPGAs for the data transfers and messaging that might occur between the FPGAs. The Xilinx FPGAs have up to 16 RocketIO MGT cores embedded into an FPGA, but the MASIV system currently employs only three RocketIO MGT cores per FPGA. Each RocketIO MGT core provides two connections - one for transmitting serial data transfers and one for receiving serial data transfers.

The RocketIO MGTs use a serial data transfer protocol defined by Xilinx. The embedded RocketIO core in the FPGA serializes any signal with a parallel bus width before transmitting it across the MGTs. Conversely, serialized signals received from the MGT are converted back to parallel bus signals by the RocketIO cores. The RocketIO MGTs use a standard 8b/10b encoding scheme: for every 8 bits of data sent through the transceivers, 2 bits are used as control characters [9]. The maximum clock rate that can be used with the RocketIO MGTs is 156 MHz.

The operation of the MASIV system is dependent on whether or not the image data can be read out of the camera electronics as fast they are being generated. Therefore, the essential threshold value to consider when designing the camera read-out electronics is the maximum data rate supported by the RocketIO MGTs. The system will only operate properly as long as the data coming off the apertures can be supported by the RocketIO MGT data rate.

2.3 Component Connections in the MASIV System

The connections between the hardware components in the MASIV system affect the operation of the aperture and how the readout architecture can be designed. Figure 2-7 shows an outline of the layout and connections of the major hardware components in one aperture of the MASIV system. The arrows represent the physical connections between the various components on the aperture board.

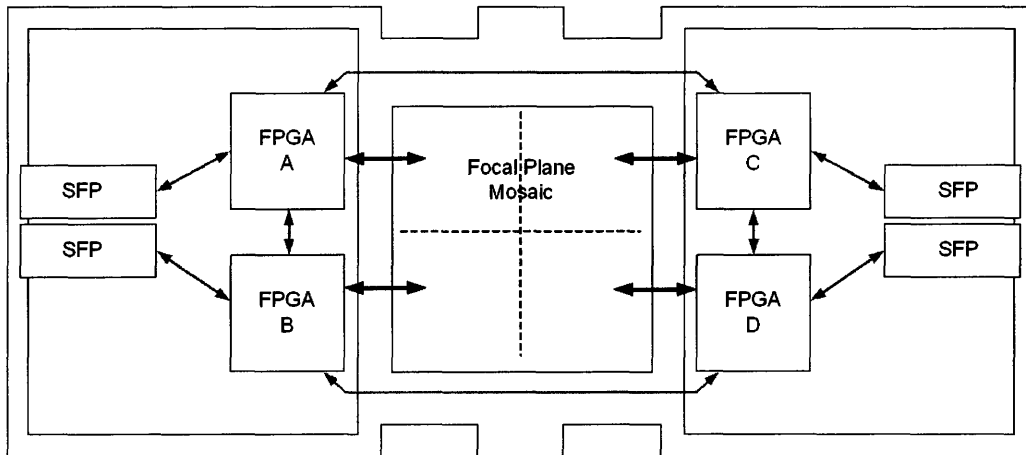


Figure 2-7: Hardware Layout for One Aperture [4]

Each FPGA in the current MASIV system uses three RocketIO MGT connections: one to interface the camera electronics with an external source via a small form-factor pluggable (SFP) device, and two to interface with the two neighboring FPGAs in the aperture. As demonstrated in Figure 2-7, the FPGAs are connected to each other using RocketIO MGTs in a ring fashion around the focal plane mosaic. The FPGA's function is to receive commands and messages from an external source, manage the occupying imagers, collect image data, and interface the data to the other components.

2.3.1 Quadrant Layout

The hardware layout for one aperture of the MASIV system can be thought of as being split into four identical quadrants. The four FPGAs on the aperture's circuit board each occupy and control one quadrant of the board. In Figure 2-7, the four FPGAs are labeled from "A" through "D," which corresponds that the four respective quadrants are "A" through "D." There are a total of sixteen quadrants spread out across four apertures in the complete MASIV system.

The quadrant layout stems from the symmetry of the layout of the CMOS imagers in the focal plane mosaic. The forty-four CMOS imagers are essentially split into quadrants of eleven imagers each as shown in Figure 2-8. Within each quadrant, the eleven imagers are indexed from "0" to "10." The eleven imagers in the quadrant are connected only to the corresponding FPGA of that quadrant. This means that a single FPGA will be controlling and handling the burst data payload for only the eleven imagers in its respective quadrant. The FPGAs control the imagers through a two-wire, Micron-defined serial communication protocol. The CMOS imagers transfer its pixel data to the camera electronics through twelve parallel bond wires as discussed in Section 2.2.1.

Notice that the aperture diagram shown in Figure 2-8 is rotated by 90 degrees from the aperture diagram shown in Figure 2-7. This is done to illustrate the direction in which the CMOS imagers are actually packed on to the focal plane. From Table 2.1, for one CMOS imager, there are 2,592 pixels in the horizontal direction, and 1,944 pixels in the vertical direction. Except for the case in which the corners of the focal plane are not populated, there are six CMOS imagers packed horizontally onto the focal plane, and eight CMOS imagers packed vertically, for a possible total of 15,552 raw pixels in both the horizontal and vertical directions of one focal plane mosaic.

2.3.2 Aperture Connections

The apertures of the MASIV system can be connected to each other or to other external devices via the small form-factor pluggable (SFP) transceivers. The SFP is

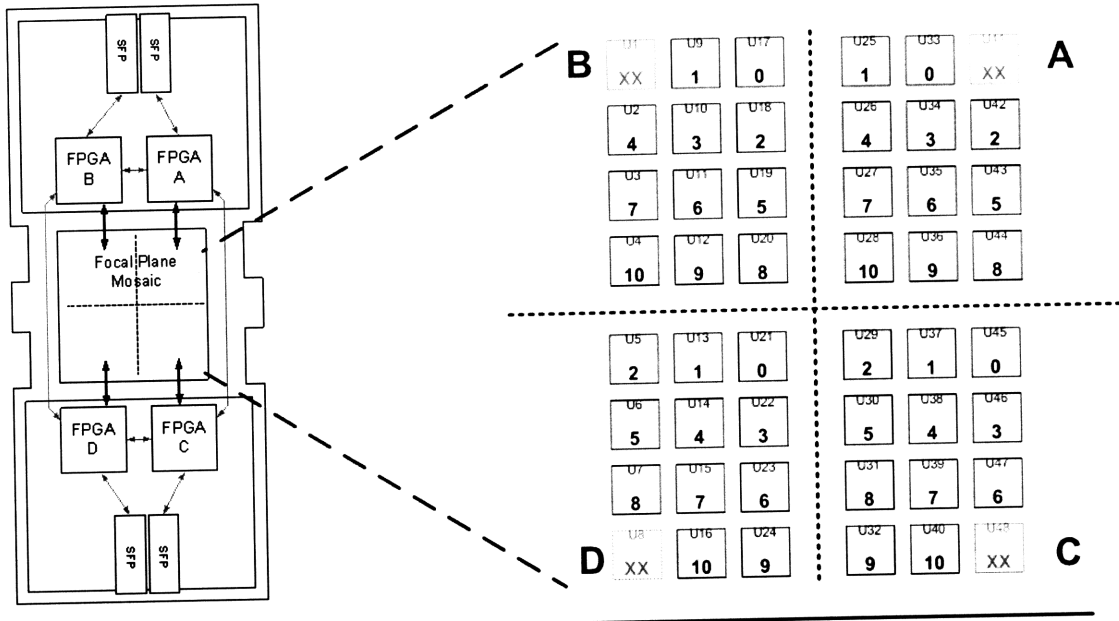


Figure 2-8: Quadrant Layout of the CMOS Imagers on Focal Plane Mosaic

an optical transceiver that can be used with the Rocket IO MGTs, and it interfaces the quadrant to a fiber optic cable. There is one SFP transceiver per quadrant, and it is used as the physical connection to an external source for the RocketIO MGTs for that particular quadrant. In the current MASIV implementation, each quadrant connects to an external device to output the respective quadrant's image data. There is no actual connection between the four apertures in the current MASIV implementation because data from the quadrants are explicitly outputted to the display terminal. Therefore there are a total of sixteen fiber optic cables coming out of the complete MASIV system to output the image data for storage on external disks.

However, for receiving commands and messages from an external source, each aperture has only one quadrant connected to the external source. The command or message is received by the quadrant and is passed down to the other quadrants in the aperture via a daisy chain. Figure 2-9 illustrates how the quadrants of the apertures are connected together and how information flows throughout the MASIV system. The solid arrows represent the connections used to output the image data from the CMOS imagers of the respective quadrants to an external source. The dashed arrows

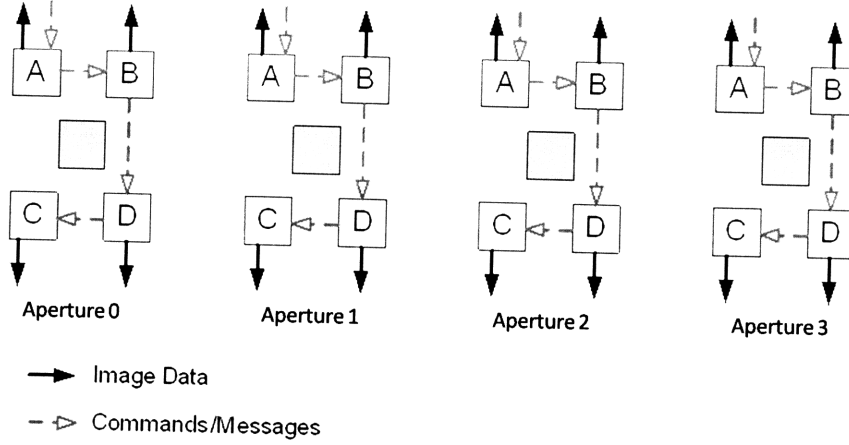


Figure 2-9: The RocketIO Connections of the Four Apertures of the Current MASIV Implementation

Table 2.2: Timing Specifications of the MASIV System

Timing Parameter	Value
Overall System Clock	100 MHz
RocketIO Clock	156 MHz
CMOS Imager Pixel Clock	25 MHz

represent the daisy chain connection used to send commands and messages sent from an external source down to the quadrants of one aperture.

2.4 Camera Electronics Operation and Specifications

The main system clock used by the camera electronics in all sixteen FPGAs of the MASIV system is 100 MHz. The clock for the RocketIO MGTs is in a different clock domain and runs at 156 MHz. The main system clock is not increased to match the RocketIO clock domain of 156 MHz because it is easier to meet timing requirements and specifications for the camera electronics at 100 MHz. These timing specifications are summarized in Table 2.2.

The minimum frame rate that is tolerable for wide-area persistent surveillance applications such as vehicle tracking is 2 Hz [1]. Therefore the current MASIV im-

plementation has an operating frame rate of 2 frames per second. The clock signal to the CMOS imagers is known as the pixel clock, and the pixel clock used in the current MASIV implementation is 25 MHz. Operating the CMOS imagers at a 25 MHz pixel clock enables the imagers to run at approximately 4 frames per second, but the camera electronics of the current MASIV implementation limits the output's final frame rate to 2 frames per second in order to minimize the number of frames written to the external disks. The maximum pixel clock that can be supported by the Micron CMOS imagers is 96 MHz, which is needed to run the imagers at the maximum frame rate of 15 frames per second when operating at full resolution. When operating the CMOS imagers at a lower resolution such as VGA, the frame rate can be increased up to 150 fps.

All 176 CMOS imagers in the MASIV system are synchronized. It uses an externally generated pulse per second signal as a reference signal to align the frame start signals of all the CMOS imagers. When a misalignment occurs, the camera electronics drops pixel clock cycles to the misaligned imager until it is realigned again.

Because the imagers are aligned, all the image data from the CMOS imagers are sent to the FPGAs at the same time. Therefore the FPGA must be able to handle all the data coming off the eleven imagers from the quadrant. The current MASIV implementation uses both BRAMs and DDR RAM to buffer the image data coming from the imagers.

Chapter 3

Design and Performance

Considerations

Virtual Pan-Tilt-Zoom allows a user to cover a wide coverage area without being limited to the physical constraints provided by a mechanical PTZ system. The current MASIV implementation outputs all 880 Megapixels of raw data continuously at 2 Hz, but by reading out only the data relevant to a user's Region of Interest, certain performance capabilities of the system can be improved.

There are three primary readout architectures that are considered for VPTZ functionality, and they differ in which component does the data decimation. The first approach considered is to decimate the data after the image data comes off the cameras. This will be identified as the Display Level, and it is an abstraction for the external devices that accept the image data coming out of the physical cameras of the MASIV system. The decimation at the Display Level can be done in software in the display terminal or anything else that can process and store the information. The second two approaches considered are implemented within the camera electronics of the MASIV system: CMOS Imager Level or the FPGA Level.

The performance metrics of the VPTZ implementation will vary depending on which approach is used. This chapter will examine the performances and tradeoffs of the various different readout architectures that are possible for VPTZ.

3.1 VPTZ Specifications

3.1.1 Data Decimation

The types of data decimation that must be considered for the VPTZ application are windowing and subsampling. The windowing process is the foundation for the virtual panning and tilting functionalities in VPTZ, while the subsampling process is the foundation for the virtual zooming functionality in VPTZ. It is deciding which components of the MASIV system should undertake this windowing and subsampling decimation for the VPTZ that will determine the different performance capabilities of the system.

The windowing process determines the field-of-view for the user's Region of Interest from the available coverage area of the system. Any part of the image data that does not fall under the windowed dimensions of a user's Region of Interest is decimated.

The subsampling process is used to reduce the image resolution. There are several ways to subsample the image. The two considered for this project are skipping and binning. The skipping process reduces the image resolution by not sampling entire rows or columns of pixels and only using the selected pixels to form the field-of view. In order to maintain the Bayer pattern color-filters used in the Micron CMOS imagers (see Figure 2-6), the pixel skipping process is performed in pairs of pixels. This means that a pair of pixels is read out before a variable number of pairs of pixels are skipped for the final output image. The binning process also outputs only a selected number of pixels, and it maintains the pattern of the Bayer filter readout as shown in Figure 2-6 by taking the adjacent same-color pixels that were to be skipped and combining them into one output pixel. This combination process can be an average or summing process. Both binning and skipping downsample and reduce the pixel count for the output image.

Subsampling the image using the binning process does not cause aliasing because the averaging of the nearby pixels acts as a low-pass filter. When using the binning process, the pixels that are read out are not the raw pixel data of each individual

color as shown in Figure 2-6, but instead a low-pass filtered readout of the pixel array. Subsampling the image using the skipping process maintains the readout of the raw pixels thus potentially causing aliasing of the image. However, the aliasing is not significant and can be mitigated by various higher level processes such as digital filtering and demosaicing in the software. Performing the skipping process in firmware and then using the aforementioned anti-aliasing techniques at a higher level reduces the amount of firmware that would need to be supported by the camera electronics because the camera electronics would only have to provide the raw data to the end user. In this thesis project, we only examine the skipping process in order to simplify the readout architecture so that the camera electronics only has to read out the raw pixels without performing the low-pass filtering. Future work could prevent aliasing in the readout architecture by implementing a binning process in the camera electronics.

3.1.2 Data Flow

At the highest level, the image data always flows from the CMOS imagers to the FPGAs to the Display Level. Figure 3-1 represents this generalized concept view of the data flow. All the different readout architectures will have to follow this general data flow as the physical layout and hardware connections in the MASIV system are of this manner. In Figure 3-1, there are four big boxes labeled from *Aperture 0* to *Aperture 3*, which represents the four apertures of the MASIV system. Inside the big boxes, there are two sets of four boxes. The first set of boxes represents the imagers and the second set of boxes represents the FPGAs. The image data from the CMOS imagers are sent to the FPGA of its respective quadrant. Finally, the image data is sent from the FPGAs to the external display.

As discussed in Section 2.3.2, the apertures of the MASIV system can be connected using the RocketIO MGTs via fiber cables. The way the apertures are connected together affects the specifics of the data flow of the overall system and can also affect certain performance metrics of the MASIV system.

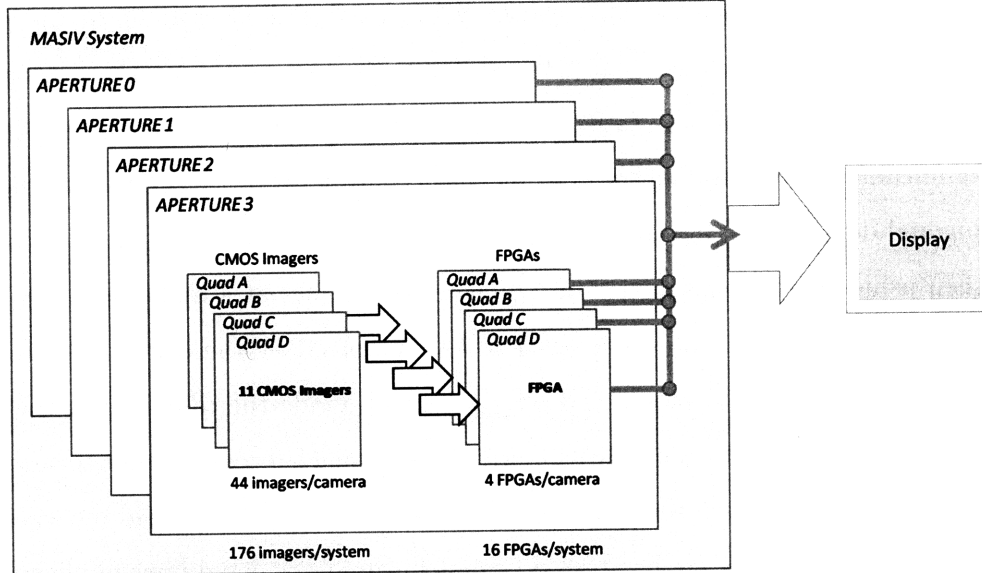


Figure 3-1: A simplified high level view of the data flow.

3.1.3 Self-Imposed Specifications and Digital Zoom

Considering the resolution support for many modern-day displays and monitors, we will limit the image size of a user's Region of Interest. The size of the Region of Interest for a particular user will be kept constant at 1 Megapixel (1024 pixels x 1024 pixels). By limiting the size of the Region of Interest to exactly 1 Megapixel, we can reduce some of the complexities in the readout architecture design. This is a reasonable limitation because 1 Megapixel is a sufficient Region of Interest size for one user considering the limitations in resolution size for typical modern-day displays. For example, most modern-day displays only support a resolution of a little over 1 Megapixels (e.g. monitor displays with 1280 pixels x 1024 pixels).

By keeping a constant image size, the field-of-view is increased when the image is subsampled. This is because subsampling processes such as skipping reduces the pixel count of an image without changing the field-of-view, but when the Region of Interest must always output a constant number of pixels, more pixels must be read out in order to compensate for the decimated pixels. These extra pixels that are read out correspond to an area that is greater than the original field-of-view of the Region of Interest prior to the subsampling. Hence subsampling and keeping the pixel count

constant for a Region of Interest gives the effect of digitally zooming out to view a wider area of the visual field.

3.2 Performance Metrics of the Readout Architectures

Table 3.1 presents an overview of the different readout architectures and performance metrics associated with each approach. The different readout architectures are inherently grouped into three categories, which are sorted by the component that is doing the VPTZ decimation: *Display Level*, *FPGA Level*, and *CMOS Imager Level*. Under each category, the different operating modes are tabulated. These operating modes may differ in how the apertures are connected, frame rate, number of users, etc. The different operating modes can also require different readout architecture designs depending on the specifications.

There are two performance metrics that we are concerned with when comparing the different readout architectures. The first is the frame rate of the MASIV system, which is how many image frames can be delivered to a user per second. The second is the number of users that can simultaneously use the VPTZ functionality for the MASIV system. The two metrics sometimes can have an indirect relationship with each other, and different scenarios will usually tradeoff between the two metrics. For example, a design that has a higher frame rate may not be able to support as many users as one with a lower frame rate.

The columns on the right in Table 3.1 list the parameters and specifications for the different readout architectures. The two right-most columns in Table 3.1 are titled "Frame Rate" and "# of Users." As the column's title suggests, these columns lists the calculations for the frame rate and number of users for the system. When calculating the frame rate or the number of multiple users, one of the values is held constant in order to calculate the other value. The essential threshold value to consider in the system is the maximum data rate value of the RocketIO MGTs, which is

Table 3.1: Calculation of Performance Metrics

Architecture	# of External Fibers (in pairs)	Frame Rate (fps)	# of Users
<i>DISPLAY LEVEL</i>			
	16	<i>~4.69</i>	large, server-side limited
	1	<i>~.3</i>	large, server-side limited
<i>FPGA Level</i>			
Mode 1	16	15	<i>16 per quadrant</i>
Mode 2	16	2	<i>126 per quadrant</i>
Mode 3	1	15	<i>16 per system</i>
<i>CMOS IMAGER Level</i>			
1 user		<i>~55.48 to 157.08</i>	1
n user on one imager, same skip mode		<i>~15 to 55.48</i>	n
n user on one imager, different skip mode		$\frac{\text{Variable_Frame_rate}}{n}$	n

*Fixed parameters in normal font. Calculated parameters in *italics*.

*Assumptions and other Parameters: No overlapping users, Read out Bandwidth Limit = 3.125 Gb/s,
Region of Interest = 1 Megapixel

approximately 3.125 Gbits/second. By holding either the frame rate or the number of users fixed, we can figure out what the maximum value for the other parameter can be and still meet the data rate requirements for the system. To distinguish between the calculated value and the fixed value in Table 3.1, the calculated value is italicized whereas the fixed value is in normal font. The MASIV system will be able to function as long as the data rate of the system is equal to or less than the maximum data rate value of the RocketIO MGTs.

The "# of External Fibers" in Table 3.1 is the number of fiber cables that are used to connect the MASIV system to the Display Level for image data transfer. For commands and messages to the MASIV camera electronics, there are four fiber channel connecting the external source to the apertures as described in Figure 2-9 of Section 2.3.2. We will only be investigating the situation when the number of fiber cables used to connect the MASIV system to the Display Level for image data transfers is 16 or 1 because these two cases are the two limiting cases for the system. The number of external fibers is an effect of how the apertures are connected to each other in the MASIV system. If the number of external fibers is 16, this means that each of the sixteen quadrants of the MASIV system is directly connected to the Display Level. There is no data flow of image data from one aperture to another because all the image data from the respective quadrants are directly transferred to the Display Level. Thus, the readout architecture for this design can be simplified to be developed strictly on a per quadrant basis, but the system would have to physically manage implementing sixteen fiber cables with receivers at the display. If the number of external fibers is 1, this means that there is only one fiber cable connecting the entire MASIV system to the Display Level. The apertures are connected such that all sixteen quadrants of the MASIV system are daisy chained, with the final quadrant from the final aperture being connected to the Display Level. The image data from one end of the daisy chain will have to go through the quadrants of the MASIV system before it can be sent to the Display Level. As a result, the readout architecture for this setup is more complex but reduces the physical clutter of having to handle sixteen fiber cables coming out of the MASIV system. Figure 3-2 shows an example of the

system having 16 external fibers versus the system having only one external fiber for image data output.

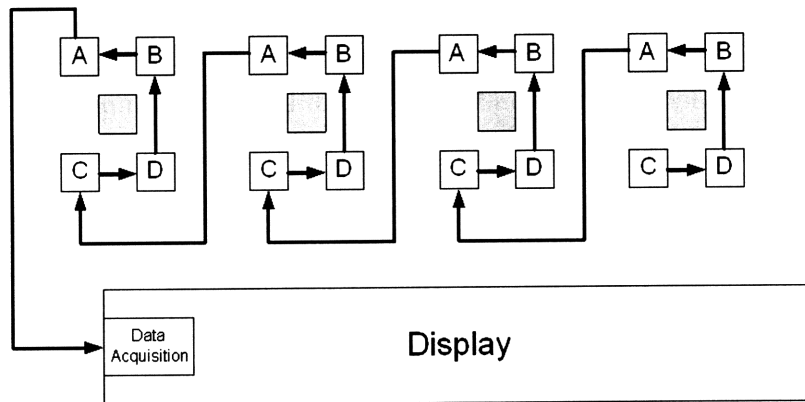
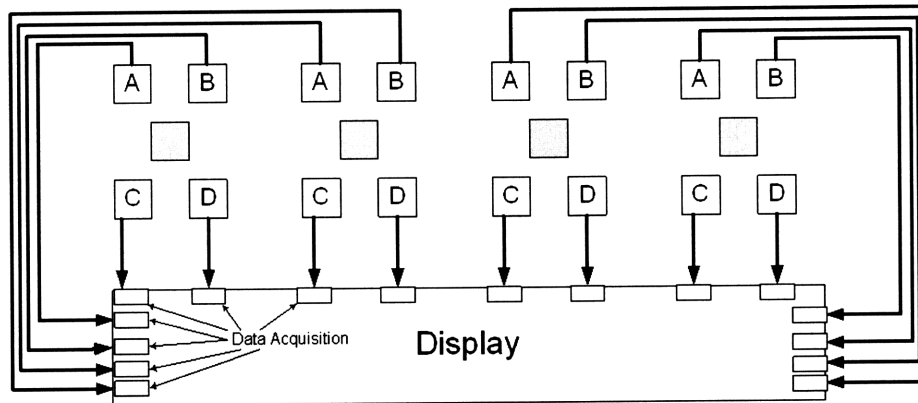


Figure 3-2: Sixteen External Cables Connections vs. One External Cable Connection for Image Data Transfer.

The number of external fibers can also affect the frame rate of the system or the number of users that can be supported by the system. Depending on the readout architecture, reducing the number of external fibers has the potential to decrease the frame rate of the MASIV system or change the number of users that can be supported by the system.

3.3 Different Readout Architectures

This section will study the performances and tradeoffs of each of the three different readout architectures more thoroughly. The three different readout architectures perform the VPTZ decimation at the Display Level, FPGA Level, or the CMOS Imager Level. Within each of the three architectures, there are differences depending on the parameters and specifications of the MASIV system, as well as depending on the different user scenarios.

3.3.1 Display Level VPTZ Decimation

One possible design approach is to have all the decimation for the VPTZ done at the Display Level. The Display Level decimation would most likely occur in software defined functions in a computer. Therefore, in order for Display Level VPTZ decimation to work, it will need to be able to access the data from the entire MASIV camera system. All the CMOS imagers in the MASIV system would have to output at its maximum resolution, and then the software decimation functions at the Display will perform the necessary decimation processes. Even for a 1 Megapixel Region of Interest image size, the MASIV camera electronics would still be required to read out all 880 Megapixels worth of data to the software decimation function in order for it to perform VPTZ. Figure 3-3 shows a high level block diagram for this readout architecture. Though this architecture would be the least complex to implement and could easily scale to many users, there would be several issues associated with this approach.

The major issue with trying to do all the decimation at the Display Level is the data throughput. The readout architecture would have to handle and buffer the data being generated. The data leaving the cameras will have to be faster than the data being generated by the cameras in order to successfully output all 880 Megapixels worth of data. Using the VPTZ functionality with this readout architecture would not reduce the bandwidth requirements in anyway because all the pixel data being generated would have to be outputted by the MASIV camera electronics. Therefore

Decimation at Display Level

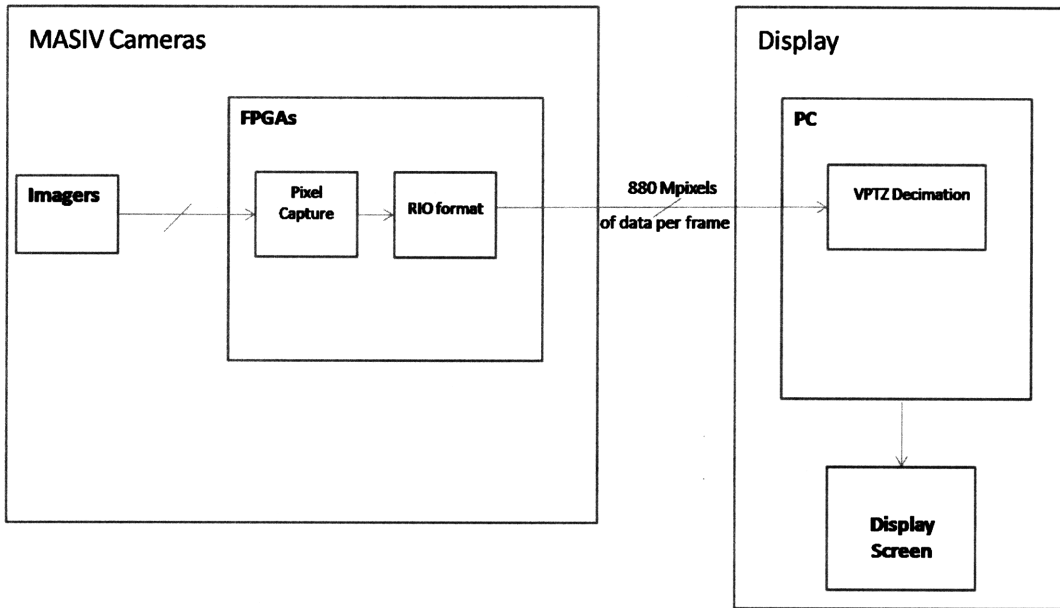


Figure 3-3: Block Diagram of Readout Architecture for Display Level Decimation.

this readout architecture cannot fully benefit from the fact that VPTZ with a constant Region of Interest size only requires a fraction of the complete image data.

Table 3.1 shows the maximum frame that is possible for Display Level VPTZ decimation. When the MASIV system uses 16 external fibers, the maximum frame rate is approximately 4.69 fps. When the MASIV system uses 1 external fiber, the maximum frame rate is cut to be a sixteenth of the frame rate, which is approximately .3 fps. This is because all the data that was being outputted by the sixteen fibers now have to fit onto a single fiber cable, thus reducing the frame rate of the overall system. This frame rate is not suitable for persistent surveillance applications, and thus the current MASIV implementation uses 16 fiber channels to output the image data ¹.

The frame rate for the MASIV system for Display Level decimation using 16

¹The current MASIV implementation's camera electronics actually reduces the overall system's frame rate from the possible ~4 fps down to the minimum required 2 fps in order to limit the number of frames written to disk. See Section 2.4 for details.

external fibers is calculated using the RocketIO's maximum data rate:

$$\begin{aligned}
 fps &= \frac{Rate_{MGT}}{N_{pix_imager} * N_{bit} * I} \\
 Rate_{MGT} &= \text{Maximum RocketIO MGT Rate [Gbits/sec]} \\
 I &= \text{\# of Imagers per Quadrant [imagers/quadrant]} \\
 N_{pix_imager} &= \text{\# of Valid Pixels per imager [pixels/imager]} \\
 N_{bit} &= \text{ADC pixel bit width} = 12 \text{ [bits/pixel]}
 \end{aligned}$$

One benefit of the Display Level VPTZ decimation is that it allows for a large number of users. Because all the data is available for the VPTZ decimation processes at the Display Level, it can be duplicated as necessary without penalty for the multiple users, as long as the external server can support it.

To summarize, developing the VPTZ functionality for Display Level decimation would be the easiest to implement because all the data that could possibly be generated by the cameras is already available, and it is relatively simple for software processes to decimate through the data. However, data throughput from the cameras remains a constraint, and there is no gain in frame rate performance for Display Level decimation. Thus, the frame rates being generated by the system will be low.

