Low Power Image Based Triggering for Extended Operation Surveillance

by

Matthew C. Waldon

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

June 2004

Author .....
Department of Electrical Engineering and Computer Science
May 19, 2004

Certified by .....
Lawrence M. Candell
MIT Lincoln Laboratory
Thesis Supervisor

Certified by .....
Dennis M. Freeman
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Low Power Image Based Triggering for Extended Operation Surveillance

by

Matthew C. Waldon

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

It is desirable for many surveillance applications to have a portable high quality imaging system capable of continuous monitoring in remote locations, often for extended periods of time. Extended operation can be achieved with low power by taking advantage of the fact that no interesting action is occurring in the area of interest most of the time, allowing the camera to be turned off. This type of operation requires some type of trigger to detect when events occur and turn on the camera to collect imagery. A novel technique for this type of detection is the use of signal processing on low spatial and temporal resolution imagery with a low-power processor to detect action events. The low-resolution imager operation and low-power processor allow the system to consume minimal power, while still taking advantage of the information available from the imager. Triggering is done by performing background subtraction on the low resolution imagery to detect scene changes. Although there is extensive research on this subject, no known research has attempted to implement this type of algorithm in a low power system, which puts a significant constraint on the computation that can be performed. This paper describes research conducted at MIT Lincoln Laboratory to develop a power constrained background subtraction technique, and design a low power hardware system that utilizes this form of detection for image based triggering.

Thesis Supervisor: Lawrence M. Candell
Title: MIT Lincoln Laboratory

Thesis Supervisor: Dennis M. Freeman
Title: Associate Professor

Acknowledgments

I would like to thank all of the staff at MIT Lincoln Laboratory that have helped me on many different projects through the last three years, and most recently on this thesis. I would specifically like to thank Mark Beattie, Larry Candell, Bill Ross, Chris Ferraiolo, Pete Lafauci, Glenn Callahan, and Jim Sopchack. I would also like to thank my MIT thesis and academic advisor Professor Dennis Freeman for his support and guidance over the last four years at MIT. Finally I would like to thank my parents, without whom none of this would have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Purpose

It is desirable in many surveillance applications to have a portable high quality imaging system capable of continuous monitoring in remote locations, often for extended periods of time. This requires that the system be small, lightweight, and consume very little electrical power in order to allow operation on batteries. Traditional, "always on", high quality imaging systems consume a significant amount of power in both the imaging sensor and control electronics, preventing their use for this application.

A low power system can be achieved by exploiting the fact that in many remote surveillance applications, nothing is happening in the area of interest most of the time. Average power consumption can be significantly reduced by turning the camera off during these periods, only turning it on to collect imagery when something interesting happens. For this type of system to be successful, a trigger is needed to detect such events, and turn the camera on. Triggering has been done in the past using external sensors such as acoustic, seismic, or single diode "fence" triggers that detect an obstruction to a line of sight.

There are several problems that arise from using external sensors to trigger camera operation. These include placement difficulty, risk, and synchronization of the event with image capture. Many of these problems are solved using a Through The Lens Trigger (TTLT). In this type of system, a trigger is determined by processing real-time image data to identify interesting movement in the scene. The problem with this is that running the camera continuously undermines the strategy of turning the camera

off during uninteresting periods to conserve power; however, continuous operation can be low power if the imager is run at low spatial and temporal resolutions with a fast readout. This allows image data to be quickly extracted from the imager, which is then turned off or put into a standby mode until the next frame is to be collected. This type of operation combined with low-power control and processing electronics allows the system to consume minimal power while running the image-based trigger.

The low power constraint on the TTLT processing electronics requires that the detection algorithm be implemented with a small amount of digital logic. The result is an algorithm that is relatively simple, and not very robust against dynamic backgrounds. As a result, the TTLT by itself would not be a very useful system, as battery life would be mostly eaten up by false triggers. The solution to this problem is a layered detection approach, whereby the power constrained TTLT algorithm can be dynamically taught by a more robust detection algorithm using higher power electronics that are only turned on when a trigger occurs. This method of layered functionality with adaptive feedback provides greatly improved performance against dynamic backgrounds with the TTLT. This approach also provides an additional level of detection processing before a lot of power is drained in image transmission.

Research is currently being conducted at MIT Lincoln Laboratory to develop a low power camera surveillance system based on the power constrained TTLT with layered functionality and adaptive feedback. This paper describes the design and implementation of the TTLT hardware and algorithms, as well as an overview of the camera system as a whole.