3.3.2 FPGA Level VPTZ Decimation

Another possible readout architecture would perform the VPTZ decimation at the FPGA Level of the MASIV system. In FPGA Level VPTZ decimation, the FPGA would decimate all the extraneous data not requested by the users, and only output the relevant pixel data for the requested Regions of Interests. As in Display Level decimation, the bottleneck for data transfers occurs at the RocketIO MGTs, but because only a fraction of the total pixels need to be transmitted across the MGTs for FPGA Level decimation, higher frame rates can be supported by the bandwidth in this readout architecture.

Similar to Display Level VPTZ decimation, VPTZ decimation by the FPGA requires the CMOS imagers to be read out at full resolution. Therefore the maximum system frame rate that can be achieved is limited by the frame rate of the imagers operating at full resolution. However, the difference between the two is that in FPGA Level VPTZ decimation, the FPGA would filter out the unnecessary pixels before sending the relevant pixel information to the RocketIO MGTs. By the time the data reaches the Display, only the relevant pixel data that was requested by the user will remain, as opposed to the entire 880 Megapixel data set. This greatly eases the data throughput levels at the RocketIO MGTs as well as the server-side requirements at the Display. Figure 3-4 shows a high level block diagram for this readout architecture.

Decimation in FPGA

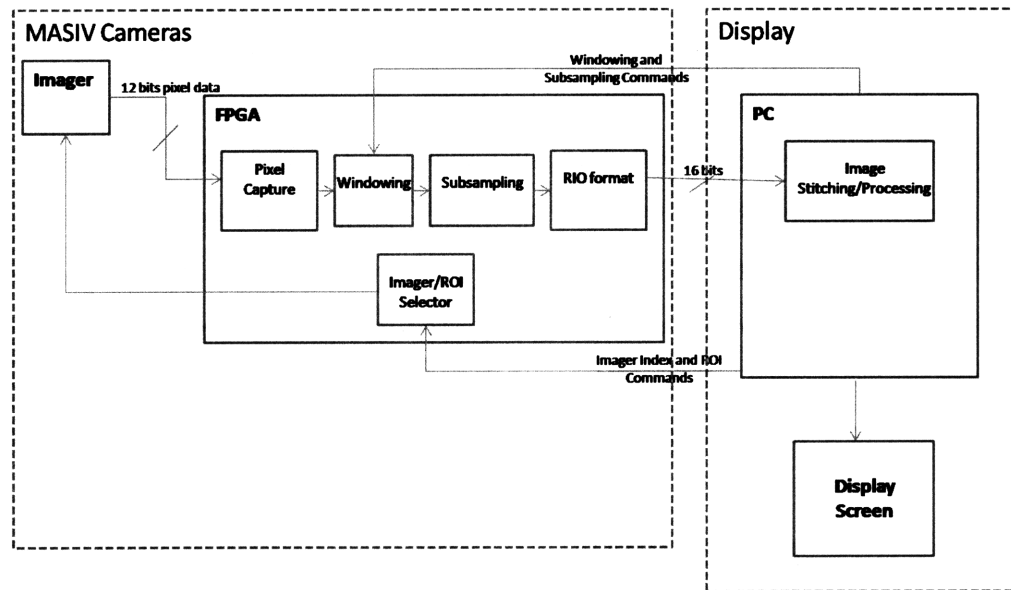


Figure 3-4: Block Diagram of Readout Architecture for FPGA Level VPTZ Decimation

The frame rates for FPGA Level VPTZ decimation is shown in Table 3.1. For the cases where there are 16 external fibers, the frame rate is held constant in order to calculate the number of users that can be supported by the data rate limits of one quadrant. We are interested in the case when all the users' Regions of Interests are

encompassed in one quadrant because this is the limiting case of this architecture. The frame rate of the system used in the calculations is 15 fps, which is the maximum frame rate of the CMOS imager when it is being read out at full resolution ². For the calculations, it is also assumed that there are no overlapping Regions of Interests between the different users in the quadrant because overlapping Regions of Interests do not contribute to the discussion of the limiting case of this architecture. For example, if many users' Regions of Interests are completely overlapping each other as shown in Figure 3-7, the readout architecture can theoretically just treat it as one Region of Interest.

From this, we can determine the maximum number of users in one quadrant that this particular architecture can support:

$$\begin{aligned}
 N_{user} &= \frac{Rate_{MGT}}{N_{pix_ROI} * N_{bit} * fps} \\
 N_{user} &= \# \text{ of Users [users]} \\
 fps &= \text{frames per second} \\
 Rate_{MGT} &= \text{Maximum RocketIO MGT Rate [Gbits/sec]} \\
 N_{pix_ROI} &= \# \text{ of Valid Pixels per ROI [pixels/frame]} \\
 N_{bit} &= \text{ADC pixel bit width} = 12 \text{ [bits/pixel]}
 \end{aligned}$$

Mode 1 in Table 3.1 describes a system with 16 external fiber cables and the imagers operating at the maximum frame rate of 15 fps. Assuming there are no overlapping Regions of Interests, Mode 1 can support up to approximately sixteen users for one quadrant. Therefore, the entire system, which consists of sixteen quadrants, can theoretically support up to 256 users, as long as the system does not exceed sixteen users per quadrant.

If we maintain the 16 external fibers and reduce the operating frame rate of the imagers, even more users can be supported by the system because more bandwidth is

²The baseline MASIV implementation only employs a 25 MHz pixel clock. See Section 2.4 for details.

now available. The frame rate of the CMOS imager can be reduced by decreasing the speed of the pixel clock to the CMOS imager. Mode 2 in Table 3.1 describes a system with the imagers operating at 2 fps, which is the frame rate for the current MASIV implementation. In this mode, the maximum number of users it can theoretically support is increased to approximately 126 users per quadrant.

If a user's Region of Interest spans across quadrants, the frame rate of the system can theoretically be increased because the amount of data that needs to be read out of each quadrant is reduced. For example, if a user's Region of Interest is evenly spaced across two quadrants, then the amount of data that needs to be read out of each quadrant is reduced by half. As a result, more bandwidth is available to support faster frame rates.

If the number of external fibers used for image data transfers is reduced to one fiber cable as shown in Figure 3-2, the performance of the entire system will always be the same as the limiting case of one quadrant in the sixteen external fiber modes. This is because all the image data of the system must pass out of one quadrant. Mode 3 in Table 3.1 describes the system with only one external fiber cable and the imagers operating at the maximum frame rate of 15 fps. In this case, the number of users the entire system can support is sixteen users.

As a result of these artifacts that occur from having multiple users, implementing VPTZ to multiple users for FPGA Level VPTZ decimation becomes more complex than implementing multiple users for Display Level VPTZ decimation. In Display Level VPTZ decimation, the number of users that can be supported is large and there are no side effects to the camera electronics from having more users. However, in FPGA Level VPTZ decimation, the number of multiple users is fundamentally limited by the data rate of the system, and increasing the number of users can reduce the frame rate of the system.

The frame rate of the MASIV system is fundamentally linked to the frame rate of the CMOS imagers. With FPGA Level VPTZ decimation, the frame rate of the system is limited to the frame rate of the imager running at full resolution because all the CMOS imagers must be operating at full resolution for FPGA Level decimation.

3.3.3 Imager Level VPTZ Decimation

The CMOS imagers used in the MASIV system has the ability to perform many camera processing functions on the chip itself. VPTZ decimation can be implemented at the CMOS Imager Level by making use of these on-chip functions. The imagers can be read out at a lower resolution depending on the requested Region of Interest, and thus the operating frame rate of the imagers can be increased to greater than 15 fps.

In Imager Level VPTZ decimation, the FPGA still plays a big role in the readout architecture because the FPGA acts as the controls for the CMOS imagers. The FPGA can be thought of as the "brains" of the CMOS imager because the FPGA commands the CMOS imager chip to perform the certain decimation functions such as windowing and subsampling by configuring the value of the corresponding register address space on the CMOS imager chip. Additionally, the data from the CMOS imager still flows through the FPGA before it is transferred to the external display terminal, just as it was for the Display Level and FPGA Level VPTZ decimation.

Figure 3-5 shows a high level block diagram view for the readout architecture required to do Imager Level VPTZ decimation. Higher level software processes in the display terminal determines the Region of Interest of the user and sends that information down to the FPGAs of the MASIV system. The FPGA then sends the corresponding windowing and subsampling commands for the user's Region of Interest to the CMOS imagers. Finally, the imagers do the VPTZ decimation before transmitting out the relevant pixel data.

Since the windowing and subsampling functions of the VPTZ are performed on the CMOS imager chip itself, only the necessary pixel data are sampled and sent to the FPGA and across the fiber cables. Therefore, the amount of data bursting off of the imagers to the FPGA is greatly reduced. Additionally, this design could allow for much faster rates because the CMOS imagers do not need to always be read out at full resolution.

Depending on different scenarios, the frame rate can change considerably, espe-

Decimation in Imager

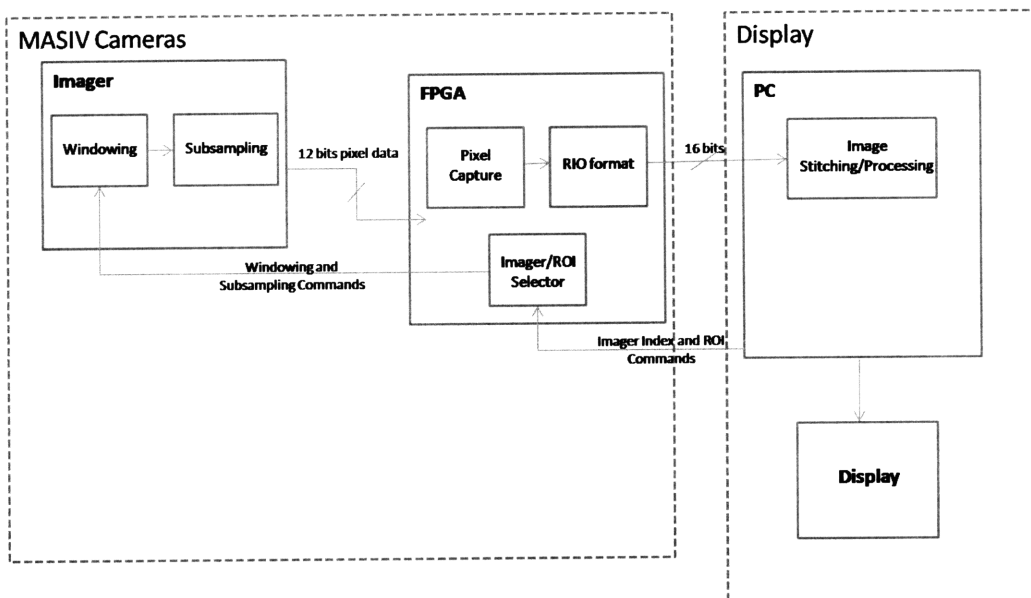


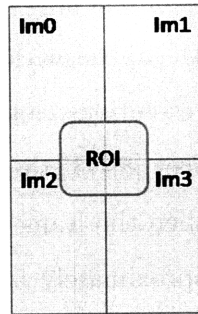
Figure 3-5: Block Diagram of Readout Architecture for VPTZ Decimation in the Imagers.

cially when multiple users are considered. In Table 3.1, we examine and calculate the performance metrics for a few of these possible scenarios. The first scenario is if there is only one user that has a Region of Interest of 1 Megapixel with no subsampling. The frame rate for the CMOS imagers at different resolutions can be calculated by using the formulas from the Micron Imager data sheet [8]. If the user's Region of Interest happens to fall within one CMOS imager, then the frame rate that can be achieved for a 1 Megapixel Region of Interest is approximately 55.48 fps. This is because if the Region of Interest has a size of 1 Megapixel, then the CMOS imager would only be required to read out at a resolution of 1 Megapixel, which would have a frame rate of 55.48 fps.

In the above scenario, the frame rate of the imager can actually be increased if the Region of Interest of the user spans across multiple CMOS imagers. Figure 3-6 shows a couple of examples in which a user's Region of Interest can span across multiple imagers. A user's Region of Interest can span across multiple CMOS imagers because of the nature of the focal plane array of the MASIV system as described in Chapter 2. Note that the four imagers represented in Figure 3-6 - *Im0* to *Im3* - are from different apertures. This is because of the layout of the imagers at the focal plane array and how the imagers are eventually combined together to create the final image. If the Region of Interest spans across multiple CMOS imagers, the resolution required from each individual active CMOS imager will only be a fraction of the Region of Interest. If the user's Region of Interest is limited to be 1 Megapixel, and that region spans across two CMOS imagers, then the resolution that will need to be read out from each of those two CMOS imagers will be less than 1 Megapixel. In the case that the Region of Interest is evenly spaced across the CMOS imagers, the maximum frame rate can reach up to 157 fps for a 1 Megapixel Region of Interest spanning evenly across multiple imagers.

Adding in multiple users for CMOS Imager Level VPTZ decimation quickly increases the complexity of the system. Let us begin by examining the case of having n multiple users with no subsampling on only one CMOS imager. This means that each of Regions of Interests for the n users will be within the area covered by one

ROI Spans 4 CMOS imagers



ROI Spans 2 CMOS imagers

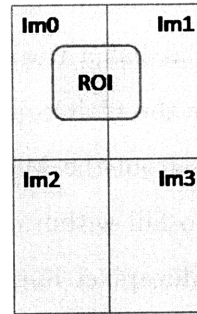


Figure 3-6: An example of a Region of Interest spanning multiple imagers.

CMOS imager. We will examine three cases that can occur in this scenario. Figure 3-7 gives a visual overview of these three situations.

The cases presented in Figure 3-7 lead to different performances because of the way frame sizing works in the CMOS imager. The windowing functionality for VPTZ decimation in the CMOS imager is essentially configuring the frame start and frame size values of the CMOS imager. The frame start values determine where in the pixel array to begin the readout of the CMOS imager, and the frame size determines how much of the pixel array to read out. Therefore, the readout of the pixel data from the CMOS imager is inherently of a rectangular shape. In the first case presented in Figure 3-7, none of the Regions of Interests of the users overlap each other, and thus

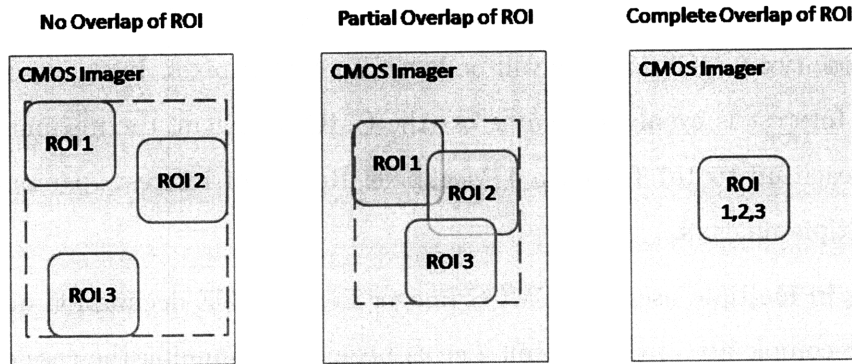


Figure 3-7: Multiple Regions of Interests scenarios with same zoom mode within one CMOS imager.

the "shape" of all the valid pixels that need to be read out from the CMOS imager is not a rectangle. There are two possible approaches to handle this case.

The first approach would be to read out the entire CMOS imager at full resolution, and then select out the necessary pixel data for the different Regions of Interests in the FPGA. In this approach, the frame rate of the CMOS imager will automatically be limited to the maximum frame rate of reading the imager out at full resolution, which is 15 fps.

The second approach makes use of the fact that the CMOS imager reads out the pixel array row by row from top to bottom. If we assume that the direction of the read out of the pixel array begins at the top left corner of the CMOS imager and ends at the bottom right corner, then we can determine a rectangular region that will encompass all the Regions of Interests on the CMOS imager. By figuring out the location of the Region of Interest that is closest to the top of the CMOS imager and the location of the Region of Interest that is closest to the left side of the CMOS imager, we can determine the start of the rectangular region. Conversely, by figuring out the location of the Region of Interest that is closest to the bottom of the CMOS imager and the location of the Region of Interest that is closest to the right side of the CMOS imager, we can determine the end of the rectangular region. The dotted line around the three Regions of Interests for the first case of Figure 3-7 represents this rectangular region. In this approach, only this rectangular region will need to be read out instead of the entire resolution of the CMOS imager. Because there are areas of the rectangular region that are not a part of the Regions of Interests, the FPGA will still have to be used to select out the pixel data for the Regions of Interests from the rectangular region. Even still, this approach will be more efficient than having to always read out the CMOS imager at full resolution, thus enabling faster frame rates.

The second case presented in Figure 3-7 has the Regions of Interests partially overlapping each other within one CMOS imager. The approach to handle this case is actually the same as before: a rectangular region that will encompass all the Regions of Interests is determined for read out. However, the rectangular region for this case will be smaller than in the previous case because there will exist pixels that are

common to more than one Region of Interest. As a result, this case will have faster frame rates than in the previous case because the readout of the rectangular region from the CMOS imager is smaller.

The third case presented in Figure 3-7 has the Regions of Interests completely overlapping with each other. In this case, the different Regions of Interests each completely share the same pixels. In essence, the size of the rectangular region that encompasses the multiple Regions of Interests in the previous two cases is now the size of one Region of Interest. Therefore, this case can be thought of as having only one Region of Interest. For an Region of Interest of 1 Megapixel, the maximum frame rate would be up to 55.48 fps.

The above three cases limited all the Regions of Interests to be within one imager with no subsampling. As soon as subsampling is introduced and the Regions of Interests are allowed to span across multiple imagers, the complexity increases further. First of all, when multiple Regions of Interests with different subsampling is introduced, the approach of defining a rectangular region that can encompass the multiple Regions of Interests becomes very complex. When the CMOS imager needs to be subsampled, the whole imager must be subsampled at that same subsampled mode; one cannot have a portion of the CMOS imager readout be subsampled at one mode while another portion of the imager subsampled in a different mode. Thus, if two Regions of Interests with different subsampling modes exist within the same CMOS imager, the imager must take turns switching back and forth between the different subsampling values of the different Regions of Interests. Therefore, as Table 3.1 shows, the frame rate for this scenario would be divided by the number of Regions of Interests with different subsampling modes. Moreover, the frame rate value that is being divided by the number of different Regions of Interests would actually be variable because the frame would dynamically change as the CMOS imager switches back and forth between the different subsampling modes.

In summary, while Imager Level VPTZ decimation offers the highest potential frame rate for the MASIV system, it comes at a substantial complexity cost. There are many special cases to handle, and each different case leads to different performances.

Table 3.2: Qualitative Comparison of the Different Architectures

	Implementation Complexity	# of Users Supported	Frame Rate
Display Level Decimation	Low Complexity	Large, Limited by Server-Side	Slowest
FPGA Decimation	Moderate Complexity	Limited by RocketIO bandwidth	Moderate
Imager Decimation	High Complexity	Limited by RocketIO bandwidth	Fastest

3.4 Summary of the Differences between the Readout Architectures

Table 3.2 provides a qualitative comparison of the different readout architectures discussed in this chapter.

Among the three design candidates for the readout architecture, the architecture for CMOS Imager Level VPTZ decimation would be the most complex to implement. This is because the nature of the readout process for the imagers makes handling multiple users very complex. However, this architecture has the potential for achieving the highest frame rate performance amongst the three readout architectures.

Both Display Level and FPGA Level decimation require the full-resolution readout of the CMOS imager, and thus the frame rate of the system is limited by the imager’s full-resolution frame rate. However, the readout architecture implementation is less complex to implement than Imager Level decimation because all the pixel data is available to the component performing the VPTZ decimation.

Display Level decimation enables support for a large number of users because the software performing the decimation has access to all the image data and can reproduce data for multiple users as necessary without penalty. However, the frame rate is severely limited in order to meet the RocketIO bandwidth limitations of outputting all 880 Megapixels worth of data from the apertures.

For FPGA Level decimation, it is easier to implement multiple user functionality

than Imager Level decimation because the FPGAs have access to all the pixel data. It can achieve faster frame rates than Display Level decimation because the Regions of Interests are limited to only 1 Megapixel, thus allowing for more bandwidth allocation than having to transfer all 880 Megapixels. However, the architecture for FPGA decimation is still fundamentally limited by the RocketIO bandwidth limitations, and multiple users can only be supported as long as all data rate requirements can be met.

In this project, we implement a readout architecture using FPGA Level decimation because it is a reasonable compromise between complexity and performance. The amount of data being read out of the system is greatly reduced, and thus the system's frame rate can be increased to meet the imager's operating frame rate of ~ 4 fps at the current pixel clock rate. As the pixel clock rate to the imagers is increased, the frame rate of the system can scale up to the maximum frame rate of the imager running at full resolution.

Chapter 4

Specific Implementation of a Camera Readout Architecture for VPTZ

The camera readout electronics for the VPTZ application are implemented with programmable logic in the FPGAs. This chapter will describe the logic implemented in the FPGA for a specific readout architecture that demonstrates the concepts of VPTZ on the MASIV system. All the firmware logic was implemented using VHDL, and was developed and compiled in the Xilinx ISE 8.2 environment. All the VHDL source code modules for the VPTZ readout architecture are included in Appendix A.

The specific readout architecture that is implemented in this thesis is the FPGA Level VPTZ decimation as described in Section 3.3.2 of Chapter 3. All the specifications and design decisions for the implemented architecture will be discussed in this chapter.

4.1 Existing MASIV Firmware Overview

The VPTZ application is essentially implementing an alternative readout architecture for the MASIV system in lieu of the current readout architecture. Thus, existing firmware used for other applications such as imager controls, housekeeping, clock

signal control and RocketIO interfacing can still be used with the readout architecture for VPTZ.

We will briefly outline some of the important existing firmware modules in the camera electronics developed at Lincoln Laboratory for the current MASIV implementation that the VPTZ readout architecture also relies on.

4.1.1 Imager Control Module

The imager control module interfaces with the eleven CMOS imagers of the quadrant. Using the 100 MHz master system clock, the imager control module generates the 25 MHz pixel clock signal to the imagers by sampling the master system clock on every fourth clock cycle. The imager control also uses an externally generated pulse per second signal to make sure that the operation of all the imagers in the quadrant are synchronized together. The pulse per second signal is used as a reference sync to accurately time-stamp new frames. If an imager happens to be misaligned, the imager control module will drop the pixel clock cycles to that misaligned imager until it is aligned again. The imager control module also detects to see if any of the eleven imagers in the quadrant have failed or malfunctioned. If a failed imager exists, then the imager control module disables that bad imager from the rest of the camera electronics.

The imager control module reads in and registers the frame valid, line valid, and twelve-bit data bus signal from each of the eleven imagers. Since all the imagers are aligned together, the imager control module selects the frame valid and line valid signals from one of the imagers to be the master frame valid and master line valid signals for the rest of the camera electronics in that quadrant. Selecting a single master frame valid and line valid signal reduces the amount of logic that is passed around in the FPGA. The imager control module then sends the registered master frame valid, master line valid, and data bus to the VPTZ readout electronics. The registered data bus sent to the VPTZ readout electronics is 132 bits wide because it encompasses the twelve bit data output bus from all eleven imagers of the quadrant.

4.1.2 Logic Modules for Interfacing with RocketIO MGT

The RocketIO multi-gigabit transceivers operate in a different clock domain from the master system clock of the camera electronics (see Section 2.4). First-in-first-out (FIFO) modules with independent clock configurations are used to interface between the two clock domains. These FIFOs are implemented using the Xilinx Core Generator software from the Xilinx ISE development environment, and they are very robust in synchronizing signals from one clock domain to another clock domain. The FIFOs created by the Xilinx Core Generator are implemented using the embedded BRAMs in the Xilinx FPGAs.

As discussed in the previous chapters, each FPGA uses three RocketIO MGT cores - one MGT core to interface with an external source, and two MGT cores to interface with the FPGAs from the two adjacent quadrants. The RocketIO MGT cores are also instantiated using the Xilinx Core Generator software. The RocketIO MGT core used in the FPGA can translate the serial data into parallel buses for use in the FPGA, and also translate parallel buses into serial data for use with the MGTs (see Section 2.2.3 for details). A logic module in the camera electronics exists to multiplex and manage the data between the three MGTs used within the FPGA so that the data can flow through the system as specified by the readout architecture. This module determines how the quadrants are functionally connected and whether data should be passed to the neighboring quadrants or transferred out through the small form-factor pluggable (SFP).

4.1.3 PowerPC and Register File

One of the two embedded PowerPC is used to perform the higher level functions of the FPGA. The PowerPC processes the commands and message data from an external source. It sends the appropriate configuration data to the camera electronics in the FPGA based on the command and message data received. The PowerPC can also accept messages generated by the camera electronics itself for tasks such as housekeeping. A custom-made Register address file in the FPGA acts as the

interface between the rest of the firmware electronics of the FPGA and the PowerPC. The Register address file is attached to the PowerPC's on-chip peripheral bus and has access to the PowerPC's address spaces. Based on the command and message data received, the PowerPC configures an address of the Register file, which in turn activates the corresponding signal in the Register file with a designated value. The Register file then sends out the corresponding signal to the rest of the camera's firmware electronics.

4.2 VPTZ Parameters

The parameters needed to perform VPTZ are the starting point of the Region of Interest, the subsampling mode, and the number of valid pixel columns and rows that needs to be read out. The parameters will be calculated by software in the Display Level and then sent to the camera readout electronics. These parameters are what define the Region of Interest for the user.

4.2.1 Coordinate Spaces and Region of Interest Start Point

The starting point of a user's Region of Interest is determined based on the fact that the Micron CMOS Imager has a pixel array of 2,592 x 1,944 pixels. As a result of the pixel array, the Region of Interest's starting point is calculated on a coordinate system with each pixel equaling one unit of the coordinate, and the origin being at the absolute start of the imager's pixel array readout. Consequently, the end coordinate point for one imager is (2591, 1943), which is located at the end of the imager's pixel array because that is the 2,592nd pixel on the 1,944th line. This is known as the imager coordinate space and it is used for one imager. It is the coordinate system that the VPTZ readout architecture will reference when handling the readout and decimation of image data for each imager.

However, from a user's perspective at the Display Level, knowing the pixel coordinates of each individual imager is not very useful. Instead, the user will determine his or her starting point for the Region of Interest from a consolidated coordinate space

that encompasses the entire image area covered by all the imagers in the MASIV system. Therefore, one pixel is still equal to one unit of the coordinate system, but the entire coordinate space will be a result of the total dimensions that occur from combining the imagers of the four apertures of the MASIV system together. The complete dimensions of the consolidated coordinate space are illustrated in Figure 4-1 (not drawn to scale) assuming that there is no overlap between the imagers when the focal planes are combined ¹.

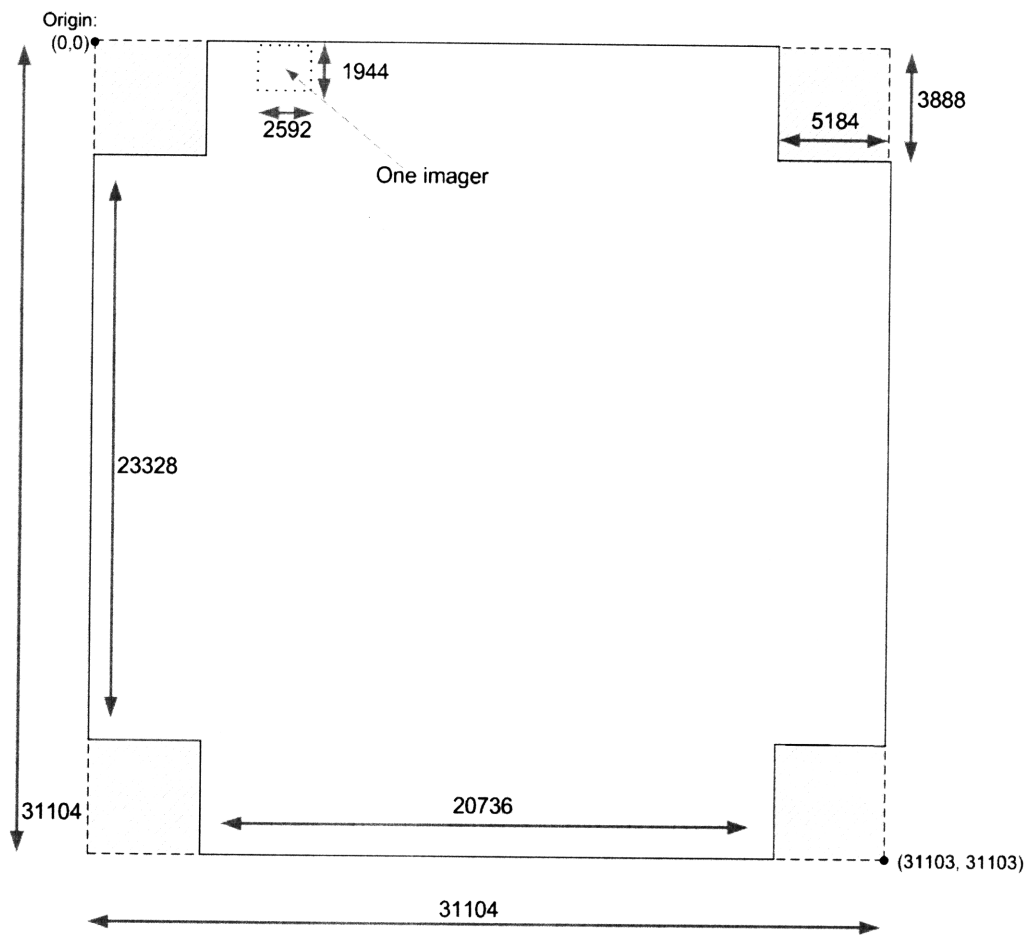


Figure 4-1: Pixel Dimensions of the Consolidated Coordinate Space of Entire Coverage Area.

Note that corners of the consolidated coordinate space are blank because the corners of the focal plane mosaic are not populated with CMOS imagers (Figure

¹In reality, there are overlaps between the imagers when the four apertures are stitched together, but this was not addressed in this thesis project

2-3). When the four focal planes of the MASIV system are combined to create the final overall image of the entire coverage area, it creates these blanks at the corners. However, in order to preserve a simple coordinate numbering system, the blank corners will still be indexed with coordinate units.

A software process in the Display Level maps the points in the consolidated coordinate space to the imager coordinate space of the corresponding imagers it belongs to. Once a user determines the starting point of his Region of Interest using the consolidated coordinate space, this software process will calculate the start point coordinate value into the imager coordinate space and send that value to the FPGA in control of the corresponding imager.

The camera electronics have no knowledge of the consolidated coordinate space used at the Display Level. The readout electronics only needs to receive a starting coordinates value defined in imager coordinate space for each of the eleven imagers under its control. If the Region of Interest in the consolidated coordinate space happens to span across multiple imagers, the software processes at the Display Level will determine a corresponding starting point for each of the spanned imagers that the Region of Interest encompasses and then send those values to the corresponding FPGAs. The implementation of the readout architecture in this chapter treats each of the imagers as separate entities, thus requiring a start coordinate value for each imager that has valid image data.

4.2.2 VPTZ Subsampling and Number of Valid Pixel Columns and Rows

In this implementation of the VPTZ architecture, a simple skipping process is used to subsample the image data. The skipping process can skip entire columns and/or rows of pixels for data decimation.

As mentioned in Section 3.1.1, the readout architecture must maintain the Bayer pattern readout employed in the CMOS imagers. The Bayer pattern filter arranges the color information in pairs in both the horizontal and vertical direction, and thus

Table 4.1: Skip Modes and # of Skipped Pixels

Skip Mode	Total Pixels/ Skip Interval	# of Skipped Pixels
2x	4	2
4x	8	6
8x	16	14
16x	32	30
32x	64	62

the skipping process must be performed in pairs. If the skip mode is 2x, then the readout architecture would read out two pixels, and then skip the next pair of pixels. A skip mode of 4x would read out two pixels, and then skip the next three pairs of pixels. Twice the value of the skip mode is equal to the total number of pixels that are in a skip interval. A skip interval is equal to the two valid pixels plus the number of skipped pixels. A user defines the skip mode desired for his Region of Interest at the Display Level, and then that value is sent as a parameter to the camera readout electronics. Table 4.1 shows a summary of the number of pixels involved for the skip mode values that are allowed in this implementation of the VPTZ readout architecture. The skip mode is limited to modes that can be divided and multiplied by a factor of 2 for easier implementation in the digital logic.

The pixel count of a Region of Interest for a user will be kept constant at 1 Megapixels, or 1024 pixels x 1024 pixels. Software processes at the Display Level can determine how many valid columns and valid rows of pixels would be needed per imager to fulfill the constant pixel count size of the Region of Interest. The number of valid columns and valid rows of pixels is calculated according to how the Region of Interest is subsampled and spanned across the imagers. For example, if a Region of Interest with no subsampling is spanned evenly across two imagers in the horizontal direction, then the number of valid pixel columns needed from each of the imagers would be 512 columns. This information is then sent down as a parameter to the corresponding FPGA. The number of valid pixel columns and rows required for every imager in the system is calculated and sent as a parameter to the camera electronics of the system.

In summary, the skip mode, the start coordinates for each imager, and the total number of valid pixel columns and rows needed for each imager are calculated by software at the Display Level and then sent as the VPTZ parameters to the camera electronics readout architecture.

4.2.3 Row Offsetting

Because all the imagers are operating simultaneously, the readout architecture has to somehow manage the pixel data bursting off of all eleven imagers of the quadrant at the same time. A restriction is placed on the system so that the readout architecture will have to manage only the burst pixel data from up to three imagers of the quadrant. The 1 Megapixel size limit of the Region of Interest means that a Region of Interest will span across multiple imagers within the same quadrant only when it is subsampled. By exploiting the fact that the Region of Interest is subsampled and that pixel data is read out row by row, row offsetting is employed to ensure that imagers along the vertical direction of the quadrant do not have the same valid rows for VPTZ readout.

Figure 4-2 gives an overview of row offsetting for one quadrant (not drawn to scale). In Figure 4-2, there is both vertical and horizontal subsampling, and the Region of Interest spans across multiple imagers in the quadrant in both the horizontal and vertical direction. There is a chance that the Region of Interest can have the same valid row for read out between the imagers placed along the vertical direction of the quadrant. In Figure 4-2, *Imager 3* and *Imager 4* have a valid row of pixels at *row n* and *Imager 6* and *Imager 7* also have a valid row of pixels at *row n*. Normally, the pixel data being burst out of those four imagers at *row n* would have to be managed by the camera electronics. However, if we offset the valid row that needs to be read out of *Imager 6* and *Imager 7* to instead be at *row n + 2*, then the readout architecture will only have to manage the data from *Imager 3* and *Imager 4* during *row n*'s read out.

Row offsetting is implemented by simply changing the start coordinate parameter to the imagers so that the imagers along the vertical direction of the quadrant do

not start reading out the same row of pixels. It is a tolerable restriction because no information is actually lost during the row offsetting. Information is not lost because the row offsetting exploits how the rows are skipped during vertical subsampling: there are always at least two rows of pixels that are immediately decimated after two valid rows are read out. The row offset restriction works in pairs of rows in order to maintain the Bayer pattern. If there were two imagers along the vertical direction of a quadrant that required the same pair of valid rows to be read out, the restriction would allow the original pair of valid rows to be read out from the first imager, but then offset the second imager's valid rows to occur during the decimation rows of the first imager. Thus, image information is not lost but slightly distorted by a couple pixels at most.

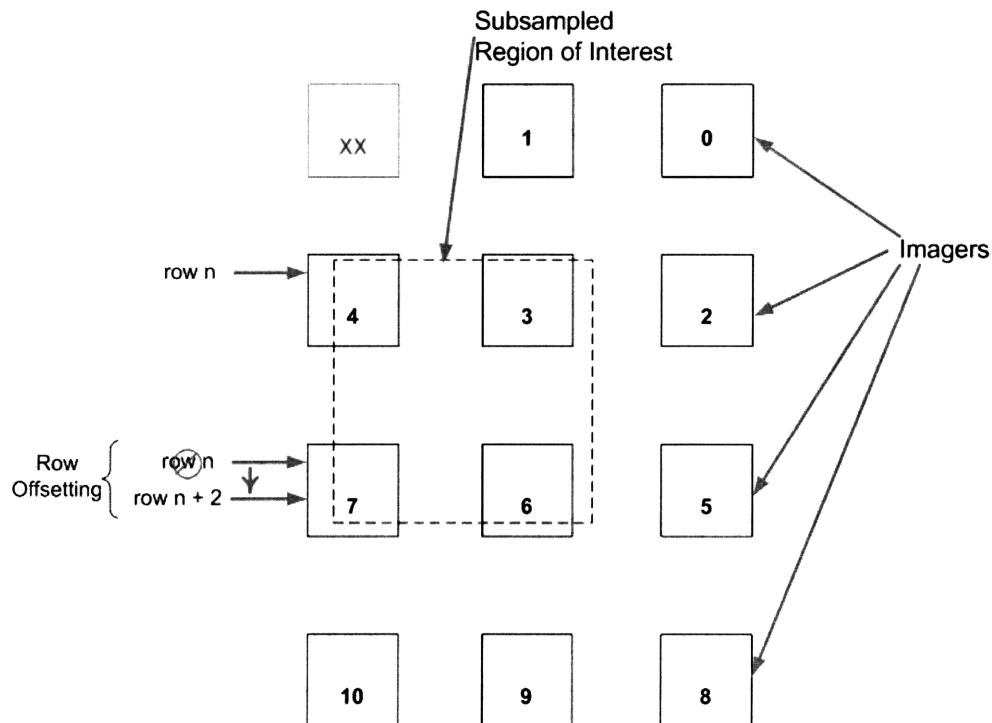


Figure 4-2: Row Offsetting During Vertical Subsampling in One Quadrant

4.3 Logic Implementation of the Readout Architecture

In the implementation that is discussed in this chapter, the camera electronics readout architecture is designed to perform the data decimation in the FPGAs and only output the pixels encompassed by the Region of Interest. The readout architecture must be able to manage the pixel data so that it meets the bandwidth requirements for the data bursting off the imagers and for the data rate limits at the RocketIO MGTs.

The symmetry of the quadrant layout of the hardware components on the aperture board enable the camera electronics' firmware to be significantly duplicated across the FPGAs. The readout architecture for the VPTZ application can be implemented for one quadrant, and then copied to the other quadrants of the system. Therefore in the subsequent sections we will examine in detail the firmware implementation of the readout architecture for one quadrant of the system.

Figure 4-3 shows a high-level block diagram of the readout architecture for one quadrant. The pixel data from the eleven imagers of the quadrant is read out at full resolution by the *Imager Control* module as described in Section 4.1.1. The *Imager Control* module then sends the pixel data and the master frame valid and line valid signals to the *Rectangles* modules. The *Rectangles* module handles the main data decimation of windowing and subsampling the pixel data as needed for VPTZ. As a result of the row offsetting described in Section 4.2.3, there should only be valid pixel data from up to three imagers being outputted by the *Rectangles* modules. The valid pixel data from up to three imagers is then sent to logic that interleaves the pixels from the different imagers into a single data stream; the data stream is divided into timeslots, and the logic places the pixel data from the different imagers into the appropriate timeslot of the data stream. As a result of the timeslot sorting, the arrangement of the pixel data on the data stream is slightly jumbled because it does not reflect the spatial placing of the imagers on the focal plane nor does it reflect the pixel array readout of the imagers. Therefore, the data stream is untangled when written into a Block Random Access Memory (BRAM) module in the *Buffer* module.

Each pixel from the jumbled data stream is written into an appropriate address of the BRAM so that when the data stream is read out by the *Read Buffer* module, the pixels on the data stream will be arranged correctly. The data stream is then sent to the *Quadrant Selector* module to prepare the data stream to be transferred out of the quadrant by the RocketIO MGTs. Because the quadrants of the system are all daisy chained together, there may be image data from another quadrant being passed through the current quadrant. The *Quadrant Selector* module handles this by alternating the readout of the pixel data from the current quadrant or from the external quadrant.

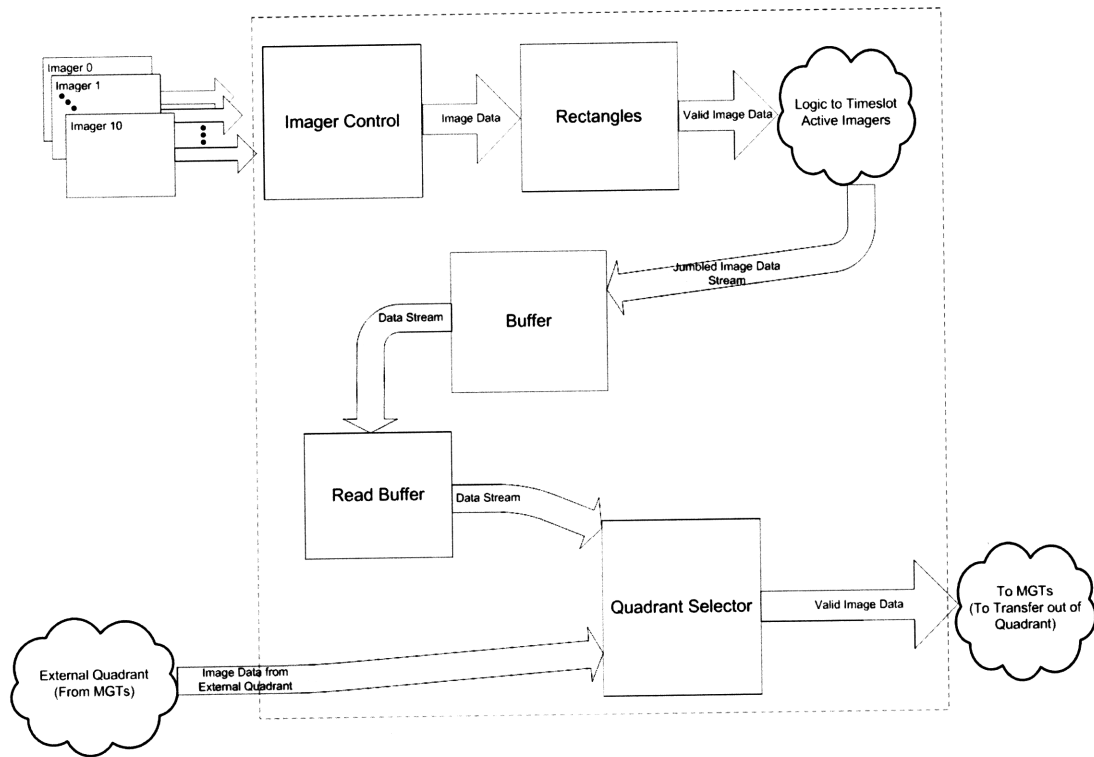


Figure 4-3: High-Level Block Diagram of Readout Architecture

A slightly more detailed block diagram for the part of the readout architecture that handles the pixel data bursting off the imagers is illustrated in Figure 4-4. The logic for this part of the architecture is encompassed in the VHDL top module labeled "Foveation Top Module." The corresponding source code, named *foveation_top.vhd*, and its underlying modules are included in Appendix A. The thick arrows in Figure

4-4 represent image data buses, and the thin arrows represent control and enable signals. The imager control module sends the image data from all eleven imagers of the quadrant plus the master frame valid and line valid signals to eleven *Rectangles* modules. There are actually eleven *Rectangles* modules instantiated because each module corresponds to the eleven imagers of the quadrant. Using the VPTZ parameters set by the user, the *Rectangles* modules decimates away the pixels that do not belong to the Region of Interest of the user. The decimated image data then goes through some logic hardware to prepare it to be written into a set of buffers. The valid pixels are buffered into BRAM memory in the *Buffer* module and then read out as specified by additional logic hardware.

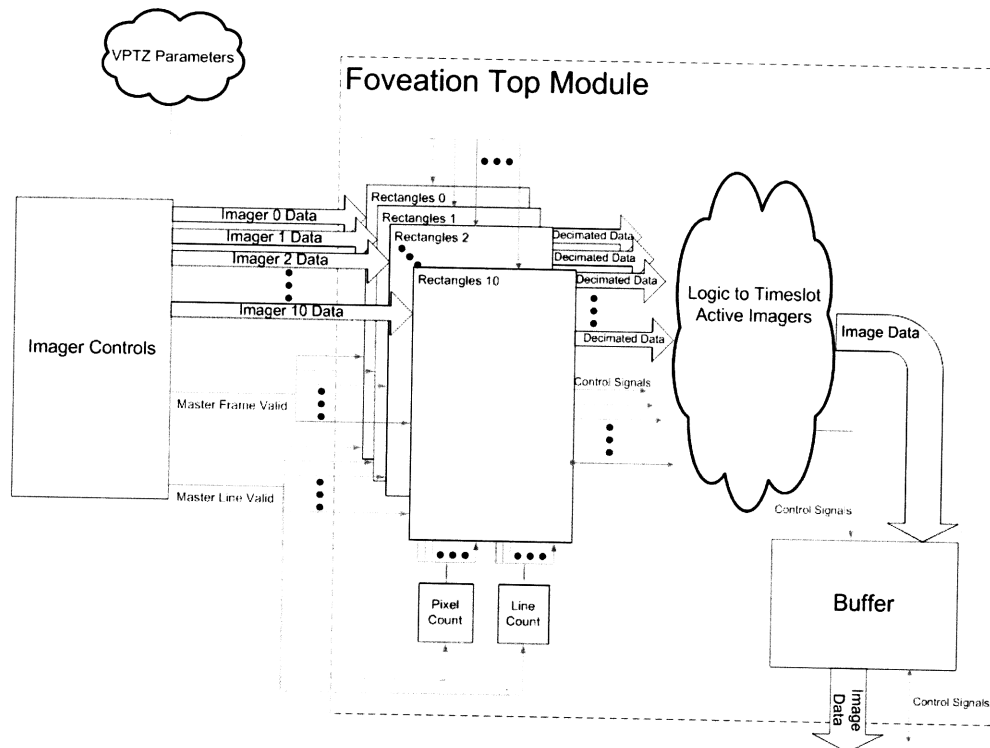


Figure 4-4: Block Diagram for the Part of the Architecture that Handles the Burst Pixel Data

4.3.1 Main Decimation Logic

The main data decimation needed for VPTZ functionality is performed in the *Rectangles* modules. The *Rectangles* module takes in data from the imager at full resolution

and then performs the windowing and subsampling decimations. It decimates the data by simply not asserting the output valid signal for the unnecessary pixels from the *Rectangles* module. The logic for the *Rectangles* module is fairly straightforward, and a block diagram is illustrated in Figure 4-5.

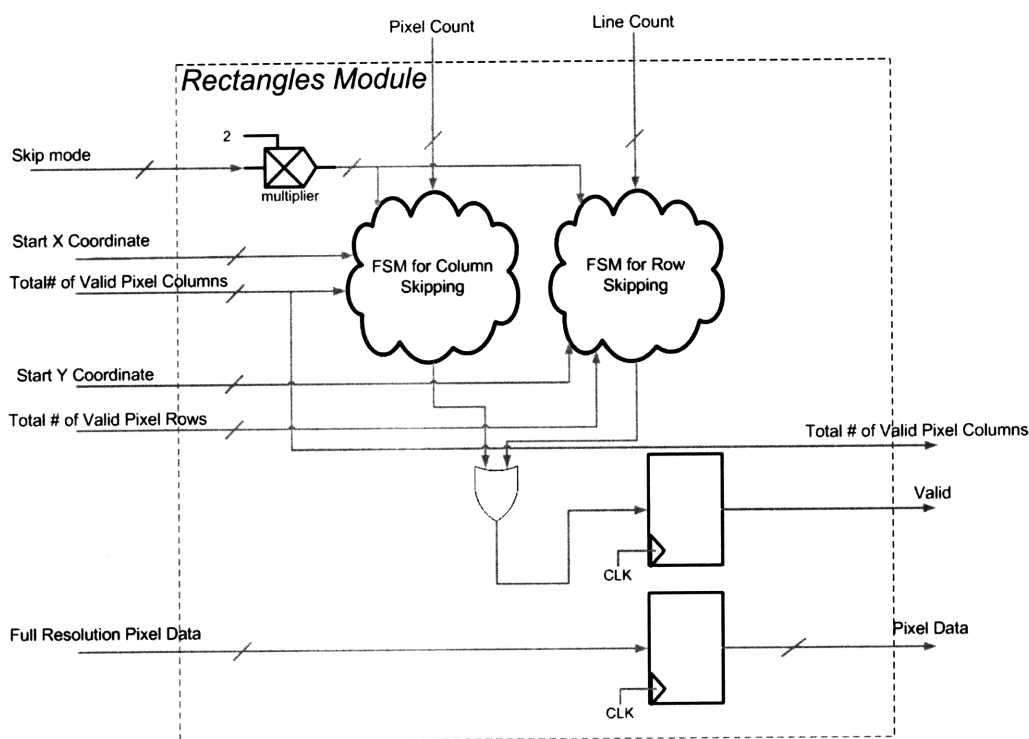


Figure 4-5: Block Diagram of *Rectangles* Module

The *Rectangles* module takes in as parameters the skip mode, the starting coordinates for the imager, and the total number of valid pixel columns and rows required for that imager. If the total number of valid pixel columns or rows is equal to zero, it means that there are no valid pixels required from that imager and the *Rectangles* module automatically decimates away the entire imager by not asserting the output valid signal for the entire imager. Otherwise, the *Rectangles* module uses two finite state machines (FSM) to enable or disable the readout of the pixel data based on the skip mode parameter. One FSM determines the readout based on column skipping and the other FSM determines the readout based on row skipping. Both FSMs are always running concurrently. The outputs of the FSMs are control signals to determine

the final output valid signal of the *Rectangles* module. The *Rectangles* module also takes in the pixel count and line count of the pixel array readout for the imagers. This is used as the reference for coordinate points in the image coordinate space: the pixel count corresponds to the columns of the pixel array, and the line count corresponds to the rows of the pixel array.

Using the skip mode parameter, the *Rectangles* module calculates the skip interval as described in Table 4.1. This information is used by the two FSMs to determine how many pixels need to be decimated for every set of valid pixels. There are two state machines to handle the pixel skipping because the pixel array of the CMOS imager is read out row by row. Row skipping determines whether or not the row is valid for read out. If a row is valid for read out, then column skipping is used to determine the valid pixels for that row. If a row is not valid for read out, then all the pixels in that row are decimated. Figure 4-6 shows the state transition diagram for the two FSMs. The two FSMs each have three analogous states: an initial wait state, a valid state, and a skip state. While in the initial wait state, the state machine waits for the pixel or line count to equal the start coordinate parameters. Once the start coordinate is reached, the state machine transitions into the valid state and enables the logic to read out the valid data. If there is no subsampling, the state machine will remain in this state until the number of valid pixels that is requested is read out. If there is subsampling, the state machine will remain in the valid state for two valid pixels in order to maintain the Bayer pattern, then transition to the skip state. It will remain in the skip state for the number of columns or rows it needs to skip before transitioning back to the valid state. It will continue to transition between these two states until the total number of valid pixel columns and rows requested by the parameters has been read out.

In summary, the *Rectangles* modules output the valid pixel data from the corresponding imagers with accompanying valid signals to some logic that will merge the valid pixel data into a single data stream. Also as shown in Figure 4-5, the total number of valid pixel columns which is accepted as an input by the *Rectangles* module is passed out as an output of the module. This information is later used to determine

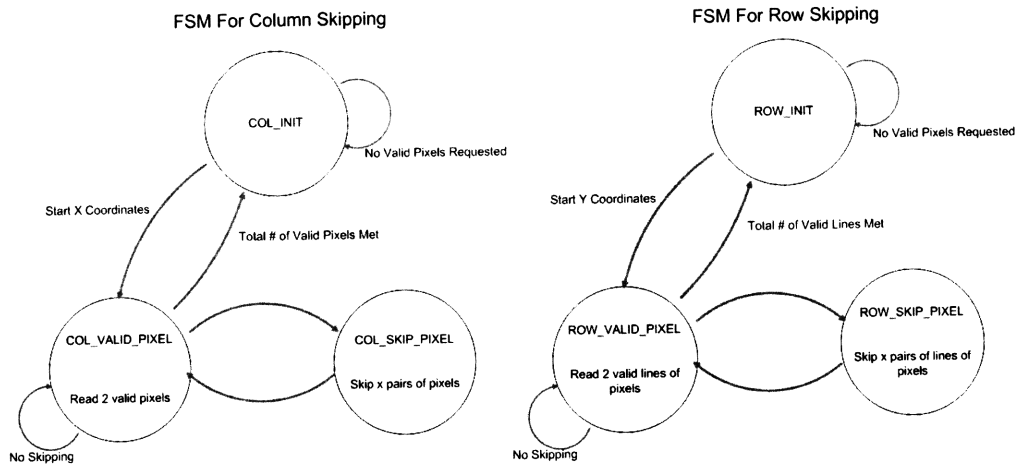


Figure 4-6: State Transition Diagram for Skipping Decimation

the addressing of the BRAM buffers.

4.3.2 Timeslot Sorting of the Pixel Data before Buffering

Logic exists to combine the valid pixel data from multiple *Rectangles* modules into a single data stream. The data stream is divided into timeslots, and pixel data from one *Rectangles* module occupies one timeslot of the data stream. As a result of the row offsetting described in Section 4.2.3, there can at most only be three *Rectangles* modules generating valid pixel data for a given row, and thus there are only three unique timeslots reserved for pixel data in the data stream. Figure 4-7 is a block diagram illustrating how the pixel data sent from the multiple *Rectangles* modules are multiplexed into one data stream before it is sent to the buffers. A single *Rectangles* module asserts a valid signal if there is valid pixel data on the current row, and it also outputs the total number of valid pixel columns expected for the present row of the module. Sequential logic uses the valid signals from the *Rectangles* modules to determine which of the eleven *Rectangles* modules actively have valid pixel data for the row. The sequential logic can find up to three active *Rectangles* modules that have valid pixel data for the row, and passes that information to a FSM. The FSM uses this information to output the number of valid pixel columns required for only the active *Rectangles* modules. The FSM also multiplexes the valid pixel data from

the active *Rectangles* modules into the appropriate timeslot of the data stream.

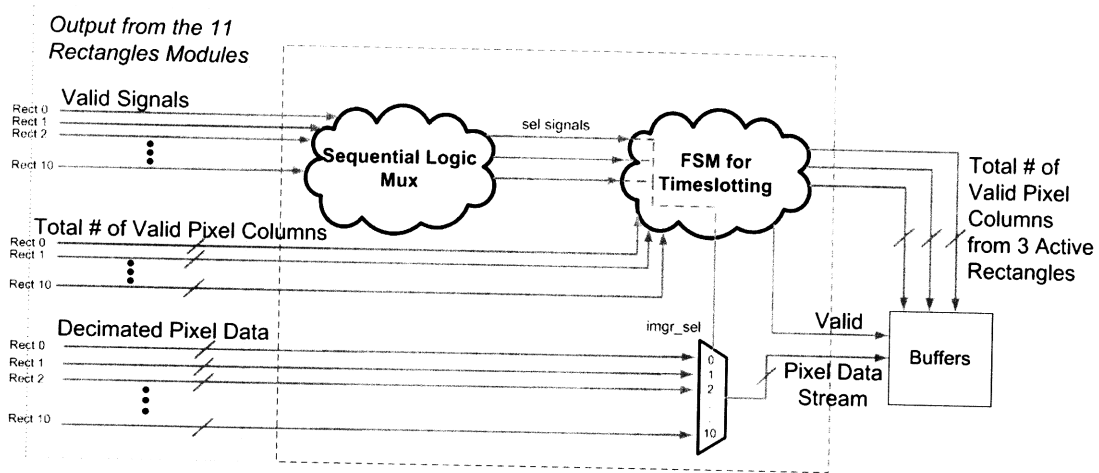


Figure 4-7: Block Diagram Showing How Single Pixel Data Stream is Created from Active *Rectangles*

The pixel data can be merged into a single data stream using timeslots because the master system clock runs at 100 MHz while the pixel clock runs at 25 MHz, and pixel data is read out from the imagers on the rising edge of the pixel clock. Therefore, pixel data remains valid for four system clock cycles because there are four master system clock cycles for every one period of a pixel clock. Hence, it is actually possible to sample a valid pixel from up to four different imagers during one pixel clock cycle, but our implementation only needs to sample up to three different imagers. Figure 4-8 illustrates a timing diagram showing how the pixel data from up to three imagers can be sampled and placed into the timeslots of one data stream.

The FSM controls a multiplexor to multiplex the valid pixel data from the corresponding *Rectangles* modules into the appropriate timeslot of the data stream. Figure 4-9 shows a state transition diagram for the FSM. There are four states in the FSM - a Start state and three analogous Find states - and each state represents a timeslot of the data stream. The FSM waits in the Start state during the blanking state of the imagers. Once the imagers enter its active state, the Start state occupies the first timeslot of the data stream and the FSM transitions into the first Find state. In the first Find state, the FSM uses the information sent from the sequential logic to find the first *Rectangles* module with valid data. If there is valid pixel data from

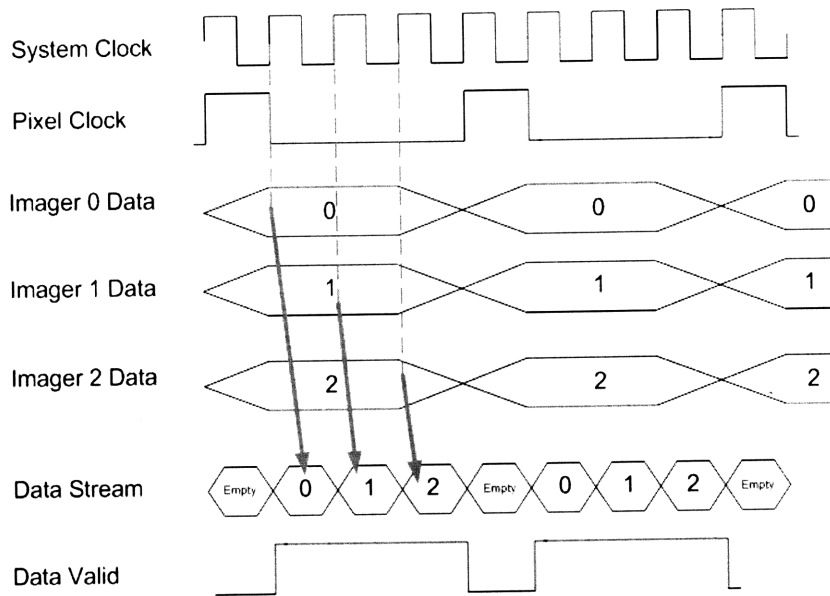


Figure 4-8: Timing Diagram for Sorting Pixel Data into Timeslots of a Single Data Stream

a *Rectangles* module during the Find state, the FSM will multiplex the pixel data onto the second timeslot of the data stream and output an accompanying valid signal. The FSM transitions into the next Find state at the rising edge of the master system clock. During the next Find state, the FSM checks to see if there is valid pixel data from the next *Rectangles* module. If there is valid data, then the FSM will multiplex the pixel data into the next timeslot of the data stream and output the accompanying valid signal. If there is no *Rectangles* module with valid pixel data at any of the Find states, then the FSM will not output a valid signal and there will be null data multiplexed into the corresponding timeslot of the data stream. Each Find state also outputs the number of valid pixel columns expected from the *Rectangles* module for the corresponding timeslot. The FSM continuously transitions through the four states at every system clock cycle until the current row is finished and the imager returns to its blanking state.

In summary, a single stream of pixel data is sent to the buffers. The FSM sends a valid signal to the buffer to indicate when the pixel data on the stream is valid. The FSM also sends to the buffer the number of valid pixel columns to expect from up to three *Rectangles* modules. This information is used to determine the addressing

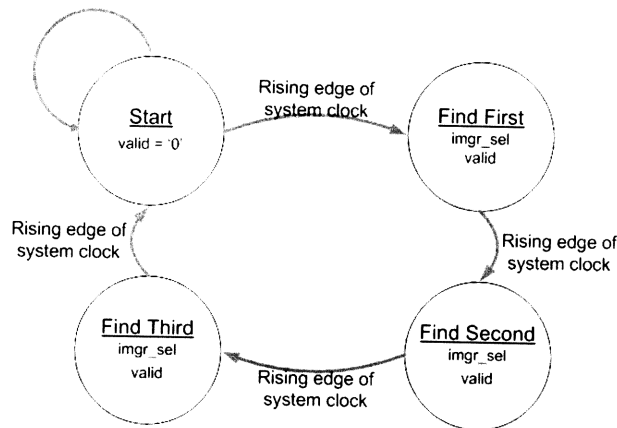


Figure 4-9: State Transition Diagram for Sorting Pixel Data into Timeslots of a Single Data Stream

scheme for the BRAM buffers.