Chapter 2

Background

Before the details of the Lincoln Laboratory camera are described, it is valuable to examine existing research that is related to the technology used in this system. There is currently no published research on the topic of low power operation using through the lens detection; however, research has been done related to low power operation and through the lens detection taken separately. These research topics provide useful insights into techniques that this system builds on, and the difficulties in combining the two areas.

2.1 Low Power Camera Operation

Low power camera operation has been a popular area of recent research for both commercial and military applications. The recent boom in digital technologies has opened up an extensive commercial market for digital cameras, most of which require some level of low power for operation on batteries. In the military domain, the ever increasing need for surveillance information acquired with minimal human intervention has led to fielded systems targeting the same type of low power application as the Lincoln Laboratory camera.

Although the application of commercial cameras and the Lincoln Laboratory camera are very different, they have the related hardware need of keeping power consumption as low as possible. The rapidly expanding commercial imaging market has led to

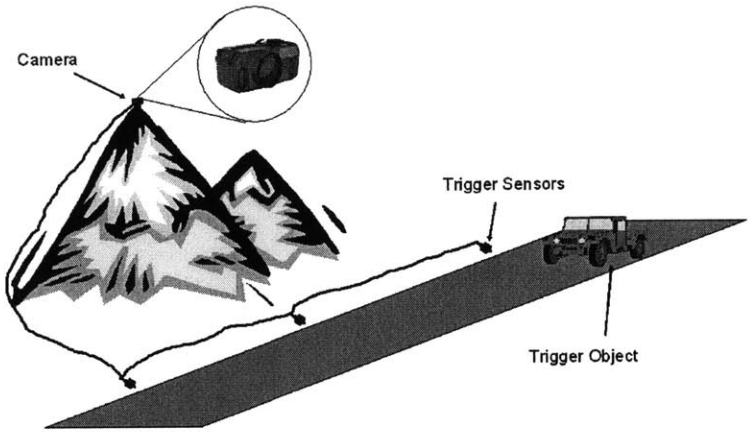

Figure 2-1: Example of Existing Trigger Based Camera System

several advances in low power imaging technology [1]. The most significant of these advances is the emergence of Complementary Metal Oxide Semiconductor (CMOS) imagers, which minimize power and space requirements by incorporating almost all the analog and digital camera electronics into a single chip. This type of device offers significant savings for any imaging application that requires low power [3].

The United States military currently has camera systems placed in the field for the same type of extended surveillance application as the Lincoln Laboratory camera. These systems use an external trigger concept, whereby the camera is left off most of the time to conserve power, and only turned on when some type of external sensor detects action in the area. Sensors used include acoustic, seismic, and single diode "fence" triggers. An example of this type of system is shown in Figure 2-1. In the system shown, a camera is placed at the top of a hill pointing down at a road. Seismic sensors are placed next to the road to detect when vehicles drive by. These sensors could be tethered to the camera either by wires or through a radio link. When a vehicle drives by, the seismic sensors would detect the vibration in the road and trigger the camera to turn on and take pictures.

There are several problems that result from this type of operation. First, these types of sensors have very limited ranges, forcing them to be placed close to areas in which targets are expected to occur. This can be difficult, especially when the area of observation is large. A second problem with placing external sensors is one of

risk. In many military applications the area of interest is hostile, making it difficult and risky to place sensors. The final problem that arises from using external sensors is synchronization. In many cameras there is a time delay between when an image sensor is powered on, and when imagery can be collected. The delay can result in the object that caused the trigger not being imaged. It is these shortfalls that prompted research into using Through The Lens Triggering to achieve low power operation.

2.2 Through the Lens Detection

Through the lens detection solves many of the problems with external trigger operation discussed in the previous section. The form of through the lens detection most relevant to this application is background subtraction. The goal of background subtraction is to distinguish between background objects that exist in the image for extended periods, and foreground objects which are only temporarily part of the image scene.

A large amount of research has been done in recent years in the area of background subtraction. Topics in this area are largely focused on theories of computer vision, without regard to strict implementation constraints. As a result, the theories and algorithms developed are implemented in high-power computer workstations, without regard to power consumption. The application of these theories and methods to the extremely power constrained processing in the Lincoln Laboratory camera will be discussed in Chapter 4.

Most of the current research in background subtraction involves using an adaptive statistical representation of the background on a pixel by pixel basis. The details of the statistical model vary, but most methods model each pixel in the background by a mixture of Gaussian distributions. The simplest of these models is a single Gaussian per pixel. This type of model taken alone is only valid under the assumption of a static scene, where the distribution represents a mean intensity value corrupted with sensor noise [10].