4.3.3 Row by Row Buffering using BRAM

The readout architecture buffers pixel data from the quadrant on a row by row basis, and it uses two BRAMs to buffer the pixel data of the quadrant. The BRAMs are customized and instantiated in the firmware by using the Xilinx Core Generator development tool. The pixel data is written to and read out of the BRAMs by "ping ponging" between the two BRAMs: while a row's pixel data is being read out of one BRAM, the next row's pixel data is being written into the other BRAM. This throttles the data being generated so that no valid pixels are lost. The *Buffer* module also inserts control words into the BRAMs to classify the data written into the BRAMs. Figure 4-10 shows a block diagram of the *Buffer* module depicting the writing process to the BRAMs. The *Buffer* module takes in a pixel data stream, a valid pixel signal, and up to three *total number of valid pixels* signal. The valid pixel signal indicates when the pixel data stream has valid pixel data and it is used as the *Write Enable* signal to the buffers. The *total number of valid pixels* signal indicates the number of valid pixels that exist on the current row of the corresponding imager and it is used to determine the addressing scheme used to write to the BRAMs. The FSM determines what type of data word gets written to the BRAMs, and which address to write the

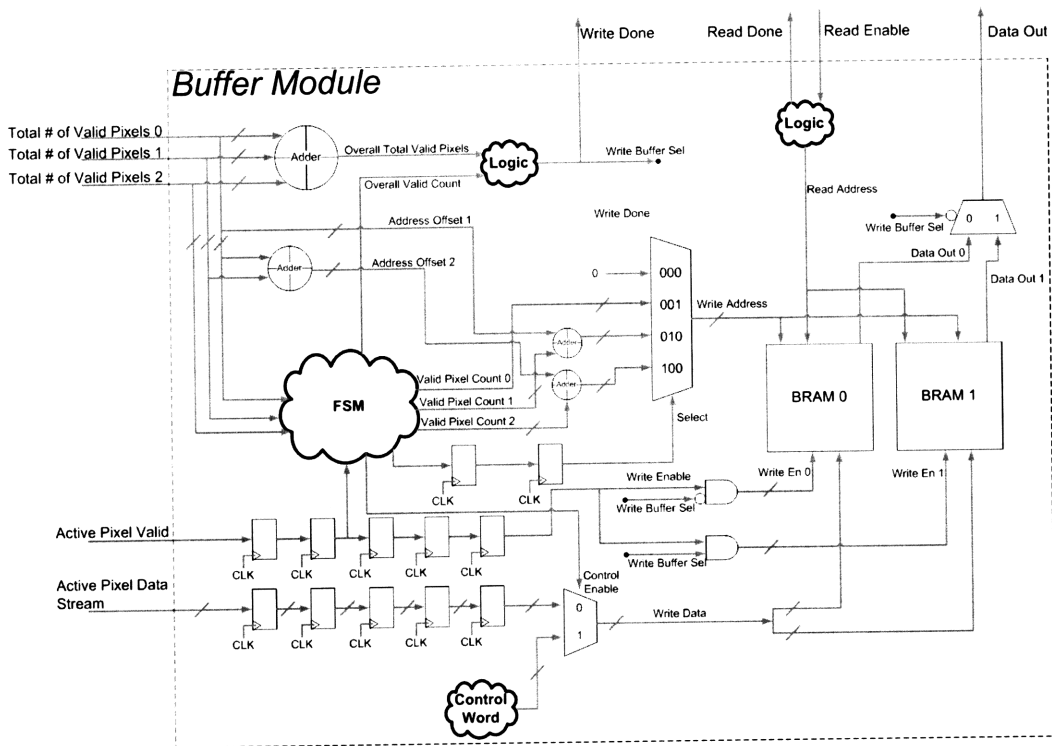


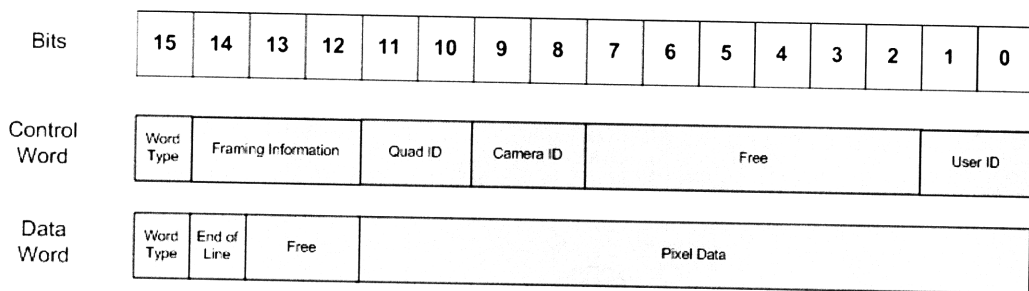
Figure 4-10: Buffer Module Block Diagram to Buffer Pixel Data into BRAMs

word into.

Pixel data is 12 bits wide but the words written into the BRAMs are 16 bits wide. The RocketIO MGTs use 16 bit words for its serial data transfers, and thus the pixel data written into the BRAMs are expanded to 16 bit data words. The control words that are written into the BRAMs are also 16 bits wide. Figure 4-11 summarizes the bit information for the two different types of words that can be written into the BRAMs. The system distinguishes between a control word and a data word by looking at the most significant bit (MSB) of the word. If a word's MSB is a '1,' then it is a control word, otherwise it is a data word. During the buffering of normal pixel data, the lower 12 bits of the data word are the pixel data and the upper four bits are kept to zero. However, on the last valid pixel, the 15th bit (bit 14) is asserted to indicate the end of the row.

A control word is written into the BRAM at the beginning of every new row of valid pixel data within the quadrant. The control word uses bits 14 down to 12 to

classify the row of data that will follow the control word. If it is the first row of the quadrant, then bits 14 down to 12 will be set to "010" to indicate the start of frame for the quadrant, otherwise it will be set to "000" to indicate a start of row for the quadrant. If bits 14 down to 12 are set to be "101" then it indicates that the words following the control word are some metadata header information, which will be discussed in detail in Section 4.3.4. The control word also indicates which quadrant and aperture the following row of pixel data belongs to.



Control Word Summary:

- bit 15: Word Type
 - "1" = Control Word
 - "0" = Data Word
- bits 14-12: Framing Information
 - "010" = Start of Frame
 - "000" = Start of Line
 - "101" = Metadata Header Start
- bits 11-10: Quad ID
- bits 9-8: Camera ID
- bits 7-2: Not used
- bits 1-0: User ID

Data Word Summary:

- bit 15: Word Type
 - "1" = Control Word
 - "0" = Data Word
- bit 14: End of Line
 - "1" = End of Line (last valid pixel)
 - "0" = Regular valid pixel
- bits 13-12: Not used
- bits 11-0: Pixel Data

Figure 4-11: Summary of the Bits in a Word

Valid pixel data from the quadrant is buffered into the BRAMs one row at a time. The limiting case that determines the size of the BRAMs is when a user's Region of Interest is entirely contained in one quadrant. In this case, there will be 1024 valid pixels for a row, and the BRAM will have to buffer all 1024 pixels. When the control words are included into the BRAMs, the BRAM will have to at least be 1025 words deep. Therefore, the BRAM that is instantiated in the *Buffer* module is 1025 words deep and 16 bits wide. The addresses of the BRAM correspond to the depth of the

BRAM, and thus there are 1025 address spaces in one instantiation of the BRAM: the first address location of the BRAM is 0x000 and the last address location of the BRAM is 0x400.

In the case when a Region of Interest spans across multiple imagers in the same quadrant, the pixel data is written into the BRAMs to reflect the spatial placement of the imagers in the quadrant. The ordering of the pixel data in the data stream coming into the *Buffer* module does not exactly reflect the spatial placement of the imagers in the quadrant because of the way the pixel data from the different imagers are merged onto the timeslots of the data stream as described in Section 4.3.2. The merging process handles the challenge of capturing valid pixel data that occur at the same time across multiple imagers, but as a result, the overall arrangement of the pixel data on the data stream do not accurately represent the spatial relationship of the imager layout on the quadrant. However, if the pixel data from the data stream can be written into the appropriate addresses of the BRAM, the pixel data can later be easily read out to reflect the spatial ordering of the imagers in the quadrant. To retain the spatial layout of the imagers onto the BRAMs, the pixel data from the same imager should be written next to each other in the BRAM's addresses; this requires the BRAM to be divided into sections that correspond to the active imagers. The BRAM can be divided into sections by offsetting the addresses so that the pixel data from the same imager can occupy a block of addresses of the BRAM. The *Buffer* module uses the *total number of valid pixels* signals to help determine the appropriate address offset for the start of the different sections in the BRAM. For example, if a Region of Interest happens to span across three imagers in the same quadrant, the BRAM will be divided into three sections corresponding to the three imagers. The start address for the first section of the BRAM corresponding to the first imager will not require an offset, but the start address for the second section of the BRAM corresponding to the second imager will require an offset equal to the total number of valid pixels from the first imager. Similarly, the start address for the third section of the BRAM corresponding to the third imager will require an offset equal to the combined total number of valid pixels from the first and second imagers. An FSM

then goes through the data stream and fills in the BRAM accordingly.

The state transition diagram for the FSM is shown in Figure 4-12. The states in this FSM correspond to the timeslots of the data stream. However, there is an extra state, labeled Write Header in Figure 4-12, which enables the control word to be written to the BRAM if it is a start of frame or start of row for the quadrant. The control word is always written into the first address space of the BRAM, which is Address 0x000. After the control word is written into the BRAM, the FSM transitions into the states that correspond to the timeslots of the data stream. Each timeslot of the data stream within a pixel clock cycle represents the pixel data from a different imager, and the FSM exploits this fact to write the pixel data to the appropriate section of the BRAM. These states output a count that keeps track of the number of valid pixels read out so far for the corresponding imager. The count is combined with the address offset information determined by the *total number of valid pixels* signal in order to determine the actual address of the BRAM that the pixel data should be written into.

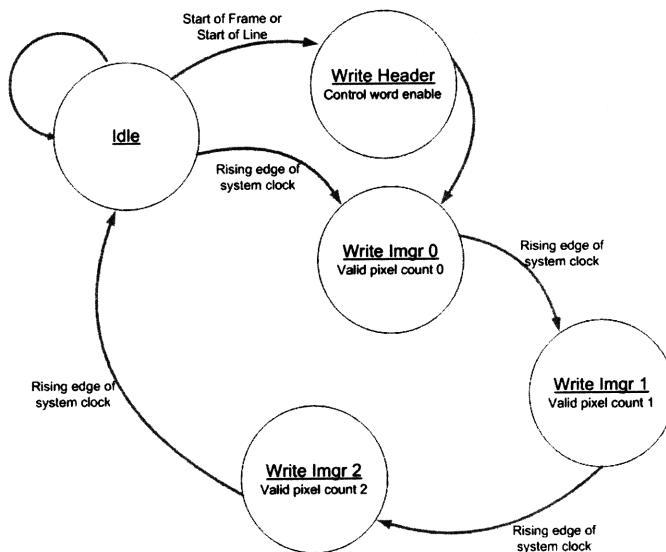


Figure 4-12: State Transition Diagram for FSM in the *Buffer* Module

After the row of valid pixel data for the quadrant is finished writing into one BRAM, the *Buffer* module then writes the next row of valid pixel data into the next BRAM. The *Buffer* module alternates the writing of the rows between the two

BRAMs. Consequently, the read out of the buffered pixel data alternates between the BRAMs that are not being actively written to.

4.3.4 Logic to Read Pixel Data from the Buffers

The *Read Buffer* module controls the readout of the pixel data from the BRAM buffers. Using a FSM, the *Read Buffer* module sends a *Read Enable* signal to the *Buffer* module to command the BRAM to start reading out the pixel data. Determining which address to read out from the BRAM is actually determined by some simple logic in the *Buffer* module as shown in Figure 4-10; the address that is read out of the BRAMs is in sequential order starting from Address 0x000. Reading out the BRAM in this order ensures that the pixel data represent the spatial layout of the imagers on the focal plane. The *Read Buffer* module also appends metadata header information to the first row of the frame in the quadrant. The metadata header information is generated from the embedded PowerPC and it conveys system and housekeeping information such as frame counts, board temperature, etc. The usage of the metadata header information is inherited from the original implementation of the MASIV system, and it is still used with this VPTZ readout architecture because of some interfacing requirements needed by the current data acquisition software and also because of its usefulness. There are thirty-two words of metadata header information that the *Read Buffer* module must append onto the first row of the frame. The *Read Buffer* module also appends a control word in front of the metadata header information in order to classify that the data being sent is the metadata header information (see Figure 4-11). While the metadata header information is being outputted, the pixel data being read out from the BRAMs are buffered into a FIFO. The FIFO is sixteen bits wide and thirty-two words deep because there is a potential of up to thirty-two pixel data words that need to be buffered while the thirty-two metadata words are read out. Because the metadata header information is only appended to the first row of every quadrant's frame, only the first row of data will need to be buffered into a FIFO. The rest of the rows are read out normally from the BRAMs. Figure 4-13 illustrates the block diagram for the *Read Buffer* module.

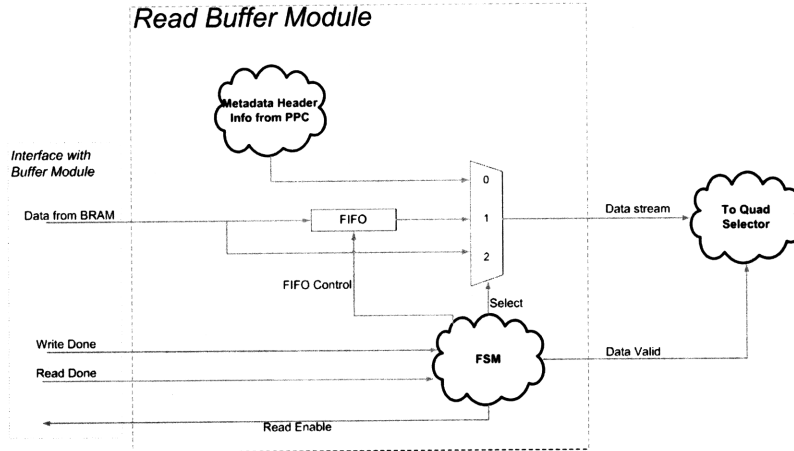


Figure 4-13: Block Diagram for the *Read Buffer* module

Using the *Write Done* signal sent from the *Buffer* module, the FSM generates the *Read Enable* signal to the BRAMs. The FSM also generates the control signals to the FIFO so that it can buffer the pixel data while multiplexing the metadata header information onto the output data stream. The state transition diagram for the FSM is shown in Figure 4-14.

There are two paths the FSM can take. The first path is when the *Read Buffer* module is reading from the BRAMs the first row of the frame and needs to append the metadata header information; the second path is for normal row readout from the BRAMs. The outputs of the FSM are the *Read Enable* signal to control the readout of the BRAMs, and a valid signal to indicate that the data stream being outputted by the *Read Buffer* module is valid. For the first path the FSM transitions into the *Frame Start Wait* state after receiving a start of frame signal. The FSM then waits in the *Frame Start Wait* state until the first row is finished writing into the BRAM. After the first row is finished being buffered into the BRAM, the FSM transitions into the Header state and starts outputting the metadata header information. It also begins the read out of the BRAM by asserting the *Read Enable* signal. While the metadata header is being outputted, the data from the BRAM are buffered into a FIFO. Once all thirty two metadata header words are transmitted, then the FSM transitions into the *Read FIFO* state to start reading out the buffered data from

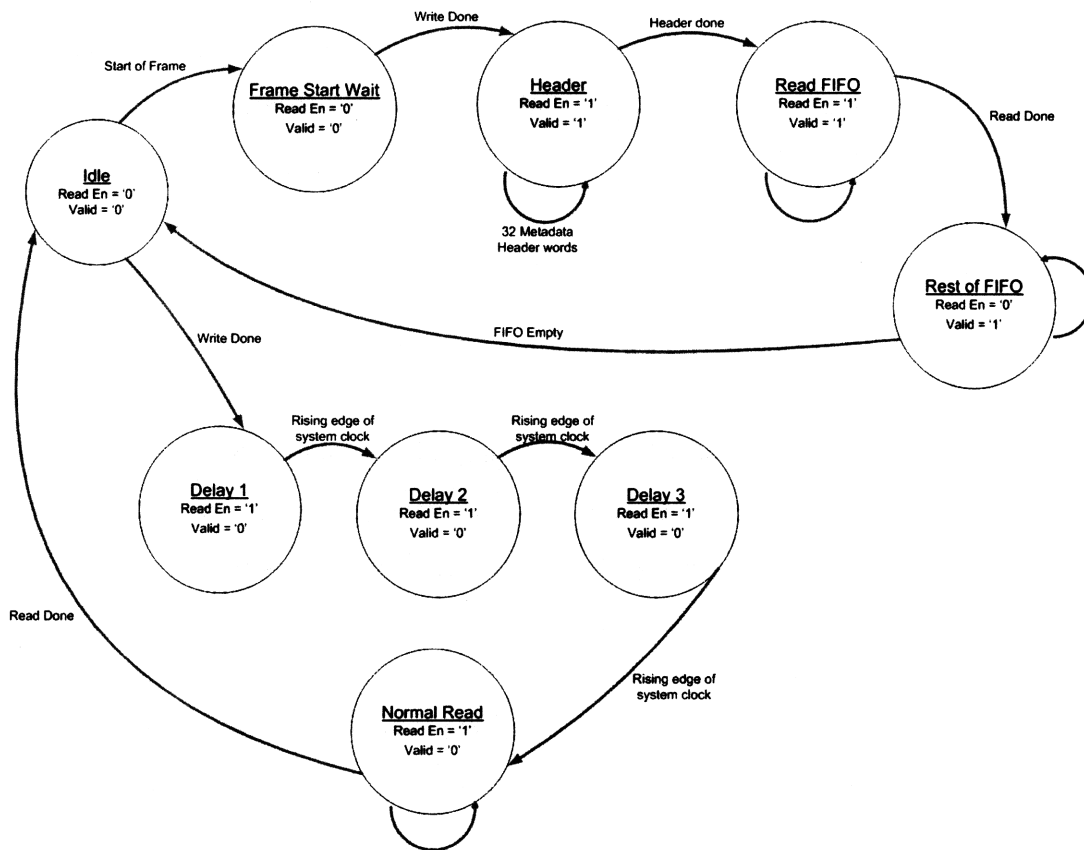


Figure 4-14: State Transition Diagram for the FSM in *Read Buffer Module*

the FIFO. After all the pixel data is finished reading out of the BRAM, the FSM transitions into the *Rest of FIFO* state to read out the rest of the data buffered in the FIFOs.

For all the other rows that are read out of the BRAMs, the FSM follows the second path. In the second path, the data is simply read directly from the BRAMs. There are three delay states because the BRAM instantiations themselves have inherent registers at the outputs that delay the readout of data by a few system clock cycles.

Finally, the *Read Buffer* module sends the pixel data from the BRAMs and the accompanying valid signal to some logic that will prepare the data to be transferred across the RocketIO MGTs.

4.3.5 Quadrant Selector Module

The *Quadrant Selector* module is the interface between the quadrant and the RocketIO MGTs. This module selects whether the data received from an external quadrant or the data generated from the current quadrant should be transmitted across the RocketIO MGT. This implementation of the VPTZ readout architecture daisy chains all the quadrants of the entire MASIV system together as shown in Figure 3-2. Therefore all the quadrants, with the exception of the first one, will have data from an external quadrant passing through it.

The *Quadrant Selector* module uses two FIFOs to buffer the image data: the *Int FIFO* is used to buffer the data generated by the current quadrant, and the *Ext FIFO* is used to buffer the data received from an external quadrant. Data received from an external quadrant are synchronized to the RocketIO clock domain, and so the FIFO used to buffer the data from the external quadrants synchronizes the data to the 100 MHz system clock. An FSM is used to decide which FIFO should be read out and passed to the RocketIO MGTs. Figure 4-15 illustrates the block diagram for the *Quadrant Selector* module.

The *Quadrant Selector* module always reads out the pixel data row by row. While the FSM is reading a quadrant's row of data from one of the FIFOs, pixel data from the other quadrant is being stored into the other FIFO. A state transition diagram for

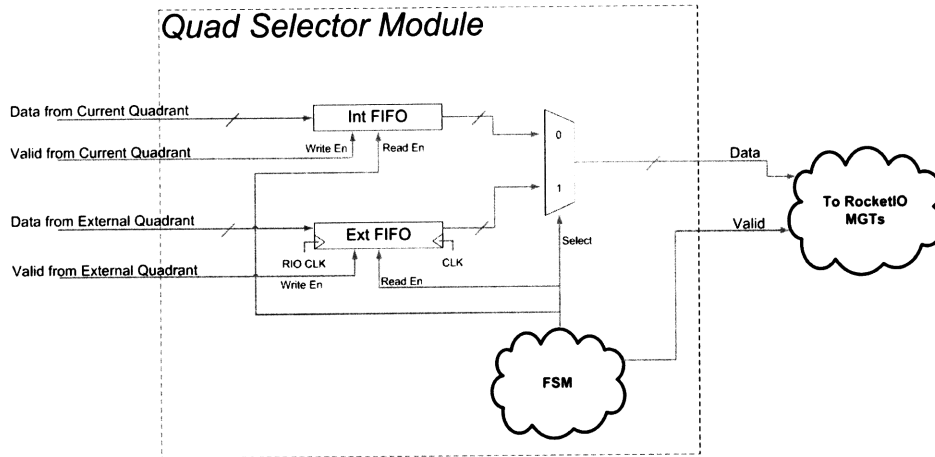


Figure 4-15: Block Diagram for the *Quadrant Selector* Module

the FSM is shown in Figure 4-16. The FSM initially waits in an Idle state until the FIFOs begin to fill up. When there is data detected in the FIFO, then the FSM will transition into a state that will start reading out the FIFO and pass the data to the RocketIO MGTs. If both FIFOs fill up at the same time, preference is given to passing out the data generated from the current quadrant, and the FSM will transition to the *Start Int FIFO* state. While in this state, the *Quadrant Selector* module reads out the data in the *Int FIFO* until the end of the row. Once the row is finished, the FSM transitions into the *Stop Int FIFO* state and checks to see if data is present in the *Ext FIFO*. If data is present in the *Ext FIFO*, then the FSM will transition to the *Start Ext FIFO* state and read out the data in the *Ext FIFO* until the end of the row. After it is finished reading the row from the *Ext FIFO*, it will check again to see if there is data present in the *Int FIFO*. If there is data present, it will transition back to the *Start Int FIFO*, and start the process over again. Otherwise, if there is no data present in the other FIFO, the FSM will transition back to the Idle state. In essence, if data is always present between the two FIFOs, the FSM will constantly alternate reading out the data between the two FIFOs.

The FIFOs used in the *Quadrant Selector* module are sixteen bits wide and 2048 words deep. Although a row of data from one quadrant will only be at most 1057 words deep (1024 pixels + 1 control word + 32 metadata header words), the FIFOs

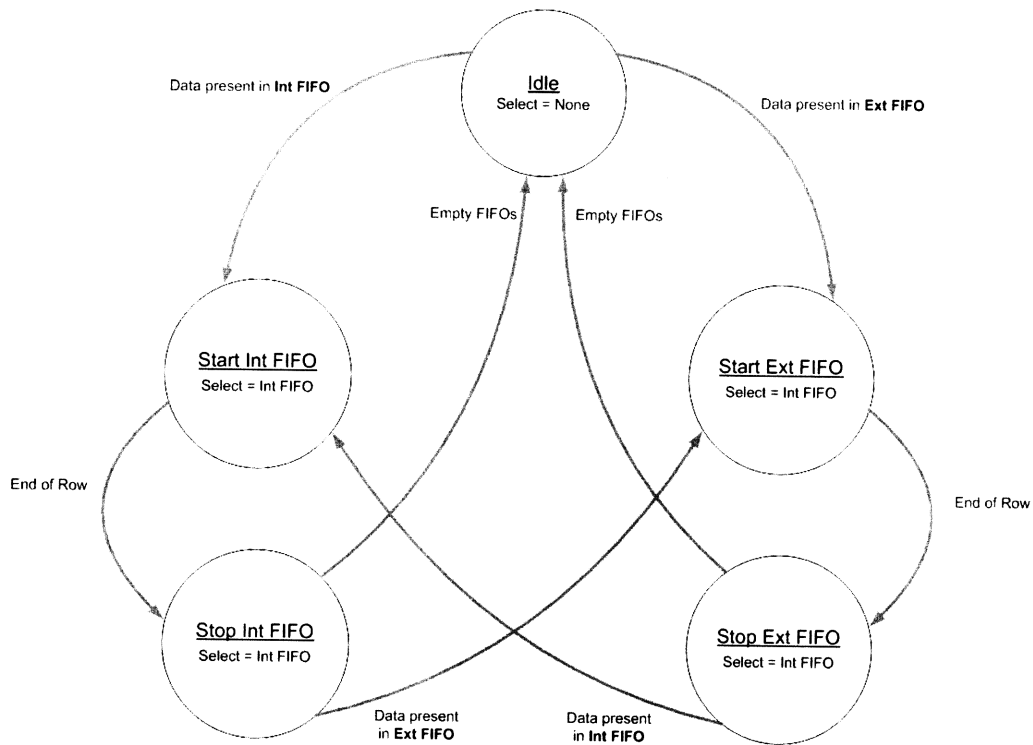


Figure 4-16: State Transition Diagram for FSM in *Quad Selector*

are a little bit bigger to absolutely ensure that no data is lost while it is being buffered.

The output of the *Quadrant Selector* module gets sent to the RocketIO MGTs, to transfer the row's data onto the next quadrant, or to the external devices if it is the last quadrant.

4.4 Testing and Debugging

At the time of this writing, a completely working display for the VPTZ application was still under development. The logic modules were tested by creating simulations of the operations of the modules using the ModelSim SE Plus 6.2f software tool. The modules were simulated both as independent entities and as well as connected entities. The test benches and other simulation codes used to simulate the modules are included in Appendix B. We will examine some of the simulations for the VHDL modules for the VPTZ readout architecture.

4.4.1 Simulating the Main Decimation Logic

Figure 4-17 shows the simulation results for the *Rectangles* module described in Section 4.3.1. The figure shows the simulation at two different time scales: the larger time scale, shown at the top of Figure 4-17, is used to show the general signal patterns created by the *Rectangles* module, and the smaller time scale, shown at the bottom of the figure, is used to show a more focused view of the simulation. Some of the parameters needed by the *Rectangles* module, such as the clock signal are generated by the test bench. Other parameters, such as the VPTZ parameters of the start coordinates are hard coded in the test bench.

A VHDL module is used simulate the behavior of the imagers. The imager test module simulates the outputs of the frame valid and line valid signals of the actual imagers, and it also simulates data that could be generated by the imagers. The pixel data output that is simulated by the imager test module is a test pattern that corresponds to the column number of the pixel array of the imager that is being read out, and it is represented by the *vid_reg* signal in Figure 4-17. Moreover, the first pixel that is outputted by the test pattern corresponds to the row number of the pixel array that is being read out. This test pattern is used because it simplifies viewing the operation of the skipping decimation process of the pixel data. The system clock is labeled *clock* and the pixel clock is labeled *imgr_sample_en*.

The simulation in Figure 4-17 shows a few select signals that demonstrate the operation of the *Rectangles* module. The name for the bottom two signals, labeled *pixel_valid* and *pixel_data* are highlighted in Figure 4-17 and they represent the outputs of the *Rectangles* module. The VPTZ parameters that were hardcoded for this simulation were the skip mode, the total valid pixel column and lines, and the start coordinates. The skip mode parameter was set to be 2x; the total valid pixel columns and lines were both set to be eight; the start coordinate was set to be (4,2).

The skipping happens in both the column and row directions of the pixel array; the results of the row skipping can be seen at the top of Figure 4-17, and the results of the column skipping can be seen at the bottom of the Figure. Because the skip mode

is 2x, two valid pixel columns or rows are outputted, and then two pixel columns or rows are skipped. The pixel column or row is skipped because the *pixel_valid* signal is not asserted during the skipped pixels.

The start coordinates are (4,2), so the *Rectangles* module does not begin outputting valid pixel data until the second row of pixels and the fourth pixel column for every row. Because the total number of valid pixel columns and lines are set to eight, the *Rectangles* module will only read out eight valid pixel columns for every row, and only eight valid rows.

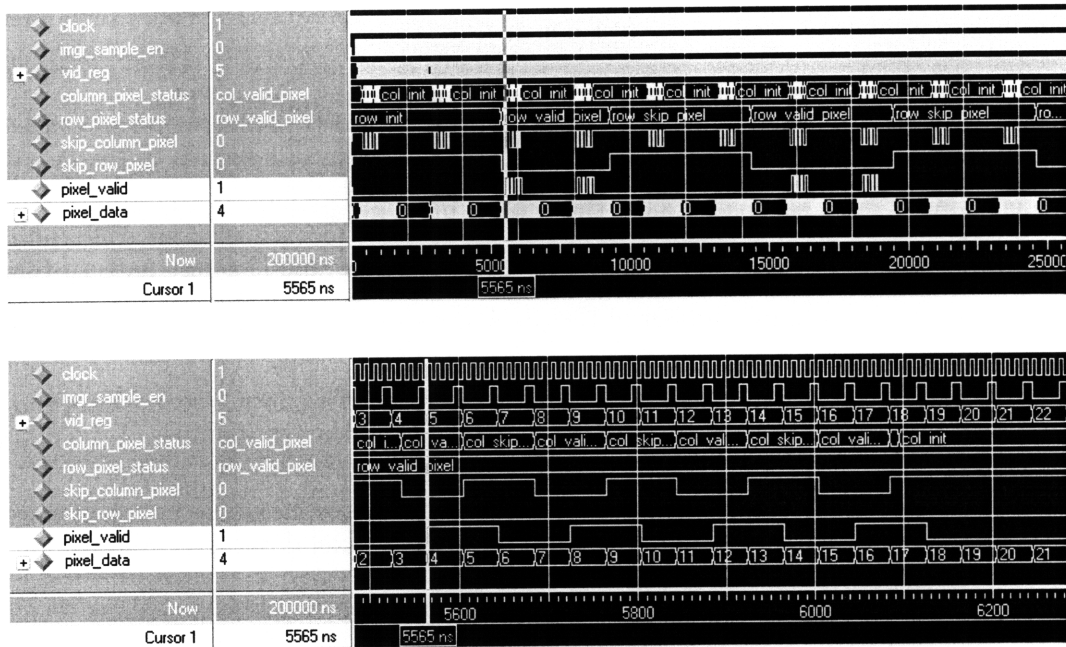


Figure 4-17: Simulation of the *Rectangles* Module

4.4.2 Simulating the Top Module

Figure 4-18 shows the simulation for the top module illustrated by Figure 4-4. The simulation for the top module includes the eleven *Rectangles* module instantiation, the logic used to timeslot the valid data into a single data stream, and the *Buffer* module. On the top of Figure 4-18 shows the simulation of the top module at the beginning of a valid row, and the bottom of the Figure shows the simulation at the

end of a row.

The hard coded parameters used in the simulation for the *Rectangles* module in Section 4.4.1 is used again in this simulation. Moreover, the parameters are sent to three *Rectangles* modules to simulate the limiting case of three imagers having valid pixel data at the same time. In order to simplify viewing the three different imagers in the simulation, the imager test pattern that is outputted by the imager test module is slightly different for the three simulated imagers. The first and third imager output the test pattern as described in Section 4.4.1, but the second imager always outputs a pixel value of 0xAAA.

The names of the important signals in Figure 4-18 are highlighted on the left of the simulation window. The timeslotted data stream and the accompanying valid signal is represented by the *active_pixel_data* and the *active_pixel_valid signal* respectively. The data words that are written into the buffer are represented by either the *data_in_0* or *data_in_1* signal. In the timing regions presented in Figure 4-18, the data word is represented by the *data_in_0* signal because the system is writing to BRAM 0 while reading from BRAM 1. The BRAM address in which the words are written to is represented by the *wr_addr* signal. The signal *wr_en_tmp(0)* represents the write enable signal to the BRAM indicating when the word should be written into the BRAM.

Notice that the first word written into the BRAM on top of Figure 4-18 is the control word with the value 0xA002. This means that the following row is the start of the frame for the quadrant, and that the data belongs to the user with an ID of 0x2. The control word is written into address location 0x000 (shown as a decimal number in the simulation window). The pixel data words are written into the corresponding address locations with the correct address offset determined by the three *total number of valid* signals (which are represented by *total_valid_pixels_imgr0/1/2_latched* signals in the simulation window).

On the bottom of Figure 4-18, notice that the last data word written is 0x4019. This means that this is the last valid pixel for the row because bit 14 of the data word is asserted.

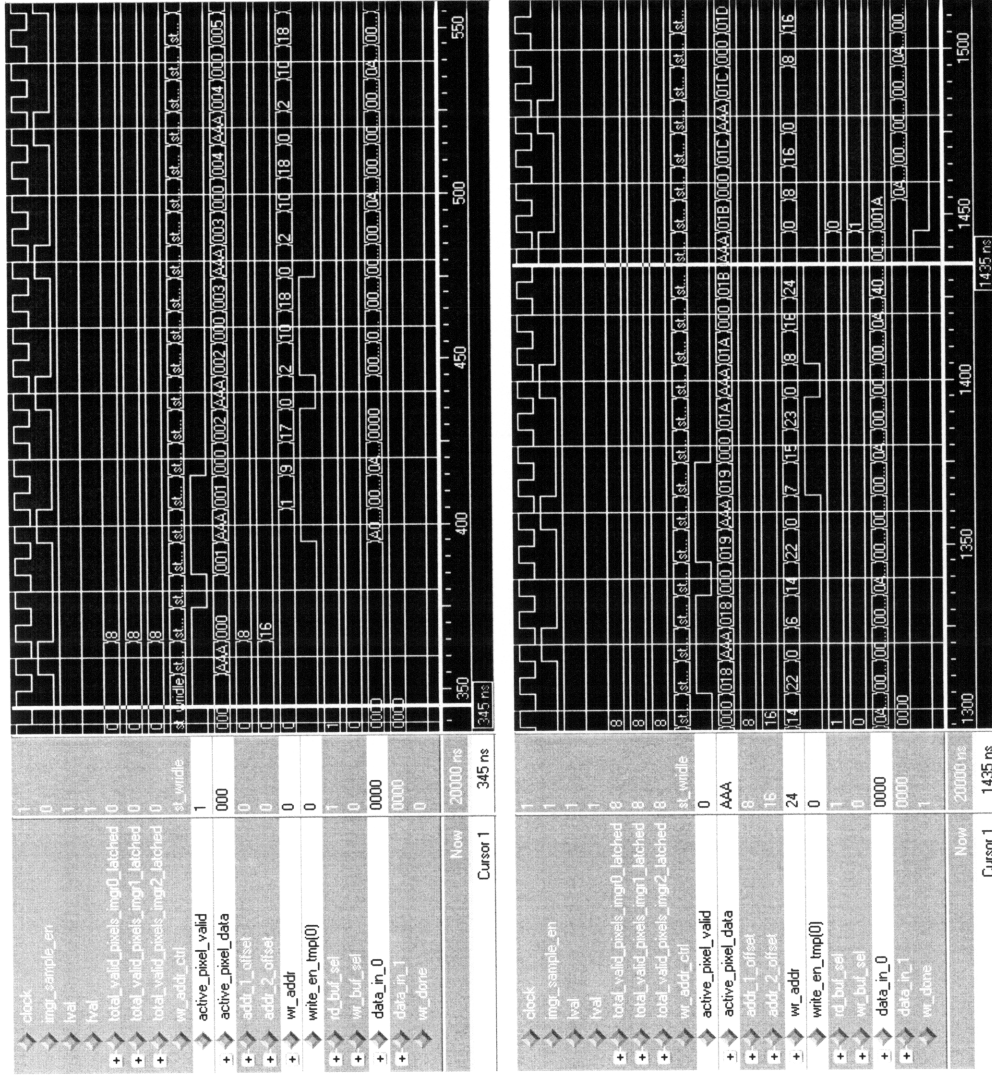


Figure 4-18: Simulation of the Top Module

4.4.3 Simulating the Reading of the Buffer

Figure 4-19 and Figure 4-20 shows the simulation for the *Read Buffer* module discussed in Section 4.3.4. This section simulates the operation of the *Read Buffer* module connected to the top module of Section 4.4.2. The top of Figure 4-19 shows the beginning stages of the *Read Buffer* module when it appends the 32 metadata header words to the output data stream. Random information is generated by the test bench and used as placeholders for the 32 metadata header words². The bottom of Figure 4-19 shows the timing region for when the *Read Buffer* module is done with appending the metadata header words and starts outputting pixel data. Figure 4-20 shows the timing region for the end of the row.

The names of the important signals are highlighted to the left of Figure 4-19 and 4-20. One of the outputs of the *Read Buffer* module is the *Read Enable* signal, represented by *readen* in the simulation windows. The *readen* signal enables the BRAMs from the *Buffer* module to start reading out. The other outputs of the *Read Buffer* module are the *riodata* and *riovalid* signal, which are the final data stream and the accompanying valid signal that will be transmitted across the RocketIO MGTs. The signal *data_out_0* represents the data readout from BRAM 0 of the *Buffer* module; in the timing regions examined in Figure 4-19 and Figure 4-20, the data is read from BRAM 0. The address of the BRAM that is being read out is represented by the *rd_addr* signal.

From the simulation window on top of Figure 4-19, notice that the first word on the *riodata* signal is a control word with the value 0xD000. This indicates that the following words on the *riodata* stream will be metadata header words. The pixel data words being read out from the BRAM is stored in the FIFO while the metadata header words are being outputted by the *Read Buffer* module. Notice that the BRAM address that is being read out is in sequential order starting from address location 0x000. The bottom of Figure 4-19 shows the region for when the *Read Buffer* module finishes outputting the thirty-two metadata header words data words and starts to read out the data words that were stored in the FIFO. Notice that *riodata* signal is

²Design and code by Tom Karolyshyn of Lincoln Laboratory

now the first word that is read out of the FIFO and it is the control word with the value 0xA000, which indicates that the pixel data to follow is the first row of the frame.

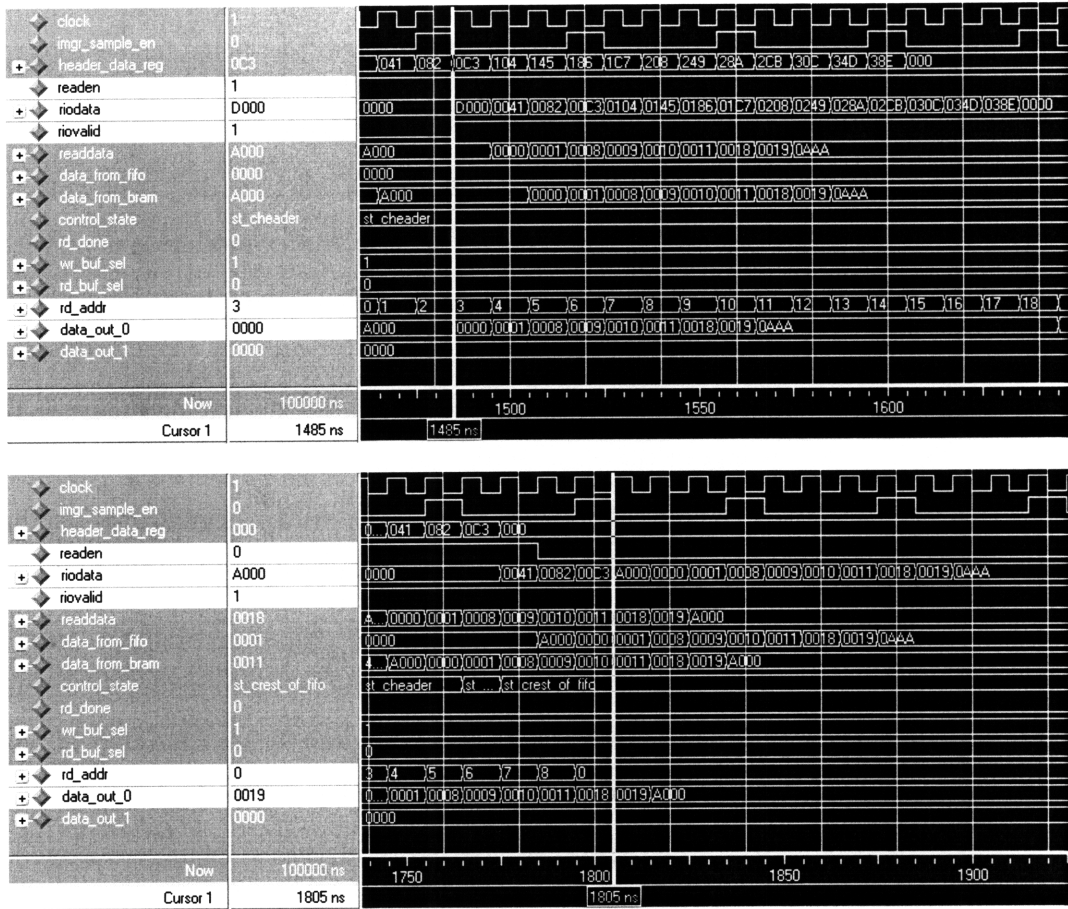


Figure 4-19: Simulation of the *Read Buffer* Module

Figure 4-20 shows the timing region for when the *riovalid* signal is the final pixel of the row, and it is the last valid data word to be outputted by the *Read Buffer* module.

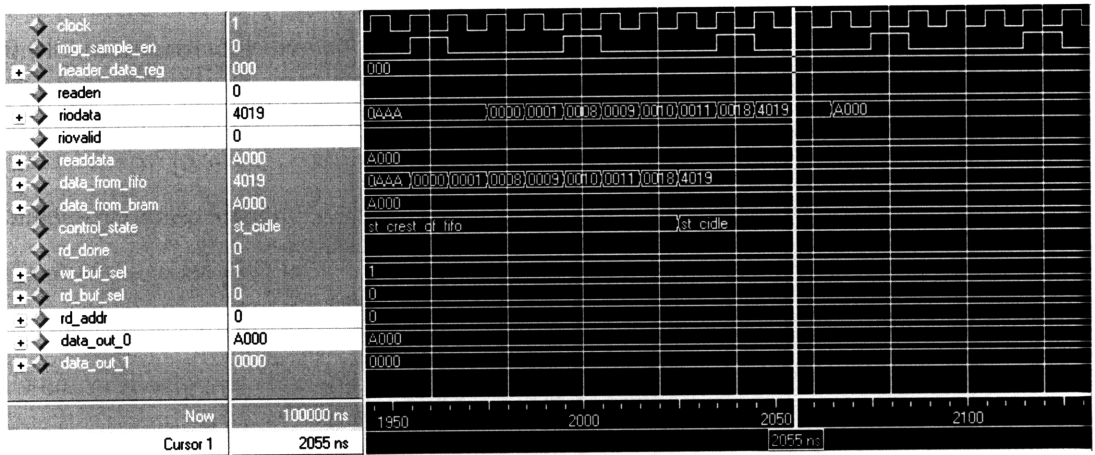


Figure 4-20: Simulation of the *Read Buffer* Module Showing the Last Pixel of the Row

Chapter 5

Conclusion

5.1 Future Work

The implementation of the readout architecture for data decimation is only a part of what is required to have a completely working VPTZ system. In this project we demonstrated the electronics portion of the VPTZ functionality: the ability to generate and decimate the data for a user's Region of Interest in the camera electronics. The decimated results were shown on a frame grabber data acquisition device to establish that the proper raw data was being produced by the cameras for the end user. Future work would expand the ability to view the image data from the MASIV cameras more suitably. First of all, the user interfaces for the display for the VPTZ application would need to be created. Also, work in image processing techniques such as image stitching, registration and demosaicing is actively being done for the current MASIV implementation, and this would need to be extended to include image data from the VPTZ read out as well.

As for the firmware implementation of the readout architecture, future work could be done to further improve the performance metrics of the camera system. For example, the pixel clock to the imagers can be increased in a future implementation of the camera electronics. The pixel clock to the imagers used in this project is 25 MHz, but the imagers can receive up to a 96 MHz pixel clock rate. Increasing the pixel clock to this frequency would change the implementation of the readout architecture,

but would enable faster frame rates. Furthermore, a new generation of the MASIV hardware is currently under development, which would create opportunities for other developments as well.

5.2 Summary

In this project, we examine and develop a video camera readout architecture to support the concepts of Virtual Pan-Tilt-Zoom for a high-pixel-count video system with a wide field-of-view. The VPTZ application allows a user to virtually pan, tilt, and zoom around a wide coverage area without having to physically move the camera. Technological advancements of the CMOS imager enable the 880 Megapixel imagery of the MASIV camera system, and the availability of large pixel counts allows for the development of novel, alternative uses such as VPTZ for the video camera system. The primary use of the MASIV system as an airborne surveillance sensor requires the readout and storage of a tremendous amount of raw data from the cameras. However, the sheer amount of pixel data that must be read out from the camera electronics imposes a limit on frame rates and other performance metrics. The VPTZ application provides a different use of the MASIV system by creating an alternative way to manage the read out of pixel data from the cameras. In VPTZ, the amount of data needed to be read out of the camera electronics is greatly reduced to reflect only a portion of the coverage area requested by an individual user. Most of the pixel data is decimated by the camera readout architecture, and only the fraction of pixels that encompass the user's Region of Interest needs to be read out by the camera electronics. As a result of the data decimation, certain performance metrics can be enhanced when the camera system employs the VPTZ functionality. Most notably, the frame rate of the system can be increased, and the video system can be used to support multiple users.

The user's Region of Interest is limited in this project to be 1 Megapixel in size because most modern displays and monitors only have support for images a little over 1 Megapixel in size. Therefore, the amount of data that is needed from the cameras is

significantly reduced from 880 Megapixels to 1 Megapixel per frame. In this thesis we discussed three approaches to decimating the extraneous data: decimation in software at the display level, decimation by the CMOS imagers, or decimation by the camera electronics via an FPGA. Each approach has its tradeoffs with regards to complexity and performance. In the readout architecture implemented in this project, the data decimation required for VPTZ functionality is performed in the camera electronics by the FPGAs. This approach provides a balance between complexity and performance - it enables faster frame rates than the software level decimation, and it is not as complex as developing the data decimation process by the imagers.

VPTZ functionality enables a new surveillance technique involving a high-performance video imaging system. Moreover, the readout architecture used to support VPTZ functionality adds new capabilities to the existing video system as demonstrated in this this project.

Bibliography

- [1] Mark Beattie, Larry Candell, Pablo Hopman, and William Ross. Multi-Aperture Sparse Imager Video System. February 2007.
- [2] BECTA. Recent trends in digital imaging. *Technical report, British Educational Communications and Technology Agency*, March 2004.
- [3] Abbas El Gamal. Trends in CMOS image sensor technology and design. *IEDM 2002*, December 2002.
- [4] James Glettler. Multi-Aperture Sparse Imager Video System: A Gigapixel Video Sensing System for Persistent Surveillance. Briefing Given at MIT Lincoln Laboratory, May 2007.
- [5] James Janesick. Dueling detectors: CMOS or CCD? The choice depends on the application. *OE Magazine*, pages 30–33, February 2002.
- [6] Canon Professional Network. Capturing the image: CCD and CMOS sensors, 2008.
- [7] Matt Renzi. Active pixel sensor. Presentation Given at MIT Lincoln Laboratory, April 2006.
- [8] Micron Technology. MT9P001 1/2.5-inch 5-Mp digital image sensor features data sheet, 2005.
- [9] Xilinx. RocketIO Transceiver User Guide, February 2007. ug024 (v3.0).
- [10] Xilinx. Virtex-II Pro and Virtex-II Pro X platform FPGAs: Complete data sheet, November 2007. DS083.

Appendix A

VHDL Source Code

A.1 foveation_top.vhd

The following VHDL code is the implementation of the top module for the VPTZ readout architecture as well as the logic described in Section 4.3.2 (to sort the pixel data into timeslots of a single data stream).

```
-- This is the top module that handles the burst pixel data from the
-- eleven imagers. It corresponds to the "Foveation Top Module" in the thesis
-- and it encompasses the eleven "Rectangles" module, the logic to timeslot the pixel
-- data into one pixel stream, and the "Buffer" module.

-- This module is instantiated in the higher level MASIV system top module, which
-- is not included in this appendix. Various comments of code that are not
-- related to the VPTZ application may have been commented out or not included
-- in this appendix.

-- This module receives the pixel data signals, and frame valid and line valid
-- signals from the imager control module.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- custom package with some helper functions
use work.sweet_package.all;
-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity foveation_top is
generic(
    QUAD_ID : in std_logic_vector(1 downto 0) := "00";
    CAMERA_ID : in std_logic_vector(1 downto 0) := "00";
    USER_ID : in std_logic_vector(1 downto 0) := "00"
);
Port (
```

```

Reset          : in std_logic; -- async reset
Clock          : in std_logic; -- system clock

-- from imgr_ctrl
imgr_sample_en : in std_logic; -- sample incoming data and framing signals from imager
fval_master    : in std_logic;
lval_master    : in std_logic;
fval_rise     : in std_logic;
fval_fall     : in std_logic;
lval_rise     : in std_logic;
lval_fall     : in std_logic;
vid_all       : in std_logic_vector(131 downto 0);

-- user defined params for size and location of region of interest (ROI)
start_x_all   : in std_logic_vector (131 downto 0);
start_y_all   : in std_logic_vector (131 downto 0);

total_valid_pixels_all : in std_logic_vector(120 downto 0);
total_valid_lines_all  : in std_logic_vector(120 downto 0);

-- user defined params for digital zoom (skip mode)
column_skip_mode : in std_logic_vector (7 downto 0);
row_skip_mode    : in std_logic_vector (7 downto 0);

ReadEn          : in std_logic;
wr_done_pulse   : out std_logic;
rd_done         : out std_logic;
first_imgr_sel  : out std_logic_vector(3 downto 0);
first_line_active_start : out std_logic;
ReadData        : out std_logic_vector(15 downto 0)
);
end foveation_top;

architecture Behavioral of foveation_top is

    signal imgr_sel, imgr_sel_reg      : std_logic_vector(3 downto 0);
    signal mux_sel0, mux_sel1, mux_sel2 : std_logic_vector(3 downto 0);
    signal mux_sel0_reg, mux_sel1_reg, mux_sel2_reg1, mux_sel2_reg2 : std_logic_vector(3 downto 0);

    signal fval_master_reg, fval_master_reg2, lval_master_reg, lval_master_reg2, lval_master_reg3 :
        std_logic;
    signal fval_master_int, lval_master_int : std_logic;

    signal lval_rise_int, lval_rise_reg_2_buf : std_logic;
    signal lval_fall_int, lval_fall_reg_2_buf : std_logic;

    signal pixel_count, next_pixel_count : std_logic_vector(11 downto 0);
    signal line_count, next_line_count   : std_logic_vector(11 downto 0);

    signal frame_count : std_logic_vector(11 downto 0);

    signal buffers_empty : std_logic;

    type pixel_coords_array is array(10 downto 0) of std_logic_vector(11 downto 0);
    signal start_x_array, start_y_array : pixel_coords_array;

    type valid_signals_array is array(10 downto 0) of std_logic;
    signal line_start_array, line_start_array_reg, frame_start_array, fval_rect_array,
        lval_rect_array : valid_signals_array;

```



```

signal pixel_valid_rect_array , pixel_valid_rect_array_reg1 , pixel_valid_rect_array_reg2 :
    std_logic_vector(10 downto 0);

type data_array is array(10 downto 0) of std_logic_vector(11 downto 0);
signal vid_array , vid_reg , vid_reg_int : data_array;
signal pixel_data_rect_array , pixel_data_rect_array_reg1 , pixel_data_rect_array_reg2 :
    data_array;

signal line_active , frame_active_start : std_logic_vector(10 downto 0);

signal first_line_active_start_int , first_line_active_start_reg1 , first_line_active_start_reg2 ,
    first_line_active_start_reg3 : std_logic;
signal first_line : std_logic_vector(10 downto 0);
signal first_line_2_buf : std_logic;

signal active_pixel_valid , active_pixel_valid_reg : std_logic;
signal active_pixel_data , active_pixel_data_reg : std_logic_vector(11 downto 0);
signal imgr_span , imgr_span_reg : std_logic;

type total_valid_pixels_array_type is array(10 downto 0) of std_logic_vector(10 downto 0);
signal total_valid_pixels_array : total_valid_pixels_array_type;

type total_valid_lines_array_type is array(10 downto 0) of std_logic_vector(10 downto 0);
signal total_valid_lines_array : total_valid_lines_array_type;

type find_valid_imgr_status_type is (st_findFirst , st_findSecond , st_findThird , st_findDone);
signal find_valid_imgr_status : find_valid_imgr_status_type;

signal total_valid_pixels_imgr0 , total_valid_pixels_imgr1 , total_valid_pixels_imgr2 :
    std_logic_vector(10 downto 0);
signal total_valid_pixels_imgr1_reg1 , total_valid_pixels_imgr1_reg2 : std_logic_vector(10
    downto 0);

signal imgr_sample_en1 , imgr_sample_en2 , imgr_sample_en3 : std_logic;

signal header_buf_en : std_logic;

begin

-- vptz_params_i : entity work.vptz_parameters_arranger
-- Port map (
--     Reset => Reset ,
--     Clock => Clock ,
--
--     OneImgrWindowStartX => OneImgrWindowStartX , --: in std_logic_vector( 15 downto 0 );
--     OneImgrWindowStartY => OneImgrWindowStartY , --: in std_logic_vector( 15 downto 0 );
--
--     OneImgrValidPixels => OneImgrValidPixels , --: in std_logic_vector( 14 downto 0 );
--     OneImgrValidLines => OneImgrValidLines , --: in std_logic_vector( 14 downto 0 );
--
--     start_x_all => start_x_all , --: out std_logic_vector (131 downto 0);
--     start_y_all => start_y_all , --: out std_logic_vector (131 downto 0);
--
--     total_valid_pixels_all => total_valid_pixels_all , --: out std_logic_vector(120 downto 0);
--     total_valid_lines_all => total_valid_lines_all , --: out std_logic_vector(120 downto 0)
-- );

-- initialize arrays
start_x_array <= (start_x_all(131 downto 120) ,
    start_x_all(119 downto 108) ,

```

```

        start_x_all(107 downto 96),
        start_x_all(95  downto 84),
        start_x_all(83  downto 72),
        start_x_all(71  downto 60),
        start_x_all(59  downto 48),
        start_x_all(47  downto 36),
        start_x_all(35  downto 24),
        start_x_all(23  downto 12),
        start_x_all(11  downto  0));

start_y_array <= (start_y_all(131 downto 120),
                 start_y_all(119 downto 108),
                 start_y_all(107 downto 96),
                 start_y_all(95  downto 84),
                 start_y_all(83  downto 72),
                 start_y_all(71  downto 60),
                 start_y_all(59  downto 48),
                 start_y_all(47  downto 36),
                 start_y_all(35  downto 24),
                 start_y_all(23  downto 12),
                 start_y_all(11  downto  0));

total_valid_pixels_array <= (total_valid_pixels_all(120 downto 110),
                             total_valid_pixels_all(109 downto 99),
                             total_valid_pixels_all(98  downto 88),
                             total_valid_pixels_all(87  downto 77),
                             total_valid_pixels_all(76  downto 66),
                             total_valid_pixels_all(65  downto 55),
                             total_valid_pixels_all(54  downto 44),
                             total_valid_pixels_all(43  downto 33),
                             total_valid_pixels_all(32  downto 22),
                             total_valid_pixels_all(21  downto 11),
                             total_valid_pixels_all(10  downto  0));

total_valid_lines_array <= (total_valid_lines_all(120 downto 110),
                             total_valid_lines_all(109 downto 99),
                             total_valid_lines_all(98  downto 88),
                             total_valid_lines_all(87  downto 77),
                             total_valid_lines_all(76  downto 66),
                             total_valid_lines_all(65  downto 55),
                             total_valid_lines_all(54  downto 44),
                             total_valid_lines_all(43  downto 33),
                             total_valid_lines_all(32  downto 22),
                             total_valid_lines_all(21  downto 11),
                             total_valid_lines_all(10  downto  0));

vid_array <= (vid_all(131 downto 120),
              vid_all(119 downto 108),
              vid_all(107 downto 96),
              vid_all(95  downto 84),
              vid_all(83  downto 72),
              vid_all(71  downto 60),
              vid_all(59  downto 48),
              vid_all(47  downto 36),
              vid_all(35  downto 24),
              vid_all(23  downto 12),
              vid_all(11  downto  0));

-- for simulation guide purposes
process(Reset, Clock)
begin

```

```

    if Reset = '1' then
        imgr_sample_en1 <= '0';
        imgr_sample_en2 <= '0';
        imgr_sample_en3 <= '0';
    elsif rising_edge(Clock) then
        imgr_sample_en1 <= imgr_sample_en;
        imgr_sample_en2 <= imgr_sample_en1;
        imgr_sample_en3 <= imgr_sample_en2;
    end if;
end process;

----- {input signal delays }-----
-- delay an imgr_sample_en cycle to align with
-- input rise/fall signals from imgr_ctrl.vhd.
process(Reset, Clock)
begin
    if (Reset = '1') then
        fval_master_int <= '0';
        lval_master_int <= '0';

        fval_master_reg <= '0';
        lval_master_reg <= '0';
    elsif rising_edge(Clock) then
        if (imgr_sample_en = '1') then
            fval_master_int <= fval_master;
            lval_master_int <= lval_master;

            fval_master_reg <= fval_master_int;
            lval_master_reg <= lval_master_int;

            -- delay for signals to fov_buf.vhd
            fval_master_reg2 <= fval_master_reg;
            lval_master_reg2 <= lval_master_reg;

            end if;

            lval_master_reg3 <= lval_master_reg2;

        end if;
end process;

-- vid_reg is delayed, thus
-- matching with fval_master_reg and the input rise/fall signals from imgr_ctrl.vhd.
vid_loop: for ii in 0 to 10 generate
    vid_delay: process(Reset, Clock)
    begin
        if (Reset = '1') then
            vid_reg(ii) <= (others => '0');
            vid_reg_int(ii) <= (others => '0');
        elsif rising_edge(Clock) then
            if imgr_sample_en = '1' then
                vid_reg_int(ii) <= vid_array(ii);
                vid_reg(ii) <= vid_reg_int(ii);
            end if;
        end if;
    end process;
end generate;

----- {END input signal delays } -----

```

```

-- keep a frame count for easy calibration identification
process(Clock, Reset)
begin
  if Reset = '1' then
    frame_count <= (others => '0');
  elsif rising_edge(Clock) then
    if (fval_fall = '1' and imgr_sample_en = '1') then
      frame_count <= frame_count + '1' ;
    end if;
  end if;
end process;

--===={{Pixel Count and Line Count added by richsinn}}====--
process(Reset, Clock)
begin
  if (Reset = '1') then
    pixel_count <= (others => '0');
    line_count <= (others => '0');
  elsif rising_edge(Clock) then
    if (imgr_sample_en = '1') then
      pixel_count <= next_pixel_count;
      line_count <= next_line_count;
    end if;
  end if;
end process;

-- next state of pixel count and line count is computed with combinational logic
next_pixel_count <= (others => '0') when (lval_master_reg = '0') else
  (pixel_count + 1);

next_line_count <= (others => '0') when (fval_master_reg = '0') else
  (line_count + 1) when (lval_fall = '1') else
  line_count;

--===={ Begin Rectangles Module Decimation process }====--
rect_loop: for ii in 0 to 10 generate
  rectangles_inst : entity work.rectangles
  Port map (
    Reset          => Reset,
    Clock          => Clock,
    imgr_sample_en => imgr_sample_en,

    vid_reg => vid_reg(ii),

    lval          => lval_master_reg,
    fval          => fval_master_reg,
    lval_fall     => lval_fall,
    pixel_count   => pixel_count,
    line_count    => line_count,

    -- user defined params for size and location of region of interest (ROI)
    start_x      => start_x_array(ii),
    start_y      => start_y_array(ii),
    total_valid_pixels => total_valid_pixels_array(ii),
    total_valid_lines  => total_valid_lines_array(ii),

    -- user defined params for digital zoom (skip mode)
    column_skip_mode => column_skip_mode,
    row_skip_mode    => row_skip_mode,

    frame_active_start => frame_active_start(ii),

```

```

        first_line      => first_line(ii),
        line_active_out => line_active(ii),

        pixel_valid => pixel_valid_rect_array(ii),
        pixel_data  => pixel_data_rect_array(ii)
    );
end generate;

-- Use first_line signal from rectangles.vhd to
-- determine the start of frame header information when writing to
-- the BRAM buffer in fov_buf.vhd.
-- If any of the first_line signals from rectangles is active it is
-- to activate the signal to fov_buf.vhd because the three imagers
-- that are spanned across will always be on the same line.
process(Clock, Reset)
begin
    if Reset = '1' then
        first_line_2_buf <= '0';
    elsif rising_edge(Clock) then
        if first_line(0) = '1' then
            first_line_2_buf <= '1';
        elsif first_line(1) = '1' then
            first_line_2_buf <= '1';
        elsif first_line(2) = '1' then
            first_line_2_buf <= '1';
        elsif first_line(3) = '1' then
            first_line_2_buf <= '1';
        elsif first_line(4) = '1' then
            first_line_2_buf <= '1';
        elsif first_line(5) = '1' then
            first_line_2_buf <= '1';
        elsif first_line(6) = '1' then
            first_line_2_buf <= '1';
        elsif first_line(7) = '1' then
            first_line_2_buf <= '1';
        elsif first_line(8) = '1' then
            first_line_2_buf <= '1';
        elsif first_line(9) = '1' then
            first_line_2_buf <= '1';
        elsif first_line(10) = '1' then
            first_line_2_buf <= '1';
        else
            first_line_2_buf <= '0';
        end if;
    end if;
end process;

-- Use frame_active_start signal from rectangles.vhd
-- to pulse the first_line_active_start signal. The first_line_active_start pulse is used to
-- in read_fov_buf.vhd to determine that the metadata header info
-- must be sent with every new frame.
first_line_active_start <= first_line_active_start_int;
process(Clock, Reset)
begin
    if Reset = '1' then
        first_line_active_start_int <= '0';
    elsif rising_edge(Clock) then
        if frame_active_start(0) = '1' then
            first_line_active_start_int <= '1';
        elsif frame_active_start(1) = '1' then
            first_line_active_start_int <= '1';
        end if;
    end if;
end process;

```

```

        elsif frame_active_start(2) = '1' then
            first_line_active_start_int <= '1';
        elsif frame_active_start(3) = '1' then
            first_line_active_start_int <= '1';
        elsif frame_active_start(4) = '1' then
            first_line_active_start_int <= '1';
        elsif frame_active_start(5) = '1' then
            first_line_active_start_int <= '1';
        elsif frame_active_start(6) = '1' then
            first_line_active_start_int <= '1';
        elsif frame_active_start(7) = '1' then
            first_line_active_start_int <= '1';
        elsif frame_active_start(8) = '1' then
            first_line_active_start_int <= '1';
        elsif frame_active_start(9) = '1' then
            first_line_active_start_int <= '1';
        elsif frame_active_start(10) = '1' then
            first_line_active_start_int <= '1';
        else
            first_line_active_start_int <= '0';
        end if;
    end if;
end process;

-- delay signal to use to latch the mux_sel0 for
-- out to be used in ImHeaderRdAddr of read_fov_buf.vhd
process(Clock, Reset)
begin
    if Reset = '1' then
        first_line_active_start_reg1 <= '0';
        first_line_active_start_reg2 <= '0';
        first_line_active_start_reg3 <= '0';
    elsif rising_edge(Clock) then
        first_line_active_start_reg1 <= first_line_active_start_int;
        first_line_active_start_reg2 <= first_line_active_start_reg1;
        first_line_active_start_reg3 <= first_line_active_start_reg2;
    end if;
end process;

line_start_loop: for ii in 0 to 10 generate
    line_start_delay: process(Clock)
    begin
        if rising_edge(Clock) then
            if imgr_sample_en = '1' then
                pixel_valid_rect_array_reg1(ii) <= pixel_valid_rect_array(ii);
                pixel_data_rect_array_reg1(ii) <= pixel_data_rect_array(ii);
            end if;
            line_start_array_reg(ii) <= line_start_array(ii);

            pixel_valid_rect_array_reg2(ii) <= pixel_valid_rect_array_reg1(ii);
            pixel_data_rect_array_reg2(ii) <= pixel_data_rect_array_reg1(ii);
        end if;
    end process;
end generate;

-- FSM to determine the three active imagers in the quadrant
process(Reset, Clock)
begin
    if Reset = '1' then
        find_valid_imgr_status <= st_findDone;
    end if;
end process;