For the single Gaussian per pixel model, the intensity mean can be obtained

through temporal averaging. A common method for this type of averaging is a running pixel mean, based on a learning coefficient, α [4],[12]. In this method, the pixel mean at frame n, μ_n, is the combination of μ_{n-1}, and the intensity at frame n, I_n. The weighting of these two values is determined by α. The result is

$$\mu_n = (1 - \alpha) \cdot \mu_{n-1} + \alpha \cdot I_n \tag{2.1}$$

The variance about this mean in a static scene is the image sensor noise.

Using this model, any pixel showing an intensity value outside its background distribution can be considered to contain foreground information. In the real world, however, the assumption of a completely static background is almost always violated. Global illumination changes and recurring background motion are common phenomena, especially in outdoor environments. An expanded description of real world phenomenon affecting model performance is given in [11].

To deal with the complexity of real world scenes, current research is focused on using algorithms much more complex than a single Gaussian [2],[4],[10]. One of the most commonly used variations is a weighted mixture of Gaussians representing each pixel. These models are built by observing recurring pixel intensities, and representing each one by a Gaussian distribution. The weighting of the Gaussians is based on the relative number of occurrences that fall into each distribution. Input pixel intensities are compared with all the Gaussians in their respective mixture model to check if they are part of any recurring trends. Pixels that are not part of a recorded trend are detected as foreground. This method accounts for multi-modal pixel intensities, such as a pixel that sees a road at some times, and a tree branch at others. A broad description of this approach is given in [10]. This type of modeling has shown promising results in a variety of indoor and outdoor scenes.

So far the discussion has focused primarily on pixel by pixel approaches to background modeling. In addition to individual pixel history, there is also information in the contextual relationships between pixels. These relationships form the basis for image segmentation and object tracking, where a priori object knowledge is used

to identify foreground shapes, and track their direction based on movements across several frames. A method using high-level knowledge that is fed to pixel by pixel algorithms to improve their performance is described in [7]. In this algorithm there is both positive and negative feedback, corresponding to reinforcing positive detections, and suppressing false detections, respectively.

In addition to contextual knowledge about inter-pixel relationships, there is also information contained in the mathematical correlations between pixels in a local region. One such approach is presented in [9]. In this method, each frame is broken into blocks, and a normalized correlation of the intensities in each block is calculated. During a training period, the variation in these correlations are used to create a probability distribution of the correlation values for each block. Any block that exhibits a correlation outside an acceptable range based on the distribution is considered foreground. Using this type of normalized correlation makes the detection algorithm independent of global illumination changes.

2.3 Discussion

This chapter has provided a survey of the existing research in the fields of low power camera operation and through the lens detection based on background subtraction. The goal of this camera system is to combine these two areas of research, and create a low power system based on through the lens triggering. This requires the development of a simple, yet robust, detection algorithm to allow computation in a low power device. In addition, low power hardware must be designed to support the algorithm, and meet the high quality imaging requirements of the application. The development of the hardware will be discussed in Chapter 3, and the development of the power constrained detection algorithm in Chapter 4.

Chapter 3

System Architecture

The purpose of this camera system is to continuously monitor an area over an extended period of time in order to collect high resolution imagery when events of interest occur. As discussed previously, the extended mission requirement prevents the use of traditional "always on" camera systems due to their high power consumption. This chapter describes the system architecture of the Lincoln Laboratory camera, and the way it achieves minimal power consumption while maintaining the desired functionality.

3.1 Principle of Operation

3.1.1 Overview

Before a detailed examination of the low power architecture of this camera can be done, a refinement of the operational scenario is necessary to be more specific about the restrictions imposed by its application. The root restriction is that the system must be portable, requiring that it be small and lightweight. The result is that all the power must be provided be a few small batteries. From this power alone, the camera must be capable of "extended operation," which for this purpose is defined as running with full functionality, without any human intervention, for a period of approximately one month.

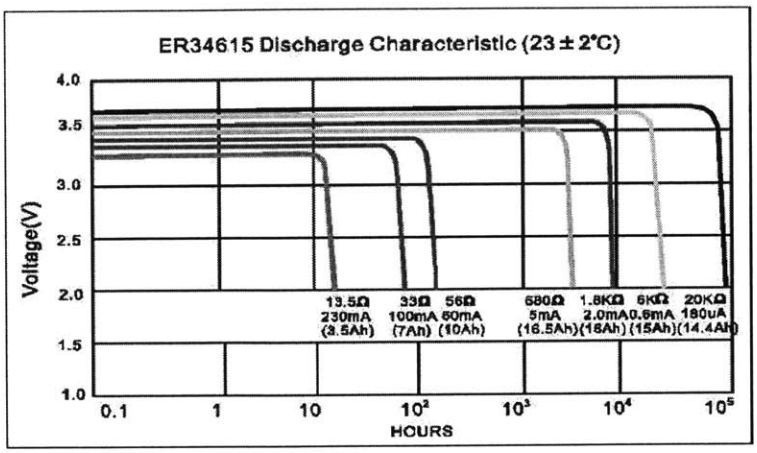

Figure 3-1: Battery Discharge Characteristic [5]