```

```

imgr_sel <= x"F";
imgr_sel_reg <= x"F";
active_pixel_valid <= '0';
active_pixel_valid_reg <= '0';
total_valid_pixels_imgr0 <= (others =>'0');
total_valid_pixels_imgr1 <= (others =>'0');
total_valid_pixels_imgr2 <= (others =>'0');
imgr_span <= '0';
elsif rising_edge(Clock) then

    case find_valid_imgr_status is

        when st_findFirst =>
            case mux_sel0 is
                when x"0" => active_pixel_valid <= pixel_valid_rect_array_reg1(0);
                    total_valid_pixels_imgr0 <= total_valid_pixels_array(0);

                when x"1" => active_pixel_valid <= pixel_valid_rect_array_reg1(1);
                    total_valid_pixels_imgr0 <= total_valid_pixels_array(1);

                when x"2" => active_pixel_valid <= pixel_valid_rect_array_reg1(2);
                    total_valid_pixels_imgr0 <= total_valid_pixels_array(2);

                when x"3" => active_pixel_valid <= pixel_valid_rect_array_reg1(3);
                    total_valid_pixels_imgr0 <= total_valid_pixels_array(3);

                when x"4" => active_pixel_valid <= pixel_valid_rect_array_reg1(4);
                    total_valid_pixels_imgr0 <= total_valid_pixels_array(4);

                when x"5" => active_pixel_valid <= pixel_valid_rect_array_reg1(5);
                    total_valid_pixels_imgr0 <= total_valid_pixels_array(5);

                when x"6" => active_pixel_valid <= pixel_valid_rect_array_reg1(6);
                    total_valid_pixels_imgr0 <= total_valid_pixels_array(6);

                when x"7" => active_pixel_valid <= pixel_valid_rect_array_reg1(7);
                    total_valid_pixels_imgr0 <= total_valid_pixels_array(7);

                when x"8" => active_pixel_valid <= pixel_valid_rect_array_reg1(8);
                    total_valid_pixels_imgr0 <= total_valid_pixels_array(8);

                when x"9" => active_pixel_valid <= pixel_valid_rect_array_reg1(9);
                    total_valid_pixels_imgr0 <= total_valid_pixels_array(9);

                when x"A" => active_pixel_valid <= pixel_valid_rect_array_reg1(10);
                    total_valid_pixels_imgr0 <= total_valid_pixels_array(10);

                when others => active_pixel_valid <= '0';
                    total_valid_pixels_imgr0 <= (others =>'0');

            end case;
            imgr_sel <= mux_sel0;
            imgr_span <= '1';
            find_valid_imgr_status <= st_findSecond;

        -----
        when st_findSecond =>
            case mux_sel1_reg is
                when x"0" => active_pixel_valid <= pixel_valid_rect_array(0);
                    total_valid_pixels_imgr1 <= total_valid_pixels_array(0);
                when x"1" => active_pixel_valid <= pixel_valid_rect_array_reg1(1);
                    total_valid_pixels_imgr1 <= total_valid_pixels_array(1);
                when x"2" => active_pixel_valid <= pixel_valid_rect_array_reg1(2);

```

```

        total_valid_pixels_imgr1 <= total_valid_pixels_array(2);
when x"3" => active_pixel_valid <= pixel_valid_rect_array_reg1(3);
        total_valid_pixels_imgr1 <= total_valid_pixels_array(3);
when x"4" => active_pixel_valid <= pixel_valid_rect_array_reg1(4);
        total_valid_pixels_imgr1 <= total_valid_pixels_array(4);
when x"5" => active_pixel_valid <= pixel_valid_rect_array_reg1(5);
        total_valid_pixels_imgr1 <= total_valid_pixels_array(5);
when x"6" => active_pixel_valid <= pixel_valid_rect_array_reg1(6);
        total_valid_pixels_imgr1 <= total_valid_pixels_array(6);
when x"7" => active_pixel_valid <= pixel_valid_rect_array_reg1(7);
        total_valid_pixels_imgr1 <= total_valid_pixels_array(7);
when x"8" => active_pixel_valid <= pixel_valid_rect_array_reg1(8);
        total_valid_pixels_imgr1 <= total_valid_pixels_array(8);
when x"9" => active_pixel_valid <= pixel_valid_rect_array_reg1(9);
        total_valid_pixels_imgr1 <= total_valid_pixels_array(9);
when x"A" => active_pixel_valid <= pixel_valid_rect_array_reg1(10);
        total_valid_pixels_imgr1 <= total_valid_pixels_array(10);

when others => active_pixel_valid <= '0';
        total_valid_pixels_imgr1 <= (others =>'0');

end case;
imgr_sel <= mux_sel_reg;
imgr_span <= '1';
find_valid_imgr_status <= st_findThird;

```

```

when st_findThird =>
    case mux_sel2_reg2 is
--      when x"0" => active_pixel_valid <= pixel_valid_rect_array(0);
--      total_valid_pixels_imgr2 <= total_valid_pixels_array(0);
--      when x"1" => active_pixel_valid <= pixel_valid_rect_array(1);
--      total_valid_pixels_imgr2 <= total_valid_pixels_array(1);
when x"2" => active_pixel_valid <= pixel_valid_rect_array_reg1(2);
        total_valid_pixels_imgr2 <= total_valid_pixels_array(2);
when x"3" => active_pixel_valid <= pixel_valid_rect_array_reg1(3);
        total_valid_pixels_imgr2 <= total_valid_pixels_array(3);
when x"4" => active_pixel_valid <= pixel_valid_rect_array_reg1(4);
        total_valid_pixels_imgr2 <= total_valid_pixels_array(4);
when x"5" => active_pixel_valid <= pixel_valid_rect_array_reg1(5);
        total_valid_pixels_imgr2 <= total_valid_pixels_array(5);
when x"6" => active_pixel_valid <= pixel_valid_rect_array_reg1(6);
        total_valid_pixels_imgr2 <= total_valid_pixels_array(6);
when x"7" => active_pixel_valid <= pixel_valid_rect_array_reg1(7);
        total_valid_pixels_imgr2 <= total_valid_pixels_array(7);
when x"8" => active_pixel_valid <= pixel_valid_rect_array_reg1(8);
        total_valid_pixels_imgr2 <= total_valid_pixels_array(8);
when x"9" => active_pixel_valid <= pixel_valid_rect_array_reg1(9);
        total_valid_pixels_imgr2 <= total_valid_pixels_array(9);
when x"A" => active_pixel_valid <= pixel_valid_rect_array_reg1(10);
        total_valid_pixels_imgr2 <= total_valid_pixels_array(10);

    when others => active_pixel_valid <= '0';
        total_valid_pixels_imgr2 <= (others =>'0');

    end case;
imgr_sel <= mux_sel2_reg2;
imgr_span <= '1';
find_valid_imgr_status <= st_findDone;

```

```

when st_findDone =>
    if imgr_sample_en = '1' and lval_master_reg3 = '1' then

```



```

        find_valid_imgr_status <= st_findFirst;
    end if;
    imgr_sel <= x"F";
    imgr_span <= '0';
    active_pixel_valid <= '0';
end case;

    imgr_sel_reg <= imgr_sel;
    imgr_span_reg <= imgr_span;
    active_pixel_valid_reg <= active_pixel_valid;
end if;
end process;

process(Clock)
begin
    if rising_edge(Clock) then
        if imgr_sample_en = '1' then
            total_valid_pixels_img1_reg1 <= total_valid_pixels_img1;
            total_valid_pixels_img1_reg2 <= total_valid_pixels_img1;
        end if;
    end if;
end process;

-- NOTE in the latency delays: active_pixel_data, active_pixel_valid_reg,
-- imgr_sel_reg are aligned.
process(Clock)
begin
    if rising_edge(Clock) then

        case imgr_sel is

            when x"0" => active_pixel_data <= pixel_data_rect_array_reg2(0);
            when x"1" => active_pixel_data <= pixel_data_rect_array_reg2(1);
            when x"2" => active_pixel_data <= pixel_data_rect_array_reg2(2);
            when x"3" => active_pixel_data <= pixel_data_rect_array_reg2(3);
            when x"4" => active_pixel_data <= pixel_data_rect_array_reg2(4);
            when x"5" => active_pixel_data <= pixel_data_rect_array_reg2(5);
            when x"6" => active_pixel_data <= pixel_data_rect_array_reg2(6);
            when x"7" => active_pixel_data <= pixel_data_rect_array_reg2(7);
            when x"8" => active_pixel_data <= pixel_data_rect_array_reg2(8);
            when x"9" => active_pixel_data <= pixel_data_rect_array_reg2(9);
            when x"A" => active_pixel_data <= pixel_data_rect_array_reg2(10);
            when others => active_pixel_data <= (others => '0');

        end case;
    end if;
end process;

process(Reset, Clock)
begin
    if (Reset = '1') then
        lval_rise_int <= '0';
        lval_rise_reg_2_buf <= '0';

        lval_fall_int <= '0';
        lval_fall_reg_2_buf <= '0';

    elsif rising_edge(Clock) then
        if imgr_sample_en = '1' then
            lval_rise_int <= lval_rise;
            lval_rise_reg_2_buf <= lval_rise_int;
        end if;
    end if;
end process;

```

```

        lval_fall_int <= lval_fall;
        lval_fall_reg_2_buf <= lval_fall_int;
    end if;
end if;
end process;

```

```

-- determine the values for mux_sel0,1,2
process(Clock, Reset)
begin
    if Reset = '1' then
        mux_sel0 <= x"F";
    elsif rising_edge(Clock) then
        if line_active(0) = '1' then
            mux_sel0 <= x"0";
        elsif line_active(1) = '1' then
            mux_sel0 <= x"1";
        elsif line_active(2) = '1' then
            mux_sel0 <= x"2";
        elsif line_active(3) = '1' then
            mux_sel0 <= x"3";
        elsif line_active(4) = '1' then
            mux_sel0 <= x"4";
        elsif line_active(5) = '1' then
            mux_sel0 <= x"5";
        elsif line_active(6) = '1' then
            mux_sel0 <= x"6";
        elsif line_active(7) = '1' then
            mux_sel0 <= x"7";
        elsif line_active(8) = '1' then
            mux_sel0 <= x"8";
        elsif line_active(9) = '1' then
            mux_sel0 <= x"9";
        elsif line_active(10) = '1' then
            mux_sel0 <= x"A";
        else
            mux_sel0 <= x"F";
        end if;

        mux_sel0_reg <= mux_sel0;

    end if;
end process;

-- latch mux_sel0 value only on new frames. To be
-- sent to read_fov_buf.vhd so that it can be used to
-- determine ImHeaderRdAddr.
process(Clock, Reset)
begin
    if Reset = '1' then
        first_imgr_sel <= (others => '0');
    elsif rising_edge(Clock) then
        if first_line_active_start_reg3 = '1' then
            first_imgr_sel <= mux_sel0;
        end if;
    end if;
end process;

process(Clock, Reset)
begin
    if Reset = '1' then

```

```

        mux_sel1 <= x"F";
        mux_sel1_reg <= x"F";
    elsif rising_edge(Clock) then
        if line_active(0) = '1' then
            mux_sel2 <= x"0";
        --
        if line_active(1) = '1' and mux_sel0 /= x"1" then
            mux_sel1 <= x"1";
        elsif line_active(2) = '1' and mux_sel0 /= x"2" then
            mux_sel1 <= x"2";
        elsif line_active(3) = '1' and mux_sel0 /= x"3" then
            mux_sel1 <= x"3";
        elsif line_active(4) = '1' and mux_sel0 /= x"4" then
            mux_sel1 <= x"4";
        elsif line_active(5) = '1' and mux_sel0 /= x"5" then
            mux_sel1 <= x"5";
        elsif line_active(6) = '1' and mux_sel0 /= x"6" then
            mux_sel1 <= x"6";
        elsif line_active(7) = '1' and mux_sel0 /= x"7" then
            mux_sel1 <= x"7";
        elsif line_active(8) = '1' and mux_sel0 /= x"8" then
            mux_sel1 <= x"8";
        elsif line_active(9) = '1' and mux_sel0 /= x"9" then
            mux_sel1 <= x"9";
        elsif line_active(10) = '1' and mux_sel0 /= x"A" then
            mux_sel1 <= x"A";
        else
            mux_sel1 <= x"F";
        end if;

        mux_sel1_reg <= mux_sel1;

    end if;
end process;

process(Clock, Reset)
begin
    if Reset = '1' then
        mux_sel2 <= x"F";
        mux_sel2_reg1 <= x"F";
        mux_sel2_reg2 <= x"F";
    elsif rising_edge(Clock) then
        if line_active(0) = '1' then
            mux_sel2 <= x"0";
        --
        if line_active(1) = '1' then
            mux_sel2 <= x"1";
        --
        if line_active(2) = '1' and mux_sel1 /= x"2" and mux_sel0 /= x"2" then
            mux_sel2 <= x"2";
        elsif line_active(3) = '1' and mux_sel1 /= x"3" and mux_sel0 /= x"3" then
            mux_sel2 <= x"3";
        elsif line_active(4) = '1' and mux_sel1 /= x"4" and mux_sel0 /= x"4" then
            mux_sel2 <= x"4";
        elsif line_active(5) = '1' and mux_sel1 /= x"5" and mux_sel0 /= x"5" then
            mux_sel2 <= x"5";
        elsif line_active(6) = '1' and mux_sel1 /= x"6" and mux_sel0 /= x"6" then
            mux_sel2 <= x"6";
        elsif line_active(7) = '1' and mux_sel1 /= x"7" and mux_sel0 /= x"7" then
            mux_sel2 <= x"7";
        elsif line_active(8) = '1' and mux_sel1 /= x"8" and mux_sel0 /= x"8" then
            mux_sel2 <= x"8";
        elsif line_active(9) = '1' and mux_sel1 /= x"9" and mux_sel0 /= x"9" then
            mux_sel2 <= x"9";

```

```

        elsif line_active(10) = '1' and mux_sel1 /= x"A" and mux_sel0 /= x"A" then
            mux_sel2 <= x"A";
        else
            mux_sel2 <= x"F";
        end if;

        mux_sel2_reg1 <= mux_sel2;
        mux_sel2_reg2 <= mux_sel2_reg1;
    end if;
end process;

-----

process(Clock)
begin
    if rising_edge(Clock) then
        if (line_count = conv_std_logic_vector(0, line_count'LENGTH)) then
            buffers_empty <= '1';
        else
            buffers_empty <= '0';
        end if;
    end if;
end process;

fov_buf_i : entity work.fov_buf
generic map(
    QUAD_ID => QUAD_ID,
    CAMERA_ID => CAMERA_ID,
    USER_ID => USER_ID
)
Port map(
    Reset          => Reset ,
    Clock          => Clock ,
    imgr_sample_en => imgr_sample_en ,

    first_line => first_line_2_buf ,
    lval       => lval_master_reg2 ,
    lval_rise  => lval_rise_reg_2_buf ,
    lval_fall  => lval_fall_reg_2_buf ,

    total_valid_pixels_imgr0 => total_valid_pixels_imgr0 ,
    total_valid_pixels_imgr1 => total_valid_pixels_imgr1 ,
    total_valid_pixels_imgr2 => total_valid_pixels_imgr2 ,

    imgr_ID          => imgr_sel_reg ,
    imgr_span        => imgr_span ,
    active_pixel_valid => active_pixel_valid_reg ,
    active_pixel_data => active_pixel_data ,

    ReadEn          => ReadEn ,
    wr_done_pulse_out => wr_done_pulse ,
    rd_done         => rd_done ,
    ReadData        => ReadData
);

end Behavioral;

```

A.2 rectangles.vhd

The following VHDL code is the implementation of *Rectangles* module described in Section 4.3.1.

```
-- Rectangles module.
-- This is instantiated in the foveation_top.vhd.
-- Each Rectangles module corresponds to one imager, thus there are eleven
-- Rectangles instantiation in one quadrant.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- custom package with some helper functions
use work.sweet_package.all;

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

entity rectangles is
Port (
    Reset          : in std_logic; -- async reset
    Clock          : in std_logic; -- system clock
    imgr_sample_en : in std_logic;

    vid_reg        : in std_logic_vector(11 downto 0);

    lval           : in std_logic;
    fval           : in std_logic;
    lval_fall      : in std_logic;
    pixel_count    : in std_logic_vector(11 downto 0);
    line_count     : in std_logic_vector(11 downto 0);

    -- user defined params for size and location of region of interest (ROI)
    start_x        : in std_logic_vector (11 downto 0);
    start_y        : in std_logic_vector (11 downto 0);
    total_valid_pixels : in std_logic_vector(10 downto 0);
    total_valid_lines  : in std_logic_vector(10 downto 0);

    -- user defined params for digital zoom (skip mode)
    column_skip_mode : in std_logic_vector (7 downto 0);
    row_skip_mode    : in std_logic_vector (7 downto 0);

    frame_active_start : out std_logic;
    first_line         : out std_logic;
    line_active_out     : out std_logic;

    pixel_valid       : out std_logic;
    pixel_data        : out std_logic_vector(11 downto 0)
);

end rectangles;

architecture Behavioral of rectangles is

    signal total_valid_pixels_latched, total_valid_lines_latched : std_logic_vector(10 downto 0);
```

```

signal valid_pixel_counter , valid_line_counter : std_logic_vector(10 downto 0);

signal max_row_skip_counter_int , max_column_skip_counter_int : std_logic_vector ( 8 downto 0);
signal max_row_skip_counter , max_column_skip_counter : std_logic_vector ( 8 downto 0);
signal row_skip_counter , column_skip_counter : std_logic_vector ( 8 downto 0);

signal start_x_valid , start_y_valid : std_logic_vector(11 downto 0);

signal column_skip_mode_valid , row_skip_mode_valid : std_logic_vector(7 downto 0);

signal skip_pixel , skip_row_pixel , skip_column_pixel : std_logic;

type column_pixel_status_type is (COL_INIT, COL_VALID_PIXEL, COL_SKIP_PIXEL);
signal column_pixel_status : column_pixel_status_type;

type row_pixel_status_type is (ROW_INIT, ROW_VALID_PIXEL, ROW_SKIP_PIXEL);
signal row_pixel_status : row_pixel_status_type;

signal first_line_int , first_line_reg : std_logic;

signal line_active_int : std_logic;

signal pixel_valid_int : std_logic;

begin

----- { Windowing Logic } -----

----- only update the parameters in between frames or in between lines. -----
process(Clock)
begin
    if rising_edge(Clock) then
        if fval = '0' then
            start_x_valid <= start_x;
            start_y_valid <= start_y;
            column_skip_mode_valid <= column_skip_mode ;
            row_skip_mode_valid <= row_skip_mode ;

            end if;

            if lval = '0' then
                total_valid_pixels_latched <= total_valid_pixels;
                total_valid_lines_latched <= total_valid_lines;
            end if;

            end if;
        end process;

----- Determine the maximum count needed to determine the skip mode -----
-- (This is the total number of pixels in an interval -> 2 valid pixels + skipped.pixels).
max_row_skip_counter_int <= (others => '0') when (row_skip_mode = conv_std_logic_vector(0,
    row_skip_mode'length)) else
    ((row_skip_mode_valid(7 downto 0) & '0') - 1 );
max_column_skip_counter_int <= (others => '0') when (column_skip_mode = conv_std_logic_vector
    (0, column_skip_mode'length)) else
    ((column_skip_mode_valid(7 downto 0) & '0') - 1 );

-- only update the new skip counter in between frames.
process(Reset , Clock)
begin

```

```

    if (Reset = '1') then
        max_row_skip_counter    <= (others => '0');
        max_column_skip_counter <= (others => '0');
    elsif rising_edge(Clock) then
        if fval = '0' then
            max_row_skip_counter    <= max_row_skip_counter_int;
            max_column_skip_counter <= max_column_skip_counter_int;
        end if;
    end if;
end process;

--== Determine Pixel skipping ==--
-- Signal 'skip_pixel' is determined by the two state machines below (row and col mode)
skip_pixel <= skip_row_pixel or skip_column_pixel;

-- State machine to determine col skip mode
process(Reset, Clock)
begin
    if (Reset = '1') then
        skip_column_pixel <= '1';
        column_skip_counter <= (others => '0');
        valid_pixel_counter <= (others => '0');
        column_pixel_status <= COL_INIT;
    elsif rising_edge(Clock) then
        case column_pixel_status is

            -----
            when COL_INIT =>
                if (lval = '0' or fval = '0' or total_valid_pixels_latched = conv_std_logic_vector
                    (0, total_valid_pixels_latched 'LENGTH)) then
                    skip_column_pixel <= '1';
                    column_pixel_status <= COL_INIT;
                elsif (pixel_count = start_x_valid) then
                    skip_column_pixel <= '0';
                    column_pixel_status <= COL_VALID_PIXEL;
                else
                    skip_column_pixel <= '1';
                    column_pixel_status <= COL_INIT;
                end if;

                column_skip_counter <= (others => '0');

            -----
            when COL_VALID_PIXEL =>
                if (lval = '0' or fval = '0' or total_valid_pixels_latched = conv_std_logic_vector
                    (0, total_valid_pixels_latched 'LENGTH)) then
                    skip_column_pixel <= '1';
                    column_pixel_status <= COL_INIT;

                elsif valid_pixel_counter = total_valid_pixels_latched then
                    valid_pixel_counter <= (others => '0');
                    skip_column_pixel <= '1';
                    column_pixel_status <= COL_INIT;

                elsif (column_skip_mode_valid = conv_std_logic_vector(0, column_skip_mode_valid '
                    LENGTH) and imgr_sample_en = '1') then
                    skip_column_pixel <= '0';
                    valid_pixel_counter <= valid_pixel_counter + 1;
                    column_pixel_status <= COL_VALID_PIXEL;

                -- before skipping, always output first two pixels to maintain bayer pattern

```

```

    elsif(column_skip_counter = x"01" and imgr_sample_en = '1') then
        column_skip_counter <= column_skip_counter + 1;
        valid_pixel_counter <= valid_pixel_counter + 1;
        skip_column_pixel <= '1';
        column_pixel_status <= COL_SKIP_PIXEL;
    elsif imgr_sample_en = '1' then
        column_skip_counter <= column_skip_counter + 1;
        valid_pixel_counter <= valid_pixel_counter + 1;
        skip_column_pixel <= '0';
        column_pixel_status <= COL_VALID_PIXEL;
    end if;

    -----
when COL_SKIP_PIXEL =>
    if(lval = '0' or fval = '0' or total_valid_pixels_latched = conv_std_logic_vector
        (0, total_valid_pixels_latched'LENGTH)) then
        skip_column_pixel <= '1';
        column_pixel_status <= COL_INIT;
    elsif valid_pixel_counter = total_valid_pixels_latched then
        skip_column_pixel <= '1';
        valid_pixel_counter <= (others => '0');
        column_pixel_status <= COL_INIT;

    elsif (column_skip_counter = max_column_skip_counter and imgr_sample_en = '1')
        then
            skip_column_pixel <= '0';
            column_skip_counter <= (others => '0');
            column_pixel_status <= COL_VALID_PIXEL;
        elsif imgr_sample_en = '1' then
            column_skip_counter <= column_skip_counter + 1;
            column_pixel_status <= COL_SKIP_PIXEL;
        end if;

    end case;
end if;
end process;

-- State machine to determine row skip mode
process(Reset, Clock)
begin
    if (Reset = '1') then
        skip_row_pixel <= '1';
        row_skip_counter <= (others => '0');
        valid_line_counter <= (others => '0');
        row_pixel_status <= ROW_INIT;
        line_active_int <= '0';
        -- frame_active_start <= '0';
    elsif rising_edge(Clock) then
        case row_pixel_status is

            -----
when ROW_INIT =>
            if (fval = '0' or total_valid_lines_latched = conv_std_logic_vector(0,
                total_valid_lines_latched'LENGTH)) then
                skip_row_pixel <= '1';
                row_pixel_status <= ROW_INIT;
            elsif (lval = '1' and line_count = start_y_valid) then
                skip_row_pixel <= '0';
                row_pixel_status <= ROW_VALID_PIXEL;
            else
                skip_row_pixel <= '1';
            end if;
        end case;
    end if;
end process;

```



```

        row_pixel_status <= ROW_INIT;
    end if;

    row_skip_counter <= (others => '0');
    line_active_int <= '0';

-----
when ROW_VALID_PIXEL =>
    if (fval = '0' or total_valid_lines_latched = conv_std_logic_vector(0,
        total_valid_lines_latched'LENGTH)) then
        skip_row_pixel <= '1';
        row_pixel_status <= ROW_INIT;

    elsif valid_line_counter = total_valid_lines_latched then
        valid_line_counter <= (others => '0');
        skip_row_pixel <= '1';
        row_pixel_status <= ROW_INIT;

    elsif (row_skip_mode_valid = conv_std_logic_vector(0, row_skip_mode_valid'LENGTH)
        and lval_fall = '1' and imgr_sample_en = '1') then
        skip_row_pixel <= '0';
        valid_line_counter <= valid_line_counter + 1;
        row_pixel_status <= ROW_VALID_PIXEL;

    -- before skipping, always output first two pixels to maintain bayer pattern
    elsif (row_skip_counter = x"01" and lval_fall = '1' and imgr_sample_en = '1') then
        skip_row_pixel <= '1';
        row_skip_counter <= row_skip_counter + 1;
        valid_line_counter <= valid_line_counter + 1;
        row_pixel_status <= ROW_SKIP_PIXEL;
    elsif (lval_fall = '1' and imgr_sample_en = '1') then
        skip_row_pixel <= '0';
        row_skip_counter <= row_skip_counter + 1;
        valid_line_counter <= valid_line_counter + 1;
        row_pixel_status <= ROW_VALID_PIXEL;
    end if;

    --this signal's assertion is used to indicate that there are valid pixels on this
    line
    if (lval = '1') then
        line_active_int <= '1';
    else
        line_active_int <= '0';
    end if;

-----
when ROW_SKIP_PIXEL =>
    if (fval = '0' or total_valid_lines_latched = conv_std_logic_vector(0,
        total_valid_lines_latched'LENGTH)) then
        skip_row_pixel <= '1';
        row_pixel_status <= ROW_INIT;

    elsif valid_line_counter = total_valid_lines_latched then
        skip_row_pixel <= '1';
        valid_line_counter <= (others => '0');
        row_pixel_status <= ROW_INIT;

    elsif (row_skip_counter = max_row_skip_counter and lval_fall = '1' and
        imgr_sample_en = '1') then
        skip_row_pixel <= '0';
        row_skip_counter <= (others => '0');

```

```

        row_pixel_status <= ROW_VALID_PIXEL;
    elsif (lval_fall = '1' and imgr_sample_en = '1') then
        skip_row_pixel <= '1';
        row_skip_counter <= row_skip_counter + 1;
        row_pixel_status <= ROW_SKIP_PIXEL;
    end if;

    line_active_int <= '0';

end case;

end if;
end process;

----- {Assign outputs} -----

first_line_int <= '0' when ( (lval = '0') or (total_valid_lines_latched =
    conv_std_logic_vector(0, total_valid_lines_latched 'LENGTH')) ) else
    '1' when ( line_count = start_y_valid )
    else '0';

first_line <= first_line_int;

process(Clock, Reset)
begin
    if Reset = '1' then
        first_line_reg <= '0';
    elsif rising_edge(Clock) then
        first_line_reg <= first_line_int;

        frame_active_start <= first_line_int and not first_line_reg;
    end if;
end process;

process(Clock, Reset)
begin
    if (Reset = '1') then
        pixel_valid <= '0';
        pixel_data <= (others => '0');
        line_active_out <= '0';
    elsif rising_edge(Clock) then
        if imgr_sample_en = '1' then
            line_active_out <= line_active_int;
            if fval = '0' or lval = '0' then
                pixel_valid <= '0';
                pixel_data <= (others => '0');
            else
                pixel_valid <= not skip_pixel;
                pixel_data <= vid_reg;
            end if;
        end if;
    end if;
end process;

end Behavioral;

```

A.3 fov_buf.vhd

The following VHDL code is the implementation of the *Buffer* module described in Section 4.3.3.

```
-- This is the Buffer module with the 2 BRAM instantiations.
-- It is instantiated in the foveation_top.vhd top module.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

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

-- custom package with some helper functions
use work.sweet_package.all;

entity fov_buf is
generic(
  QUAD_ID   : in std_logic_vector(1 downto 0) := "00";
  CAMERA_ID : in std_logic_vector(1 downto 0) := "00";
  USER_ID   : in std_logic_vector(1 downto 0) := "00"
);
Port (
  Reset           : in std_logic; -- async reset
  Clock           : in std_logic; -- system clock
  imgr_sample_en : in std_logic;

  first_line      : in std_logic;
  lval            : in std_logic;
  lval_rise       : in std_logic;
  lval_fall       : in std_logic;

  -- total_valid_pixels : in std_logic_vector(9 downto 0); -- # of valid pixels on one line of one
  -- imgr
  total_valid_pixels_imgr0 : in std_logic_vector(10 downto 0);
  total_valid_pixels_imgr1 : in std_logic_vector(10 downto 0);
  total_valid_pixels_imgr2 : in std_logic_vector(10 downto 0);

  imgr_ID         : in std_logic_vector(3 downto 0);
  imgr_span       : in std_logic;
  active_pixel_valid : in std_logic;
  active_pixel_data : in std_logic_vector(11 downto 0);

  ReadEn         : in std_logic;
  wr_done_pulse_out : out std_logic;
  rd_done        : out std_logic;
  ReadData       : out std_logic_vector(15 downto 0)
);
end fov_buf;

architecture Behavioral of fov_buf is

  component burst_pixel_buf
    port (
      clka : IN std_logic;

```

```

        dina : IN std_logic_VECTOR(15 downto 0);
        addra : IN std_logic_VECTOR(10 downto 0);
        wea : IN std_logic_VECTOR( 0 downto 0);
        clk_b : IN std_logic;
        addr_b : IN std_logic_VECTOR(10 downto 0);
        dout_b : OUT std_logic_VECTOR(15 downto 0)
    );
END component;

constant START_OF_FRAME : std_logic_vector( 3 downto 0) := x"A" ;
constant START_OF_LINE : std_logic_vector( 3 downto 0) := x"8" ;
constant END_OF_LINE : std_logic_vector( 3 downto 0) := x"4" ;

signal lval_reg1, lval_reg2 : std_logic;
signal lval_rise_reg : std_logic;
signal lval_fall_reg : std_logic;

signal valid_pixel_count_imgr0, valid_pixel_count_imgr1, valid_pixel_count_imgr2 :
    std_logic_vector(10 downto 0);
signal valid_pixel_count_imgr0_reg, valid_pixel_count_imgr1_reg, valid_pixel_count_imgr2_reg :
    std_logic_vector(10 downto 0);

signal latch_en : std_logic;
signal overall_valid_pixels_latched : std_logic_vector(10 downto 0);
signal overall_valid_pixel_count : std_logic_vector(10 downto 0);

signal active_pixel_data_reg1, active_pixel_data_reg2, active_pixel_data_reg3,
    active_pixel_data_reg4, active_pixel_data_reg5 : std_logic_vector(11 downto 0);
signal active_pixel_valid_reg1, active_pixel_valid_reg2, active_pixel_valid_reg3,
    active_pixel_valid_reg4, active_pixel_valid_reg5 : std_logic;

signal imgr_ID_reg1, imgr_ID_reg2, imgr_ID_reg3, imgr_ID_reg4, imgr_ID_reg5 : std_logic_vector(3
    downto 0);
signal imgr_span_reg1, imgr_span_reg2 : std_logic;

signal total_valid_pixels_imgr0_latched, total_valid_pixels_imgr1_latched,
    total_valid_pixels_imgr2_latched : std_logic_vector(10 downto 0);

signal header_en_buf, header_en_buf_reg1, header_en_buf_reg2 : std_logic;
signal frame_start_flag, frame_start_flag_reg1, frame_start_flag_reg2, frame_start_flag_reg3 :
    std_logic;
signal wr_buf_sel : std_logic_vector(0 downto 0);
signal rd_buf_sel : std_logic_vector(0 downto 0);

signal data_prefix : std_logic_vector(3 downto 0);

signal imgr_active, imgr_active_reg1, imgr_active_reg2 : std_logic_vector(2 downto 0);

signal addr_1_offset, addr_2_offset : std_logic_vector(10 downto 0);

signal wr_addr, rd_addr : std_logic_vector(10 downto 0);
signal write_en_0, write_en_1 : std_logic_vector(0 downto 0);

signal write_en_tmp : std_logic_vector(0 downto 0);

signal data_in_0, data_in_1 : std_logic_vector(15 downto 0);
signal data_out_0, data_out_1 : std_logic_vector(15 downto 0);

type wr_addr_ctrl_type is (st_wrIdle, st_wrHeaderPacket, st_wrImgr0, st_wrImgr1, st_wrImgr2);
signal wr_addr_ctrl : wr_addr_ctrl_type;

```

```

signal wr_done : std_logic;

signal end_of_line_flag, end_of_line_flag_reg1, end_of_line_flag_reg2 : std_logic;

begin

  -- signal delays
  -- to do later: too many system clock delays->when appropriate should instead make use of
    imgr_sample_en delays.

  process(Clock, Reset)
  begin
    if Reset = '1' then
      lval_reg1 <= '0';
      lval_fall_reg <= '0';
      lval_rise_reg <= '0';
      active_pixel_valid_reg1 <= '0';
      active_pixel_valid_reg2 <= '0';
      active_pixel_valid_reg3 <= '0';
      active_pixel_valid_reg4 <= '0';
      active_pixel_valid_reg5 <= '0';
      active_pixel_data_reg1 <= (others => '0');
      active_pixel_data_reg2 <= (others => '0');
      active_pixel_data_reg3 <= (others => '0');
      active_pixel_data_reg4 <= (others => '0');
      active_pixel_data_reg5 <= (others => '0');
      imgr_ID_reg1 <= (others => '0');
      imgr_ID_reg2 <= (others => '0');
      imgr_ID_reg3 <= (others => '0');
      imgr_ID_reg4 <= (others => '0');
      imgr_ID_reg5 <= (others => '0');
      imgr_span_reg1 <= '0';
      imgr_span_reg2 <= '0';
      valid_pixel_count_imgr0_reg <= (others => '0');
      valid_pixel_count_imgr1_reg <= (others => '0');
      valid_pixel_count_imgr2_reg <= (others => '0');
    elsif rising_edge(Clock) then
      if imgr_sample_en = '1' then
        lval_reg1 <= lval;
        lval_reg2 <= lval_reg1;

        lval_fall_reg <= lval_fall;

      end if;

      lval_rise_reg <= lval_rise;

      active_pixel_valid_reg1 <= active_pixel_valid;
      active_pixel_valid_reg2 <= active_pixel_valid_reg1;
      active_pixel_valid_reg3 <= active_pixel_valid_reg2;
      active_pixel_valid_reg4 <= active_pixel_valid_reg3;
      active_pixel_valid_reg5 <= active_pixel_valid_reg4;

      active_pixel_data_reg1 <= active_pixel_data;
      active_pixel_data_reg2 <= active_pixel_data_reg1;
      active_pixel_data_reg3 <= active_pixel_data_reg2;
      active_pixel_data_reg4 <= active_pixel_data_reg3;
      active_pixel_data_reg5 <= active_pixel_data_reg4;

      imgr_ID_reg1 <= imgr_ID;
      imgr_ID_reg2 <= imgr_ID_reg1;
      imgr_ID_reg3 <= imgr_ID_reg2;

```

```

    imgr_ID_reg4 <= imgr_ID_reg3;
    imgr_ID_reg5 <= imgr_ID_reg4;

    imgr_span_reg1 <= imgr_span;
    imgr_span_reg2 <= imgr_span_reg1;

    valid_pixel_count_imgr0_reg <= valid_pixel_count_imgr0;
    valid_pixel_count_imgr1_reg <= valid_pixel_count_imgr1;
    valid_pixel_count_imgr2_reg <= valid_pixel_count_imgr2;

end if;
end process;

-- latch the total_valid_pixels for the entire line. Only updates the new total valid pixels
-- at the beginning of an entire line for the active imagers.
process(Clock, Reset)
begin
    if Reset = '1' then
        total_valid_pixels_imgr0_latched <= (others => '0');
        total_valid_pixels_imgr1_latched <= (others => '0');
        total_valid_pixels_imgr2_latched <= (others => '0');

        overall_valid_pixels_latched <= (others => '0');
    elsif rising_edge(Clock) then
        if latch_en = '1' then
            total_valid_pixels_imgr0_latched <= total_valid_pixels_imgr0;
            total_valid_pixels_imgr1_latched <= total_valid_pixels_imgr1;
            total_valid_pixels_imgr2_latched <= total_valid_pixels_imgr2;

            overall_valid_pixels_latched <= total_valid_pixels_imgr0 + total_valid_pixels_imgr1 +
                total_valid_pixels_imgr2;
        end if;
    end if;
end process;

-- Must use delayed signals for the offset calculation because the
-- input from foveation_top module resets values before valid pixels ends.
-- Notice in addr_2_offset, we have to use total_valid_pixels_imgr0_reg2,
-- which is twice as delayed.
addr_1_offset <= total_valid_pixels_imgr0_latched;
addr_2_offset <= total_valid_pixels_imgr0_latched + total_valid_pixels_imgr1_latched;

-- Mux the appropriate write address for the BRAMs
-- imgr_active signal is determined by the wr_addr_ctrl FSM below.
write_en_0(0) <= write_en_tmp(0) when wr_buf_sel = "0" else '0';
write_en_1(0) <= write_en_tmp(0) when wr_buf_sel = "1" else '0';

process(Reset, Clock)
begin
    if Reset = '1' then
        wr_addr <= (others => '0');
        write_en_tmp <= "0";
        overall_valid_pixel_count <= (others => '0');
    elsif rising_edge(Clock) then
        if overall_valid_pixels_latched = "0000000000" then
            overall_valid_pixel_count <= (others => '0');
            write_en_tmp <= "0";
        elsif overall_valid_pixel_count >= overall_valid_pixels_latched then
            overall_valid_pixel_count <= (others => '0');
            write_en_tmp <= "0";
        else
            write_en_tmp <= "0";
        end if;
    end if;
end process;

```

```

case imgr_active_reg2 is
-----
when "000" =>
    -- write header information into address space 0 of BRAM.
    wr_addr <= (others => '0');
    if header_en_buf_reg2 = '1' then
        write_en_tmp <= "1";
    else
        write_en_tmp <= "0";
    end if;

-----

when "001" =>
    wr_addr <= valid_pixel_count.imgr0_reg ;
    if active_pixel_valid_reg5 = '1' then
        write_en_tmp <= "1";
        overall_valid_pixel_count <= overall_valid_pixel_count + 1;
    else
        write_en_tmp <= "0";
    end if;

-----

when "010" =>
    wr_addr <= addr_1_offset + valid_pixel_count.imgr1_reg ;
    if active_pixel_valid_reg5 = '1' then
        write_en_tmp <= "1";
        overall_valid_pixel_count <= overall_valid_pixel_count + 1;
    else
        write_en_tmp <= "0";
    end if;

-----

when "100" =>
    wr_addr <= addr_2_offset + valid_pixel_count.imgr2_reg ;
    if active_pixel_valid_reg5 = '1' then
        write_en_tmp <= "1";
        overall_valid_pixel_count <= overall_valid_pixel_count + 1;
    else
        write_en_tmp <= "0";
    end if;

-----

when others =>
    wr_addr <= (others => '0');
    write_en_tmp <= "0";
end case;
end if;
end if;
end process;

process(Clock, Reset)
begin
    if (Reset = '1') then
        end_of_line_flag <= '0';
    elsif rising_edge(Clock) then
        if (wr_addr = overall_valid_pixels_latched - 1) then
            end_of_line_flag <= '1';
        else
            end_of_line_flag <= '0';
        end if;
    end if;

```

```

        end_of_line_flag_reg1 <= end_of_line_flag;
        end_of_line_flag_reg2 <= end_of_line_flag_reg1;

    end if;
end process;

process(Clock, Reset)
begin
    if Reset = '1' then
        wr_done <= '0';
    elsif rising_edge(Clock) then
        if overall_valid_pixels_latched = "0000000000" then
            wr_done <= '0';
        elsif overall_valid_pixel_count >= overall_valid_pixels_latched then
            wr_done <= '1';
        else
            wr_done <= '0';
        end if;
    end if;
end process;
wr_done_pulse_out <= wr_done;

-- write_buf_sel is a std_logic_vector(0 downto 0), hence the
-- double quotes (") around the single digit.
-- To be used for "ping-ponging" between the BRAMs line by line.
process(Reset, Clock)
begin
    if Reset = '1' then
        wr_buf_sel <= "0";
    elsif rising_edge(Clock) then
        if wr_done = '1' then
            wr_buf_sel <= wr_buf_sel + 1;
        end if;
    end if;
end process;

-- Assign the data that goes into the BRAM
data_prefix <= (others => '0');
process(Reset, Clock)
begin
    if Reset = '1' then
        data_in_0 <= (others => '0');
        data_in_1 <= (others => '0');
    elsif rising_edge(Clock) then

        if wr_buf_sel = "0" then
            -- if bit15 (MSB) is high, then it's a control word
            if frame_start_flag_reg3 = '1' then
                data_in_0 <= START_OF_FRAME & QUAD_ID & CAMERA_ID & "00" & x"0" & USER_ID ;
            elsif header_en_buf_reg2 = '1' then
                data_in_0 <= START_OF_LINE & QUAD_ID & CAMERA_ID & "00" & x"0" & USER_ID;
            elsif end_of_line_flag_reg2 = '1' then
                data_in_0 <= END_OF_LINE & active_pixel_data_reg5;
            else
                data_in_0 <= data_prefix & active_pixel_data_reg5;
            end if;

        elsif wr_buf_sel = "1" then
            -- if bit15 (MSB) is high, then it's a control word

```



```

        if frame_start_flag_reg3 = '1' then
            data_in_1 <= START_OF_FRAME & QUAD_ID & CAMERA_ID & "00" & x"0" & USER_ID ;
        elsif header_en_buf_reg2 = '1' then
            data_in_1 <= START_OF_LINE & QUAD_ID & CAMERA_ID & "00" & x"0" & USER_ID;
        elsif end_of_line_flag_reg2 = '1' then
            data_in_1 <= END_OF_LINE & active_pixel_data_reg5;
        else
            data_in_1 <= data_prefix & active_pixel_data_reg5;
        end if;

    end if;
end if;
end process;

process(Reset, Clock)
begin
    if Reset = '1' then
        header_en_buf_reg1 <= '0';
        header_en_buf_reg2 <= '0';
        frame_start_flag_reg1 <= '0';
        frame_start_flag_reg2 <= '0';
        frame_start_flag_reg3 <= '0';
        imgr_active_reg1 <= (others => '0');
        imgr_active_reg2 <= (others => '0');
    elsif rising_edge(Clock) then
        header_en_buf_reg1 <= header_en_buf;
        header_en_buf_reg2 <= header_en_buf_reg1;

        frame_start_flag_reg1 <= frame_start_flag;
        frame_start_flag_reg2 <= frame_start_flag_reg1;
        frame_start_flag_reg3 <= frame_start_flag_reg2;

        imgr_active_reg1 <= imgr_active;
        imgr_active_reg2 <= imgr_active_reg1;
    end if;
end process;

process(Reset, Clock)
begin
    if Reset = '1' then
        header_en_buf <= '0';
        wr_addr_ctrl <= st_wrIdle;
        valid_pixel_count_imgr0 <= (others => '0');
        valid_pixel_count_imgr1 <= (others => '0');
        valid_pixel_count_imgr2 <= (others => '0');
        imgr_active <= "000";
        latch_en <= '0';
        frame_start_flag <= '0';
    elsif rising_edge(Clock) then
        case wr_addr_ctrl is

            when st_wrIdle =>
                if lval_fall_reg = '1' then
                    if valid_pixel_count_imgr0 = total_valid_pixels_imgr0_latched then
                        valid_pixel_count_imgr0 <= (others => '0');
                    end if;

                    if valid_pixel_count_imgr1 = total_valid_pixels_imgr1_latched then
                        valid_pixel_count_imgr1 <= (others => '0');
                    end if;
                end if;
            end case;
        end if;
    end if;
end process;

```

```

    if valid_pixel_count_imgr2 = total_valid_pixels_imgr2_latched then
        valid_pixel_count_imgr2 <= (others => '0');
    end if;

    wr_addr_ctrl <= st_wrIdle;
    -- use earlier imgr_span_reg1 value to set Header packet
    elsif imgr_span_reg1 = '1' and lval_rise_reg = '1' and first_line = '1' then
        wr_addr_ctrl <= st_wrHeaderPacket;
        frame_start_flag <= '1';
        latch_en <= '1';
    elsif imgr_span_reg1 = '1' and lval_rise_reg = '1' then
        wr_addr_ctrl <= st_wrHeaderPacket;
        -- set latch_en high one clock cycle early
        latch_en <= '1';
        -- imgr_span_reg2 is one clock cycle earlier than
        -- active_pixel_valid_reg2 (which is used in later states to determine valid
        -- pixel count)
    elsif imgr_span_reg2 = '1' then
        wr_addr_ctrl <= st_wrImgr0;
    end if;

    header_en_buf <= '0';
    imgr_active <= "000";

-----
when st_wrHeaderPacket =>
    -- Check to make sure line is valid:
    -- Use earlier lval_reg1 to make sure the header word gets written correctly.
    -- (no longer used) Later states use lval_reg2 to make sure the last pixels get
    -- written correctly.
    if lval_reg1 = '0' then
        wr_addr_ctrl <= st_wrIdle;
    elsif imgr_span_reg2 = '1' then
        wr_addr_ctrl <= st_wrImgr0;

    --
        -- ground signal one clock cycle early
    end if;

    frame_start_flag <= '0';
    latch_en <= '0';
    header_en_buf <= '1';
    imgr_active <= "000";

-----
when st_wrImgr0 =>
    if valid_pixel_count_imgr0 = total_valid_pixels_imgr0_latched then
        valid_pixel_count_imgr0 <= (others => '0');
    elsif active_pixel_valid_reg2 = '1' then
        valid_pixel_count_imgr0 <= valid_pixel_count_imgr0 + 1;
    end if;

    latch_en <= '0';
    imgr_active <= "001";
    header_en_buf <= '0';
    wr_addr_ctrl <= st_wrImgr1;

-----
when st_wrImgr1 =>
    if valid_pixel_count_imgr1 = total_valid_pixels_imgr1_latched then

```

```

        valid_pixel_count_imgr1 <= (others => '0');
    elsif active_pixel_valid_reg2 = '1' then
        valid_pixel_count_imgr1 <= valid_pixel_count_imgr1 + 1;
    end if;

    latch_en <= '0';
    imgr_active <= "010";
    header_en_buf <= '0';
    wr_addr_ctrl <= st_wrlmgr2;

-----

when st_wrlmgr2 =>
    if valid_pixel_count_imgr2 = total_valid_pixels_imgr2_latched then
        valid_pixel_count_imgr2 <= (others => '0');
    elsif active_pixel_valid_reg2 = '1' then
        valid_pixel_count_imgr2 <= valid_pixel_count_imgr2 + 1;
    end if;

    latch_en <= '0';
    imgr_active <= "100";
    header_en_buf <= '0';
    wr_addr_ctrl <= st_wrlidle;

-----

    end case;
end if;
end process;

-- Instantiate the two BRAMs
buf_0 : burst_pixel_buf
port map (
    clka => Clock      ,
    dina => data_in_0  ,
    addra => wr_addr   ,
    wea  => write_en_0 ,
    clkb => Clock      ,
    addrb => rd_addr   ,
    doutb => data_out_0 );

buf_1 : burst_pixel_buf
port map (
    clka => Clock      ,
    dina => data_in_1  ,
    addra => wr_addr   ,
    wea  => write_en_1 ,
    clkb => Clock      ,
    addrb => rd_addr   ,
    doutb => data_out_1 );

=====
-- Read Control Logic --
=====

process(Reset, Clock)
begin
    if Reset = '1' then
        rd_addr <= (others => '0');
    elsif rising_edge(Clock) then
        if ReadEn = '1' then
            if rd_addr >= overall_valid_pixels_latched then

```

```

        rd_addr <= (others => '0');
    else
        rd_addr <= rd_addr + 1;
    end if;
else
    rd_addr <= (others => '0');
end if;
end if;
end process;

process(Reset, Clock)
begin
    if Reset = '1' then
        rd_done <= '0';
    elsif rising_edge(Clock) then
        if (rd_addr = (overall_valid_pixels_latched - 1)) then
            rd_done <= '1';
        else
            rd_done <= '0';
        end if;
    end if;
end if;
end process;

rd_buf_sel <= not wr_buf_sel;

process(Reset, Clock)
begin
    if Reset = '1' then
        ReadData <= (others => '0');
    elsif rising_edge(Clock) then
        if rd_buf_sel = "0" then
            ReadData <= data_out_0;
        elsif rd_buf_sel = "1" then
            ReadData <= data_out_1;
        end if;
    end if;
end if;
end process;

end Behavioral;

```

A.4 read_fov_buf.vhd

The following VHDL code is the implementation of the *Read Buffer* module described in Section 4.3.4.

```

-- This is the Read Buffer module. It controls the read out
-- of the pixel data from the BRAMs in the "Buffer" module.
-- This is instantiated in the overall system top module.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if instantiating

```

```

---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity read_fov_buf is
generic (
    VERSION          : std_logic_vector(15 downto 0) := x"DEAD"
);
Port (
    Reset           : in  std_logic;  -- async reset
    Clock           : in  std_logic;  -- system clock

    FrHeaderRdAddr  : out std_logic_vector( 5 downto 0);
    FrHeaderRdData  : in  std_logic_vector(11 downto 0);
    ImHeaderRdAddr  : out std_logic_vector( 5 downto 0);
    ImHeaderRdData  : in  std_logic_vector(11 downto 0);
    CameraId        : in  std_logic_vector( 1 downto 0);
    QuadId          : in  std_logic_vector( 1 downto 0);

    imgr_sample_en  : in  std_logic;  -- sample incoming data and framing signals from imager (
        most likely falling-edge of imager master clock)

    -- interface with foveation_top.vhd
    wr_done_pulse   : in  std_logic;
    rd_done         : in  std_logic;
    first_line_active_start : in std_logic;
    first_imgr_sel  : in  std_logic_vector(3 downto 0);
    ReadData        : in  std_logic_vector(15 downto 0);
    ReadEn          : out std_logic;

    -- to Rocket I/O
    riodata         : out std_logic_vector(15 downto 0);
    riovalid        : out std_logic
);

end read_fov_buf;

architecture Behavioral of read_fov_buf is

    component pixel_header_buf -- 32 words deep, 12 bit words, single clock
    port (
        clk: IN std_logic;
        din: IN std_logic_VECTOR(15 downto 0);
        rd_en: IN std_logic;
        rst: IN std_logic;
        wr_en: IN std_logic;
        almost_empty: OUT std_logic;
        dout: OUT std_logic_VECTOR(15 downto 0);
        empty: OUT std_logic;
        full: OUT std_logic;
        valid: OUT std_logic);
    END component;

    constant HEADER_INFO          : std_logic_vector( 3 downto 0) := x"D" ;  -- to indicate start
        of metadata header

    signal rd_done_reg1, rd_done_reg2 : std_logic;
    signal ReadData_reg1, ReadData_reg2 : std_logic_vector(15 downto 0);

    signal rd_done_latch : std_logic;

```

```

signal read_fifo : std_logic;
signal write_fifo , write_fifo_tmp , write_fifo_reg1 , write_fifo_reg2 : std_logic;
signal fifo_empty , fifo_almost_empty , fifo_full , fifo_valid : std_logic;
signal data_from_fifo : std_logic_vector(15 downto 0);

signal end_of_line_flag , end_of_line_flag_reg1 : std_logic;

signal data_from_BRAM : std_logic_vector(15 downto 0);

type control_state_type is (st_cIdle , st_cFrameStartWait , st_cHeader , st_cRead_Fifo ,
    st_cRest_of_Fifo ,
        st_cDelay1 , st_cDelay2 , st_cDelay3 , st_cNormalLineRead);
signal control_state : control_state_type;

signal header_en , header_en_reg : std_logic;
signal imgr_hdr_en : std_logic;

signal header_prefix : std_logic_vector(3 downto 0);
signal header_data_reg , header_data_reg2 : std_logic_vector(11 downto 0);

signal riovalid_tmp , riovalid_reg1 , riovalid_reg2 : std_logic;
signal riodata_int : std_logic_vector(15 downto 0);

signal header_count , header_count_reg : std_logic_vector(4 downto 0);
signal fifo_readout_en , fifo_readout_en_reg : std_logic;

begin

-- taken from ddr_read_ctrl.vhd
FrHeaderRdAddr <= '0' & header_count;
ImHeaderRdAddr <= first_imgr_sel & header_count(1 downto 0);

fifo_during_header_i : pixel_header_buf
port map (
    clk => Clock , --IN
    din => ReadData, --image_data_reg , --IN
    rd_en => read_fifo , --IN
    rst => Reset , --IN
    wr_en => write_fifo , --header_en , --IN
    almost_empty => fifo_almost_empty , --: OUT std_logic;
    dout => data_from_fifo , --OUT
    empty => fifo_empty , --OUT
    full => fifo_full , --OUT
    valid => fifo_valid --OUT
);

process(Clock , Reset)
begin
    if (Reset = '1') then
        rd_done_reg1 <= '0';

        end_of_line_flag_reg1 <= '0';
    elsif rising_edge(Clock) then
        rd_done_reg1 <= rd_done;
        rd_done_reg2 <= rd_done_reg1;

        fifo_readout_en_reg <= fifo_readout_en;

        header_count_reg <= header_count;
        header_en_reg <= header_en;

```

```

    ReadData_reg1 <= ReadData;
    ReadData_reg2 <= ReadData_reg1;

    end_of_line_flag_reg1 <= end_of_line_flag;

end if;
end process;

process(Clock, Reset)
begin
    if (Reset = '1') then
        write_fifo_reg1 <= '0';
        write_fifo_reg2 <= '0';
        write_fifo <= '0';
    elsif rising_edge(Clock) then
        write_fifo_reg1 <= write_fifo_tmp;
        write_fifo_reg2 <= write_fifo_reg1;
        write_fifo <= write_fifo_reg2;
    end if;
end process;

rd_done_latch <= '0' when (header_en = '0') else
    '1' when ( rd.done = '1' and header_en = '1') else
    rd_done_latch;

process(Clock, Reset)
begin
    if (Reset = '1') then
        fifo_readout_en <= '0';
        header_count <= (others => '0');
        header_en <= '0';
        read_fifo <= '0';
        write_fifo_tmp <= '0';
        riovalid_tmp <= '0';
        ReadEn <= '0';
        control_state <= st_cIdle;
        end_of_line_flag <= '0';
    elsif rising_edge(Clock) then
        case control_state is
            -----
            when st_cIdle =>
                if (first_line_active_start = '1') then
                    control_state <= st_cFrameStartWait;
                elsif (wr_done_pulse = '1') then
                    control_state <= st_cDelay1;
                else
                    control_state <= st_cIdle;
                end if;

                ReadEn <= '0';
                fifo_readout_en <= '0';
                header_count <= (others => '0');
                header_en <= '0';
                read_fifo <= '0';
                write_fifo_tmp <= '0';
                riovalid_tmp <= '0';
                end_of_line_flag <= '0';
            -----
            when st_cFrameStartWait =>
                if (wr_done_pulse = '1') then

```

```

        control_state <= st_cHeader;
    else
        control_state <= st_cFrameStartWait;
    end if;

    ReadEn <= '0';
    fifo_readout_en <= '0';
    header_count <= (others => '0');
    header_en <= '0';
    read_fifo <= '0';
    write_fifo_tmp <= '0';
    riovalid_tmp <= '0';
    end_of_line_flag <= '0';

```

```

when st_cHeader =>
    -- State transition control:
    -- Use header_count to determine how long to stay in this state,
    -- which should be for 32 clock cycles
    if (header_count = "11111") then
        control_state <= st_cRead_Fifo;
        -- add a start of frame flag here to indicate to start reading out from FIFO?
    else
        control_state <= st_cHeader;
    end if;

    if (rd_done_latch = '1') then
        write_fifo_tmp <= '0'; -- no more pixels to buffer
    else
        write_fifo_tmp <= '1'; -- buffer pixels while outputting metadata info (headers
        )
    end if;

    ReadEn <= '1';
    header_count <= header_count + 1;
    header_en <= '1';
    read_fifo <= '0';
    riovalid_tmp <= '1';
    end_of_line_flag <= '0';

```

```

-- keep reading and writing to fifo for the first line.
when st_cRead_Fifo =>
    if (rd_done_latch = '1') then
        write_fifo_tmp <= '0';
        read_fifo <= '1';
        control_state <= st_cRest_of_Fifo;
    elsif (rd_done = '1') then
        write_fifo_tmp <= '0'; -- stop writing to Fifo
        read_fifo <= '1';
        control_state <= st_cRest_of_Fifo;
    else
        write_fifo_tmp <= '1'; -- continue writing to fifo for first line of valid
        pixels
        read_fifo <= '1'; -- read from fifo
        control_state <= st_cRead_Fifo;
    end if;

    ReadEn <= '1';
    fifo_readout_en <= '1';
    header_en <= '0';
    riovalid_tmp <= '1';

```



```

end_of_line_flag <= '0';

-----
-- Read out the remaining pixels in the fifo for the first line.
-- At most, 32 more pixels to buffer after line ends
when st_cRest_of_Fifo =>
  if (fifo_almost_empty = '1') then
    control_state <= st_cIdle;
    read_fifo <= '0';
    riovalid_tmp <= '0';
    end_of_line_flag <= '1';
  else
    control_state <= st_cRest_of_Fifo;
    read_fifo <= '1';
    riovalid_tmp <= '1';
    end_of_line_flag <= '0';
  end if;
  ReadEn <= '0';
  fifo_readout_en <= '1';
  header_en <= '0';
  write_fifo_tmp <= '0';      -- stop writing to FIFOs

-----

when st_cDelay1 =>
  control_state <= st_cDelay2;

  ReadEn <= '1';
  fifo_readout_en <= '0';
  header_en <= '0';
  read_fifo <= '0';
  write_fifo_tmp <= '0';
  riovalid_tmp <= '0';
  end_of_line_flag <= '0';

when st_cDelay2 =>
  control_state <= st_cDelay3;

  ReadEn <= '1';
  fifo_readout_en <= '0';
  header_en <= '0';
  read_fifo <= '0';
  write_fifo_tmp <= '0';
  riovalid_tmp <= '0';
  end_of_line_flag <= '0';

when st_cDelay3 =>
  control_state <= st_cNormalLineRead;

  ReadEn <= '1';
  fifo_readout_en <= '0';
  header_en <= '0';
  read_fifo <= '0';
  write_fifo_tmp <= '0';
  riovalid_tmp <= '0';
  end_of_line_flag <= '0';

-----

when st_cNormalLineRead =>
  if (rd_done_reg2 = '1') then
    control_state <= st_cIdle;
    end_of_line_flag <= '1';
  else

```

```

        control_state <= st_cNormalLineRead;
        end_of_line_flag <= '0';
    end if;

    ReadEn <= '1';
    fifo_readout_en <= '0';
    header_en <= '0';
    read_fifo <= '0';
    write_fifo_tmp <= '0';
    riovalid_tmp <= '1';

    end case;
end if;
end process;

-- set flag to read from imager ram for last 4 pixels
process(Clock, Reset)
begin
    if (Reset = '1') then
        imgr_hdr_en <= '0';
    elsif rising_Edge(Clock) then
        if header_en = '0' then --reset
            imgr_hdr_en <= '0';
        elsif header_count_reg = "11010" then -- set for last 4 pixels
            imgr_hdr_en <= '1';
        end if;
    end if;
end process;

-- select which Header data to read: ImHeaderRdData or FrHeaderRdData or none.
process(Clock, Reset)
begin
    if (Reset = '1') then
        header_data_reg <= (others => '0');
        header_data_reg2 <= (others => '0');
    elsif rising_Edge(Clock) then
        if header_en = '0' then -- 1st pixel
            header_data_reg <= CameraId & QuadId & first_imgr_sel & x"0";
        elsif imgr_hdr_en = '1' then
            header_data_reg <= ImHeaderRdData;
        else
            header_data_reg <= FrHeaderRdData;
        end if;
        header_data_reg2 <= header_data_reg;
    end if;
end process;

process(Clock)
begin
    if rising_edge(Clock) then
        if header_en = '1' then
            if header_count_reg = "00000" then
                header_prefix <= HEADER.INFO;
            else
                header_prefix <= x"0";
            end if;
        else
            header_prefix <= (others => '0');
        end if;
    end if;
end process;

```

```

process(Clock, Reset)
begin
    if (Reset = '1') then
        data.from.BRAM <= (others => '0');
    elsif rising_edge(Clock) then
        data.from.BRAM <= ReadData;
    end if;
end process;

-- output final riodata to mgt
process(Clock, Reset)
begin
    if (Reset = '1') then
        riodata <= (others => '0');
        riodata_int <= (others => '0');
    elsif rising_edge(Clock) then
        if header.en_reg = '1' then
            riodata_int <= header_prefix & header_data_reg2;
        elsif (fifo_readout_en_reg = '1') then
            riodata_int <= data_from_fifo;
        else
            riodata_int <= data_from_BRAM;
        end if;
        riodata <= riodata_int;
    end if;
end process;

-- output final riovalid to mgt
process(Reset, Clock)
begin
    if Reset = '1' then
        riovalid <= '0';
        riovalid_reg1 <= '0';
        riovalid_reg2 <= '0';
    elsif rising_edge(Clock) then
        riovalid_reg1 <= riovalid_tmp;
        riovalid_reg2 <= riovalid_reg1;
        riovalid <= riovalid_reg2;
    end if;
end process;

end Behavioral;

```

A.5 quad_selector.vhd

The following VHDL code is the implementation of the *Quadrant Selector* module described in Section 4.3.5.

```

-- This is the Quadrant Selector module which decides whether to send to the MGTs
-- pixel data from this quadrant or an external quadrant. This is instantiated
-- in the overall system top module.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