Extended operation is achieved through low power, which is best defined by a specific example. The discharge characteristics of one of the top of the line small batteries on the market is shown in Figure 3-1. From this chart, it can be determined that this battery can provide on the order of 35,000 mW/hours of battery life, depending on the rate of power drain. Assuming a power conversion efficiency of 75%, there are approximately 26,000 mW/hours of usable power per battery, or 105,000 mW/hours with four batteries. Therefore, if the system is to operate for 30 days, or 720 hours, it must have a time average power consumption on the order of 150mW.

Traditional "always on" camera systems typically consume approximately 1000mW. Thus, a significant reduction in power from traditional camera systems is necessary to meet the requirements for this type of application. In existing systems, this kind of power reduction is achieved by turning the camera off most of the time, triggering it to turn on with external sensors designed to detect the desired type of trigger event. Several problems arise from using external sensor triggering, as discussed in Chapter 2.

Through The Lens Triggering solves many of the problems involved with external sensor triggering, but has the disadvantage that it requires the camera to be on all the time, going against the principle of reducing power through triggered operation. Because full camera functionality is not necessary to compute the TTLT, the cam-

era can be run in a low resolution, low frame rate mode, which does not provide imagery sufficient for the end application, but does provide imagery good enough to detect motion. Detecting motion from this imagery with extremely power constrained processing is a non-trivial task, and the subject of Chapter 4. Low frame rate and resolution operation, coupled with appropriate processing, results in TTLT operation on the order of 75 mW, as will be shown in this chapter.

Although 75 mW is well under the 150 mW time average power budget, there are many other functions the camera must perform once the TTLT has determined an event. These functions include more power intensive processing, image compression, and image transmission. These topics will be touched on briefly, as they are part of the overall Lincoln Laboratory system, but arc not the main focus this paper.

3.1.2 Layered Functionality

The functionality beyond the TTLT requires hardware that would far exceed the power budget if it were powered all the time. To solve this problem, a layered functionality approach is taken, where the high powered hardware is only turned on a small fraction of the time, significantly reducing the time-average power consumption.

There are three layers, or modes, that make up the layered approach. The first mode is the TTLT, in which the camera operates most of the time. This will be referred to as the Trigger mode. If the TTLT detects an event, the second layer of hardware is turned on, referred to as the Collection mode. In this mode the image sensor is run at full resolution and frame rate to collect high quality images. A higher power processor is turned on to process the high quality imagery and better determine if an event of interest was in fact captured. If that determination is positive, a third layer of hardware is turned on to transmit the imagery. This is the Transmission mode.

The reduction in power consumption achieved by the layered approach depends on the power consumption and time spent in each layer. Power consumption in each layer is a function of the system hardware. Table 3.1 shows the power consumed by

Parameter	Trigger	Collection	Transmission
Total Power	74mW	764mW	2698mW
Time/Activation	N/A	1 min.	5 min.

Table 3.1: Power Dissipation and Time/Activation of Operational Modes

the system during each layer of operation; these values will be explained in Section 3.3. Time spent in each layer is a function of the processing time per event in each mode, and the number of real and false events detected by the triggering algorithm. Table 3.1 shows the time spent per activation of each layer; these are estimated values, as the DSP and transceiver have not yet been implemented.

The effect of layering on power is well illustrated by an example. A common mode of operation would be observation of an area where three events would occur per day. In addition to these three events, a number of false triggers would occur that is much greater than the number of actual events. For each trigger, the DSP would be turned on, and the high resolution imagery processed. For simplicity, it is assumed that the DSP will correctly identify all three real events, and falsely identify one, resulting in four image transmits. For each of these events, the transceiver is turned on for five minutes to transmit the data. The power dissipation that results from this scenario is calculated by multiplying the percentage of time spent per day in each mode by the corresponding power consumption, which is dependent on the number of false triggers. The result of this calculation is shown by the lower line in Figure 3-2. From this line, it is shown that layered operation can meet the 150 mW power requirement under the given scenario if there are fewer than 75 false triggers per day. The upper line represents the power consumption without layering, which does not come close to an acceptable average power.