```

```

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

entity quad_selector is
  Port (
    Reset    : in  std_logic;
    Clock    : in  std_logic;
    usr_rst  : in  std_logic;
    usrclk   : in  std_logic;

    data_from_internal    : in  std_logic_vector(15 downto 0);
    valid_from_internal  : in  std_logic;

    data_from_external    : in  std_logic_vector(15 downto 0);
    charisk_from_external : in  std_logic_vector(1 downto 0);

    -- to mgt_block.vhd
    sel_data_out    : out std_logic_vector(15 downto 0);
    sel_data_valid  : out std_logic
  );
end quad_selector;

architecture Behavioral of quad_selector is

  -- component declaration for FIFO with Common Clock
  component data_from_quad_fifo
  port (
    clk: IN std_logic;
    din: IN std_logic_vector(15 downto 0);
    rd_en: IN std_logic;
    rst: IN std_logic;
    wr_en: IN std_logic;
    dout: OUT std_logic_vector(15 downto 0);
    empty: OUT std_logic;
    full: OUT std_logic;
    valid : out std_logic);
  end component;

  -- component declaration for FIFO with Independent Clocks
  component data_from_ext_fifo IS
  port (
    din: IN std_logic_VECTOR(15 downto 0);
    rd_clk: IN std_logic;
    rd_en: IN std_logic;
    rst: IN std_logic;
    wr_clk: IN std_logic;
    wr_en: IN std_logic;
    dout: OUT std_logic_VECTOR(15 downto 0);
    empty: OUT std_logic;
    full: OUT std_logic;
    valid: OUT std_logic);
  END component;

  constant END_OF_LINE    : std_logic_vector( 3 downto 0) := x"4"  ;

  type fifo_rd_ctrl_state_type is (st_rdIdle, st_rdStartInt, st_rdStopInt, st_rdStartExt,
    st_rdStopExt);
  signal fifo_rd_ctrl_state : fifo_rd_ctrl_state_type;

```

```

signal int_wr_en_fifo , int_rd_en_fifo : std_logic;
signal int_fifo_data_out : std_logic_vector(15 downto 0);
signal int_fifo_empty , int_fifo_full , int_fifo_valid : std_logic;

signal ext_wr_en_fifo , ext_rd_en_fifo : std_logic;
signal ext_fifo_data_out : std_logic_vector(15 downto 0);
signal ext_fifo_empty , ext_fifo_full , ext_fifo_valid : std_logic;

signal data_from_internal_reg , data_from_external_reg : std_logic_vector(15 downto 0);

signal int_data_prefix , ext_data_prefix : std_logic_vector(3 downto 0);

signal rd_stop_flag : std_logic;

signal data_sel : std_logic_vector(1 downto 0);

begin
-----==== { FIFO Instantiation } =====
-- There are two FIFOs in this module. The "int" FIFO is a Common Clock FIFO,
-- and is used to buffer the data coming from this quadrant.
-- The "ext" FIFO has Independent Clocks, and is used
-- to buffer data coming from the external quadrants. The corresponding
-- enable/valid signals are labeled accordingly: int_XXX_XXX or ext_XXX_XXX.

-- FIFO to buffer data coming from this quadrant.
-- Note: This FIFO is a Common Clock FIFO.
int_quad_fifo_i : data_from_quad_fifo
port map (
    clk => Clock ,
    rst => Reset ,
    wr_en => int_wr_en_fifo ,
    din => data_from_internal_reg ,
    rd_en => int_rd_en_fifo ,
    dout => int_fifo_data_out ,
    empty => int_fifo_empty ,
    full => int_fifo_full ,
    valid => int_fifo_valid
);

-- FIFO to buffer data coming from the external quadrant.
-- Note: This FIFO has Independent Clocks. Used to change
-- the clock domains of the data coming from external quads.
ext_quad_fifo_i : data_from_ext_fifo
port map (
    rst => Reset ,
    wr_clk => usrclk ,
    wr_en => ext_wr_en_fifo ,
    din => data_from_external_reg ,
    rd_clk => Clock ,
    rd_en => ext_rd_en_fifo ,
    dout => ext_fifo_data_out ,
    empty => ext_fifo_empty ,
    full => ext_fifo_full ,
    valid => ext_fifo_valid
);

-----==== { Signal Delays } =====
-- Delay the data before buffering into the FIFO.
process(Reset , Clock)
begin

```

```

    if (Reset = '1') then
        data_from_internal_reg <= (others => '0');
    elsif rising_edge(Clock) then
        data_from_internal_reg <= data_from_internal;
    end if;
end process;

process(usr_rst , usrclk)
begin
    if (usr_rst = '1') then
        data_from_external_reg <= (others => '0');
    elsif rising_edge(usrclk) then
        data_from_external_reg <= data_from_external;
    end if;
end process;

-----==== { Write FIFO signals } =====
-- Determine the write enable signals
process(Reset , Clock)
begin
    if (Reset = '1') then
        int_wr_en_fifo <= '0';
    elsif rising_edge(Clock) then
        int_wr_en_fifo <= valid_from_internal;
    end if;
end process;

process(usr_rst , usrclk)
begin
    if (usr_rst = '1') then
        ext_wr_en_fifo <= '0';
    elsif rising_edge(usrclk) then
        if (charisk_from_external = "00") then
            ext_wr_en_fifo <= '1';
        else
            ext_wr_en_fifo <= '0';
        end if;
    end if;
end process;

-----==== { Read FIFO Signals } =====
-- Determine the read enable signals
int_data_prefix <= (others => '0') when int_fifo_valid = '0'
    else int_fifo_data_out(15 downto 12);

ext_data_prefix <= (others => '0') when ext_fifo_valid = '0'
    else ext_fifo_data_out(15 downto 12);

int_rd_en_fifo <= '0' when ( int_data_prefix = END_OF_LINE ) else
    '0' when ( rd_stop_flag = '1' ) else
    '1' when ( (int_fifo_empty = '0') and (data_sel = "01") ) else
    '0';

ext_rd_en_fifo <= '0' when ( ext_data_prefix = END_OF_LINE ) else
    '0' when ( rd_stop_flag = '1' ) else
    '1' when ( (ext_fifo_empty = '0') and (data_sel = "10") ) else
    '0';

-----==== { State Machine } =====
-- State machine to determine the data_sel signal, which is the mux selection that
-- determines which FIFO (int or ext) to read from.

```

```

-- Will read out an entire (buffered) line of a quadrant before switching to the other FIFO.
-- Also, sets the valid signal rd_stop_flag, which is used to control the read enable signals.
process(Reset, Clock)
begin
  if (Reset = '1') then
    rd_stop_flag <= '0';
    data_sel <= "00";
    fifo_rd_ctrl.state <= st_rdIdle;
  elsif rising_edge(Clock) then

    case fifo_rd_ctrl.state is
      -----
    when st_rdIdle =>
      -- State transition
      -- Preference is to read from the int FIFO first.
      if (int_fifo_empty = '0') then
        fifo_rd_ctrl.state <= st_rdStartInt;
      elsif (ext_fifo_empty = '0') then
        fifo_rd_ctrl.state <= st_rdStartExt;
      end if;

      rd_stop_flag <= '0';
      data_sel <= "00";

      -----
    when st_rdStartInt =>
      -- State transition
      -- Also, the pulse rd_stop_flag is asserted one clock cycle early, so that
      -- it goes high with the state transition.
      if (int_data_prefix = END_OF_LINE) then
        fifo_rd_ctrl.state <= st_rdStopInt;
        rd_stop_flag <= '1';
      else
        rd_stop_flag <= '0';
      end if;

      data_sel <= "01";

      -----
    when st_rdStopInt =>

      if (ext_fifo_empty = '0') then
        fifo_rd_ctrl.state <= st_rdStartExt;
      else
        fifo_rd_ctrl.state <= st_rdIdle;
      end if;

      rd_stop_flag <= '0';
      data_sel <= "00";

      -----
    when st_rdStartExt =>
      -- State transition
      -- Also, the pulse rd_stop_flag is asserted one clock cycle early, so that
      -- it goes high with the state transition.
      if (ext_data_prefix = END_OF_LINE) then
        fifo_rd_ctrl.state <= st_rdStopExt;
        rd_stop_flag <= '1';
      else
        rd_stop_flag <= '0';
      end if;
    end case;
  end if;
end process;

```

```

data_sel <= "10";

-----
when st_rdStopExt =>

    if (int_fifo_empty = '0') then
        fifo_rd_ctrl_state <= st_rdStartInt;
    else
        fifo_rd_ctrl_state <= st_rdIdle;
    end if;

    rd_stop_flag <= '0';
    data_sel <= "00";

end case;
end if;
end process;

-- Select which FIFO's data to output (int or ext).
process(Clock, Reset)
begin
    if Reset = '1' then
        sel_data_out    <= (others => '0');
        sel_data_valid <= '0';
    elsif rising_edge(Clock) then

        case data_sel is

            when "00" =>
                sel_data_out    <= (others => '0');
                sel_data_valid <= '0';

            when "01" =>
                sel_data_out    <= int_fifo_data_out;
                sel_data_valid <= int_fifo_valid;

            when "10" =>
                sel_data_out    <= ext_fifo_data_out;
                sel_data_valid <= ext_fifo_valid;

            when others =>
                sel_data_out    <= (others => '0');
                sel_data_valid <= '0';

        end case;
    end if;
end process;

end Behavioral;

```


Appendix B

VHDL Test Benches

B.1 `tb_rectangles_new.vhd`

The following VHDL test bench is the simulation described in Section 4.4.1.

```
-- This is the test bench to test the Rectangles module.
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY tb_rectangles_new.vhd IS
END tb_rectangles_new.vhd;

ARCHITECTURE behavior OF tb_rectangles_new.vhd IS

    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT rectangles
    PORT(
        Reset : IN std_logic;
        Clock : IN std_logic;
        imgr_sample_en : IN std_logic;
        vid_reg : IN std_logic_vector(11 downto 0);
        lval : IN std_logic;
        fval : IN std_logic;
        lval_fall : IN std_logic;
        pixel_count : IN std_logic_vector(11 downto 0);
        line_count : IN std_logic_vector(11 downto 0);
        start_x : IN std_logic_vector(11 downto 0);
        start_y : IN std_logic_vector(11 downto 0);
        total_valid_pixels : IN std_logic_vector(10 downto 0);
        total_valid_lines : in std_logic_vector(10 downto 0);
        column_skip_mode : IN std_logic_vector(7 downto 0);
        row_skip_mode : IN std_logic_vector(7 downto 0);
        frame_active_start : OUT std_logic;
        first_line : OUT std_logic;
        line_active_out : OUT std_logic;
        pixel_valid : OUT std_logic;
        pixel_data : OUT std_logic_vector(11 downto 0)
    );
```

```

END COMPONENT;

constant SAMPLEPOINT : std_logic_vector(1 downto 0) := "01";

--Inputs
SIGNAL Reset : std_logic := '0';
SIGNAL Clock : std_logic := '0';
SIGNAL imgr_sample_en : std_logic := '0';
SIGNAL lval_master_reg : std_logic := '0';
SIGNAL fval_master_reg : std_logic := '0';
SIGNAL master_lval_fall : std_logic := '0';
SIGNAL vid_reg : std_logic_vector(11 downto 0) := (others='0');
SIGNAL pixel_count : std_logic_vector(11 downto 0) := (others='0');
SIGNAL line_count : std_logic_vector(11 downto 0) := (others='0');
SIGNAL start_x : std_logic_vector(11 downto 0) := (others='0');
SIGNAL start_y : std_logic_vector(11 downto 0) := (others='0');
SIGNAL total_valid_pixels : std_logic_vector(10 downto 0) := (others='0');
signal total_valid_lines : std_logic_vector(10 downto 0) := (others='0');
SIGNAL column_skip_mode : std_logic_vector(7 downto 0) := (others='0');
SIGNAL row_skip_mode : std_logic_vector(7 downto 0) := (others='0');

--Outputs
-- SIGNAL line_start : std_logic;
SIGNAL frame_active_start : std_logic;
SIGNAL first_line : std_logic;
SIGNAL line_active_out : std_logic;
SIGNAL pixel_valid : std_logic;
SIGNAL pixel_data : std_logic_vector(11 downto 0);

signal fval_master_int : std_logic := '0';
signal fval_master : std_logic := '0';
signal lval_master_int : std_logic := '0';
signal lval_master : std_logic := '0';

signal master_fval_rise : std_logic := '0';
signal master_fval_fall : std_logic := '0';
signal master_lval_rise : std_logic := '0';

signal next_pixel_count, next_line_count : std_logic_vector(11 downto 0);

SIGNAL data_reg : std_logic_vector(131 downto 0) := (others='0');
signal vid_reg_int : std_logic_vector(11 downto 0) := (others => '0');

signal TP_Framevalid, TP_LineValid : std_logic;
signal TP_Video : std_logic_vector(11 downto 0);

signal clkdiv_counter : std_logic_vector(1 downto 0);
signal imgr_CE : std_logic;

BEGIN

Reset <= '1', '0' after 145 ns;
Clock <= not Clock after 5 ns; --100 mhz

-- Instantiate the Unit Under Test (UUT)
ut: rectangles PORT MAP(
    Reset => Reset,
    Clock => Clock,
    imgr_sample_en => imgr_sample_en,
    vid_reg => vid_reg,

```

```

        lval => lval_master_reg ,
        fval => fval_master_reg ,
        lval_fall => master_lval_fall ,
        pixel_count => pixel_count ,
        line_count => line_count ,
        start_x => x"004" ,
        start_y => x"002" ,
        total_valid_pixels => "000" & x"0" & x"8" , -- total_valid_pixels ,
total_valid_lines => "000" & x"0" & x"8" ,
        column_skip_mode => x"02" ,
        row_skip_mode => x"02" ,

--      line_start => line_start ,
        frame_active_start => frame_active_start ,
        first_line => first_line ,
        line_active_out => line_active_out ,
        pixel_valid => pixel_valid ,
        pixel_data => pixel_data
    );

-- This is the module that simulates the behavior of an imager.
-- It simulates pixel data and frame valid and line valid signals
imgr_TestPattern_i : entity work.imgr_TestPattern
generic map(
    BUS_WIDTH => 12 ,
    ACTIVE_ROWS => x"020" , -- good for simulation
    ACTIVE_COLS => x"020" ,
    TOTAL_ROWS => x"040" ,
    TOTAL_COLS => x"040"
)
port map(
    Reset => Reset ,
    Clock => Clock ,

    ImagerClockEn => imgr_sample_en , -- use as a clock enable at the master clock rate...

--
    Enable => '1' , -- '0' = Pass Through, '1' = overwrite with image data
    ModeSelect => '0' , -- '0' = external timing, '1' = internal timing

-- Data Out module
    FrameValidOut => TP_Framevalid ,
    LineValidOut => TP_LineValid ,
    DataOut => TP_Video ,

-- output to foveation_top.vhd for simulation
    fval_master => fval_master ,
    lval_master => lval_master ,
    master_fval_rise => master_fval_rise ,
    master_fval_fall => master_fval_fall ,
    master_lval_rise => master_lval_rise ,
    master_lval_fall => master_lval_fall ,
    data_reg => data_reg
);

-- delay an imgr_sample_en cycle to align with
-- input rise/fall signals from imgr_ctrl.vhd.
process(Reset, Clock)
begin
    if (Reset = '1') then
        fval_master_int <= '0';

```

```

    lval_master_int <= '0';

    fval_master_reg <= '0';
    lval_master_reg <= '0';
elseif rising_edge(Clock) then
    if (imgr_sample_en = '1') then
        fval_master_int <= fval_master;
        lval_master_int <= lval_master;

        fval_master_reg <= fval_master_int;
        lval_master_reg <= lval_master_int;

        vid_reg_int <= data_reg(11 downto 0);
        vid_reg <= vid_reg_int;
    end if;
end if;
end process;

pix_line_count:
process(Reset, Clock)
begin
    if (Reset = '1') then
        pixel_count <= (others => '0');
        line_count <= (others => '0');
    elseif rising_edge(Clock) then
        if (imgr_sample_en = '1') then
            pixel_count <= next_pixel_count;
            line_count <= next_line_count;
        end if;
    end if;
end process;

-- next state of pixel count and line count is computed with combinational logic
next_pixel_count <= (others => '0') when (lval_master_reg = '0') else
    (pixel_count + 1);

next_line_count <= (others => '0') when (fval_master_reg = '0') else
    (line_count + 1) when (master_lval_fall = '1') else
    line_count;

imgr_clock :
process(Reset, Clock )
begin
    if Reset = '1' then
        clkdiv_counter <= "00";
    elseif rising_edge(Clock) then
        clkdiv_counter <= clkdiv_counter + '1' ;
    end if;
end process;

process(Reset, Clock )
begin
    if (Reset = '1' ) then
        imgr_CE <= '0';
    elseif rising_edge(Clock) then
        if clkdiv_counter = SAMPLEPOINT then
            imgr_CE <= '1';
        else
            imgr_CE <= '0';
        end if;
    end if;
end process;

```

```

    end process;
    imgr_sample_en <= imgr.CE;
END;

```

B.2 tb_foveation_top.vhd

The following VHDL test bench is the simulation described in Section 4.4.2.

```

-- This is the test bench to test the "Foveation Top Module" as described in
-- the thesis.
-- Simulations were done in Modelsim
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY tb_foveation_top_vhd IS
END tb_foveation_top_vhd;

ARCHITECTURE behavior OF tb_foveation_top_vhd IS

    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT foveation_top
    PORT(
        Reset : IN std_logic;
        Clock : IN std_logic;
        imgr_sample_en : IN std_logic;
        fval_master : IN std_logic;
        lval_master : IN std_logic;
        fval_rise : IN std_logic;
        fval_fall : IN std_logic;
        lval_rise : IN std_logic;
        lval_fall : IN std_logic;
        vid_all : IN std_logic_vector(131 downto 0);
        start_x_all : in std_logic_vector (131 downto 0);
        start_y_all : in std_logic_vector (131 downto 0);
        total_valid_pixels_all : in std_logic_vector(120 downto 0);
        total_valid_lines_all : in std_logic_vector(120 downto 0);
        column_skip_mode : IN std_logic_vector(7 downto 0);
        row_skip_mode : IN std_logic_vector(7 downto 0);

        ReadEn : in std_logic;
        wr_done_pulse : out std_logic;
        rd_done : out std_logic;
        first_imgr_sel : out std_logic_vector(3 downto 0);
        first_line_active_start : out std_logic;
        ReadData : out std_logic_vector(15 downto 0)
    );
    END COMPONENT;

    constant SAMPLE_POINT : std_logic_vector(1 downto 0) := "01";

    -- Inputs
    SIGNAL Reset : std_logic := '0';
    SIGNAL Clock : std_logic := '0';
    SIGNAL imgr_sample_en : std_logic := '0';
    SIGNAL fval_master : std_logic := '0';

```

```

    SIGNAL lval_master : std_logic := '0';
    SIGNAL master_fval_rise : std_logic := '0';
    SIGNAL master_fval_fall : std_logic := '0';
    SIGNAL master_lval_rise : std_logic := '0';
    SIGNAL master_lval_fall : std_logic := '0';
    SIGNAL data_reg : std_logic_vector(131 downto 0) := (others=>'0');
    SIGNAL start_x_all : std_logic_vector(131 downto 0) := (others=>'0');
    SIGNAL start_y_all : std_logic_vector(131 downto 0) := (others=>'0');
    signal total_valid_pixels_all : std_logic_vector(120 downto 0);
    signal total_valid_lines_all : std_logic_vector(120 downto 0);
    SIGNAL column_skip_mode : std_logic_vector(7 downto 0);--(43 downto 0) := (others=>'0');
    SIGNAL row_skip_mode : std_logic_vector(7 downto 0);--(43 downto 0) := (others=>'0');
    signal ReadEn : std_logic := '0';

    --Outputs
    SIGNAL frame_count_out : std_logic_vector(11 downto 0);
    SIGNAL line_start : std_logic;
    SIGNAL frame_start : std_logic;
    SIGNAL fval_fov : std_logic;
    SIGNAL lval_fov : std_logic;
    SIGNAL pixel_valid_fov : std_logic;
    SIGNAL pixel_data_fov : std_logic_vector(11 downto 0);
    signal wr_done_pulse : std_logic;
    signal rd_done : std_logic;
    signal first_imgr_sel : std_logic_vector(3 downto 0);
    signal first_line_active_start : std_logic;
    signal ReadData : std_logic_vector(15 downto 0);

    signal TP_Framevalid, TP_LineValid : std_logic;
    signal TP_Video : std_logic_vector(11 downto 0);

    signal clkdiv_counter : std_logic_vector(1 downto 0);
    signal imgr_CE : std_logic;

```

BEGIN

```

    Reset <= '1', '0' after 145 ns;
    Clock <= not Clock after 5 ns; --100 mhz

    -- for sim purposes
    process(Clock, Reset)
    begin
        if Reset = '1' then
            ReadEn <= '0';
        elsif rising_edge(Clock) then
            if wr_done_pulse = '1' then
                ReadEn <= '1';
            end if;
        end if;
    end process;

    -- Instantiate the Unit Under Test (UUT)
    uut: foveation_top PORT MAP(
        Reset => Reset,
        Clock => Clock,
        imgr_sample_en => imgr_sample_en,
        fval_master => fval_master,
        lval_master => lval_master,
        fval_rise => master_fval_rise,
        fval_fall => master_fval_fall,

```



```

        elsif rising_edge(Clock) then
            clkdiv_counter <= clkdiv_counter + '1' ;
        end if;
    end process;

    process(Reset, Clock)
    begin
        if (Reset = '1') then
            imgr_CE <= '0';
        elsif rising_edge(Clock) then
            if clkdiv_counter = SAMPLEPOINT then
                imgr_CE <= '1';
            else
                imgr_CE <= '0';
            end if;
        end if;
    end process;
    imgr_sample_en <= imgr_CE;
END;

```

B.3 tb_read_fov_buf.vhd

The following VHDL test bench is the simulation described in Section 4.4.3.

```

-- This is the test bench to test the "Read Buffer" module.
-- Tests were run under ModelSim.
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY tb_read_fov_buf_vhd IS
END tb_read_fov_buf_vhd;

ARCHITECTURE behavior OF tb_read_fov_buf_vhd IS

    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT read_fov_buf
    PORT(
        Reset : IN std_logic;
        Clock : IN std_logic;
        FrHeaderRdData : IN std_logic_vector(11 downto 0);
        ImHeaderRdData : IN std_logic_vector(11 downto 0);
        CameraId : IN std_logic_vector(1 downto 0);
        QuadId : IN std_logic_vector(1 downto 0);
        imgr_sample_en : IN std_logic;
        wr_done_pulse : IN std_logic;
        rd_done : IN std_logic;
        first_line_active_start : IN std_logic;
        first_imgr_sel : in std_logic_vector(3 downto 0);
        ReadData : IN std_logic_vector(15 downto 0);
        FrHeaderRdAddr : OUT std_logic_vector(5 downto 0);
        ImHeaderRdAddr : OUT std_logic_vector(5 downto 0);
        ReadEn : OUT std_logic;
        riodata : OUT std_logic_vector(15 downto 0);
        riovalid : OUT std_logic
    );

```



```

END COMPONENT;

component header_ram
  port (
    a      : IN  std_logic_VECTOR(5 downto 0);
    d      : IN  std_logic_VECTOR(11 downto 0);
    dpra   : IN  std_logic_VECTOR(5 downto 0);
    clk    : IN  std_logic;
    we     : IN  std_logic;
    spo    : OUT std_logic_VECTOR(11 downto 0);
    dpo    : OUT std_logic_VECTOR(11 downto 0));
  END component;

constant SAMPLE_POINT : std_logic_vector(1 downto 0) := "01";

  --Inputs
  SIGNAL Reset : std_logic := '0';
  SIGNAL Clock : std_logic := '0';
  SIGNAL imgr.sample_en : std_logic := '0';
  -- SIGNAL fval_master : std_logic := '0';
  -- SIGNAL lval_master : std_logic := '0';
  SIGNAL wr_done_pulse : std_logic := '0';
  SIGNAL rd_done : std_logic := '0';
  SIGNAL first_line_active_start : std_logic := '0';
  signal first_imgr_sel : std_logic_vector(3 downto 0);
  SIGNAL FrHeaderRdData : std_logic_vector(11 downto 0) := (others=>'0');
  SIGNAL ImHeaderRdData : std_logic_vector(11 downto 0) := (others=>'0');
  SIGNAL CameraId : std_logic_vector(1 downto 0) := (others=>'0');
  SIGNAL QuadId : std_logic_vector(1 downto 0) := (others=>'0');
  SIGNAL ReadData : std_logic_vector(15 downto 0) := (others=>'0');

  --Outputs
  SIGNAL FrHeaderRdAddr : std_logic_vector(5 downto 0);
  SIGNAL ImHeaderRdAddr : std_logic_vector(5 downto 0);
  SIGNAL ReadEn : std_logic;
  SIGNAL riodata : std_logic_vector(15 downto 0);
  SIGNAL riovalid : std_logic;

  --internal signals
  signal fval_master : std_logic := '0';
  signal lval_master : std_logic := '0';
  signal master.fval_rise : std_logic := '0';
  signal master.fval_fall : std_logic := '0';
  signal master.lval_rise : std_logic := '0';
  signal master.lval_fall : std_logic := '0';
  signal data_reg : std_logic_vector(131 downto 0) := (others=>'0');

  signal TP_Framevalid, TP_LineValid : std_logic;
  signal TP_Video : std_logic_vector(11 downto 0);

  signal clkdiv_counter : std_logic_vector(1 downto 0);
  signal imgr_CE : std_logic;

  signal FrHeaderWrAddr : std_logic_vector(5 downto 0);
  signal FrHeaderWrData : std_logic_vector(11 downto 0);
  signal FrHeaderWrEn : std_logic;

  signal ImHeaderWrAddr : std_logic_vector(5 downto 0);
  signal ImHeaderWrData : std_logic_vector(11 downto 0);
  signal ImHeaderWrEn : std_logic;

```

BEGIN

```
Reset <= '1', '0' after 145 ns;
Clock <= not Clock after 5 ns;  --100 mhz

-- Instantiate the Unit Under Test (UUT)
ut: read_fov_buf PORT MAP(
    Reset => Reset ,
    Clock => Clock ,
    FrHeaderRdAddr => FrHeaderRdAddr ,
    FrHeaderRdData => FrHeaderRdData ,
    ImHeaderRdAddr => ImHeaderRdAddr ,
    ImHeaderRdData => ImHeaderRdData ,
    CameraId => CameraId ,
    QuadId => QuadId ,
    imgr_sample.en => imgr_sample.en ,
    wr_done.pulse => wr_done.pulse ,
    rd_done => rd.done ,
    first_line.active.start => first_line_active_start ,
    first_imgr_sel => first_imgr_sel , --: in std_logic_vector(3 downto 0);
    ReadData => ReadData ,
    ReadEn => ReadEn ,
    riodata => riodata ,
    riovalid => riovalid
);

foveation_top.i : entity work.foveation_top PORT MAP(
    Reset => Reset ,
    Clock => Clock ,
    imgr_sample.en => imgr_sample.en ,
    fval_master => fval_master ,
    lval_master => lval_master ,
    fval_rise => master.fval_rise ,
    fval_fall => master.fval_fall ,
    lval_rise => master.lval_rise ,
    lval_fall => master.lval_fall ,
    vid_all => data_reg ,
    start_x_all => x"FFF00000000000000000000000000000",
    start_y_all => x"FFF00000000000000000000000000000",

    total.valid_pixels_all => "
        0000000000000000000000000000000000000000000000000000000000000000" & "
        000000000000" & "00000001000" & "00000001000" & "00000001000" ,
    total.valid_lines_all => "
        0000000000000000000000000000000000000000000000000000000000000000"
        & "00000000000" & "00000001000" & "00000001000" & "00000001000" ,
    column_skip_mode => x"04" ,
    row_skip_mode => x"02" ,

    ReadEn => ReadEn , --: in std_logic;
    wr_done.pulse => wr_done.pulse , --: out std_logic;
    rd_done => rd.done , --: out std_logic;
    first_imgr_sel => first_imgr_sel , --: out std_logic;
    first_line.active.start => first_line_active_start,--: out std_logic;
    ReadData => ReadData --: out std_logic_vector(15 downto 0)
);

-- This is the module that simulates the behavior of an imager.
-- It simulates pixel data and frame valid and line valid signals
imgr_TestPattern.i : entity work.imgr_TestPattern
generic map(
```

```

    BUS_WIDTH => 12 ,
    ACTIVE_ROWS => x"020" , -- good for simulation
    ACTIVE_COLS => x"020" ,
    TOTAL_ROWS => x"040" ,
    TOTAL_COLS => x"040"
)
port map(
    Reset => Reset ,
    Clock => Clock ,

    ImagerClockEn => imgr_sample_en , -- use as a clock enable at the master clock rate...

    --
    Enable      => '1' , -- '0' = Pass Through, '1' = overwrite with image data
    ModeSelect  => '0' , -- '0' = external timing, '1' = internal timing

    -- Data Out module
    FrameValidOut => TP.Framevalid ,
    LineValidOut  => TP.LineValid ,
    DataOut       => TP.Video ,

    -- output to foveation_top.vhd for simulation
    fval_master => fval_master ,
    lval_master => lval_master ,
    master_fval_rise => master_fval_rise ,
    master_fval_fall => master_fval_fall ,
    master_lval_rise => master_lval_rise ,
    master_lval_fall => master_lval_fall ,
    data_reg => data_reg
);

imgr_clock :
process(Reset, Clock )
begin
    if Reset = '1' then
        clkdiv_counter <= "00";
    elsif rising_edge(Clock) then
        clkdiv_counter <= clkdiv_counter + '1' ;
    end if;
end process;

process(Reset, Clock )
begin
    if (Reset = '1' ) then
        imgr_CE <= '0';
    elsif rising_edge(Clock) then
        if clkdiv_counter = SAMPLE_POINT then
            imgr_CE <= '1';
        else
            imgr_CE <= '0';
        end if;
    end if;
end process;

imgr_sample_en <= imgr_CE;

-- write the header with random information (written by Tom Karolyshyn of LL)

frheader_ram_i : header_ram
port map(
    a      => FrHeaderWrAddr ,

```

```

    d    => FrHeaderWrData ,
    clk  => Clock          ,
    we   => FrHeaderWrEn  ,
    spo  => open           ,
    dpra => FrHeaderRdAddr ,
    dpo  => FrHeaderRdData
);

imheader_ram_i : header_ram
port map(
    a    => ImHeaderWrAddr ,
    d    => ImHeaderWrData ,
    clk  => Clock          ,
    we   => ImHeaderWrEn  ,
    spo  => open           ,
    dpra => ImHeaderRdAddr ,
    dpo  => ImHeaderRdData
);

process
begin
    FrHeaderWrAddr <= "000000";
    FrHeaderWrData <= x"000";
    FrHeaderWrEn   <= '0';
    wait for 1 us;
    wait until rising_edge(Clock);
    for ii in 0 to 31 loop
        wait until rising_edge(Clock);
        FrHeaderWrEn   <= '1';
        wait until rising_edge(Clock);
        FrHeaderWrEn   <= '0';
        FrHeaderWrAddr <= FrHeaderWrAddr + '1';
        wait until rising_edge(Clock);
        FrHeaderWrData <= FrHeaderWrAddr & FrHeaderWrAddr;
        wait until rising_edge(Clock);
    end loop;
    wait;
end process;

process
begin
    ImHeaderWrAddr <= "000000";
    ImHeaderWrData <= x"000";
    ImHeaderWrEn   <= '0';
    wait for 1 us;
    wait until rising_edge(Clock);
    for ii in 0 to 31 loop
        wait until rising_edge(Clock);
        ImHeaderWrEn   <= '1';
        wait until rising_edge(Clock);
        ImHeaderWrEn   <= '0';
        ImHeaderWrAddr <= ImHeaderWrAddr + '1';
        wait until rising_edge(Clock);
        ImHeaderWrData <= ImHeaderWrAddr & ImHeaderWrAddr ;
        wait until rising_edge(Clock);
    end loop;
    wait;
end process;
END;

```