Before further explanation of the system operation, the hardware architecture will be defined to provide a background for this topic.

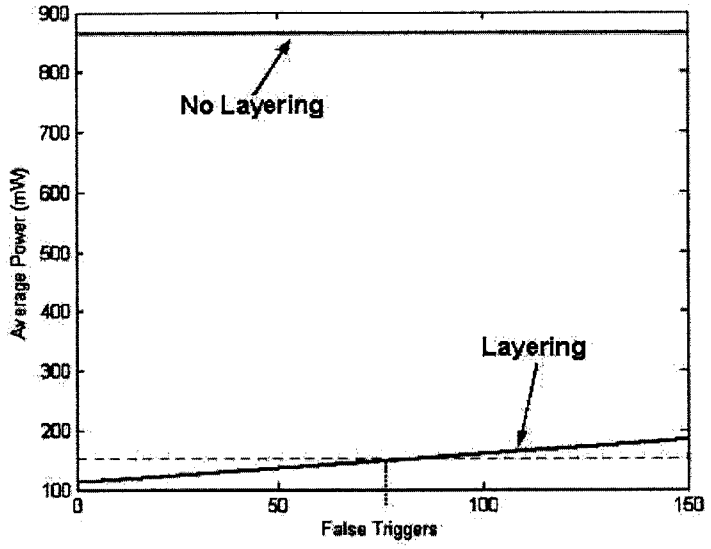

Figure 3-2: Layered Powered Consumption Versus No Layering

3.2 Hardware Architecture

The functionality of the Lincoln Laboratory camera system depends on the components used and the way in which they are connected. In addition to the functionality, the hardware also defines the power characteristics of the system. The hardware architecture of this system is designed to optimize the tradeoff between these two characteristics by using the lowest power hardware that just meets functional needs.

3.2.1 Components

The first step in designing any type of power constrained system is picking parts designed for low power applications. These parts typically offer low nominal power consumption, and power saving features that allow them to be powered down when not in use.

The most fundamental piece of hardware for any camera system is the image sensor. This defines the imaging capabilities, and is typically one of the more power hungry components of the system. The image sensor is controlled and read out by external electronics. In addition to readout electronics, in this system there are also

low power processing electronics to compute the TTLT. The image sensor, control electronics, and processing electronics are the main components of the TTLT. These devices must be extremely low power, as they are on all the time.

Beyond the TTLT, there must be additional processing electronics for more advanced event detection and image compression. These electronics can be higher power, as they are not on for most of the time, but should still consume the minimum power necessary for the task.

The final component is a transceiver. This device is comparatively very high power, and is on for an even shorter amount of time than the advanced processing electronics. There is not a wide range of transceivers available for remote applications, leaving little choice for optimized power and functionality.

The major components of this system will be discussed in detail, with a full parts list given in Appendix A.

Image Sensor. The requirements on the image sensor are the strictest of any of the devices. The first requirement is that the image sensor have low nominal power consumption, and be capable of a low power standby mode. It also must be capable of both low and high resolution readouts. The sensor used is a Rockwell ProCam 2-Megapixel CMOS device. It provides a 12-bit digital output, with several programmable registers to control all aspects of operation. The most important of these programmable registers are several power saving features, and programmable resolution. The sensor runs on the order of 80mW when powered, and 40mW when powered down.

Control and Processing Electronics. The control and processing electronics are implemented with programmable logic and static memory.

Two Xilinx CoolRunnerII CPLDs were chosen for the programmable logic component. These devices are re-programmable, and designed for ultra-low power applications. Nominal power consumption at 15 MHz, with the majority of logic in use, is on the order of 4mW. The tradeoff for such low power is a limited amount of

logic in each device. This puts a major constraint on the complexity of the trigger algorithm. An advantage of these devices is that they have a +1.8V core voltage, with four independently powered I/O banks. This allows the CPLDs to interface to devices of different voltage levels without separate signal level conversion.

In addition to the CPLDs, there are four low power Samsung SRAMs. These are used in the computation of the triggering algorithm, and as a rate buffer to transfer image data to the DSP. These devices are designed for low power applications, with static power dissipation on the order of 80μW, and read/write power dissipation of approximately 6mW.

Digital Signal Processor. A Digital Signal Processor (DSP) is needed for high level image processing, data compression, and transceiver interfacing. This device needs a lot of computational power, with as little power consumption as possible. The device used is a Texas Instruments C5509 low power DSP. This is a 288 MIPS device, on the order of the computational power of an Intel Pentium processor. The power consumption is around 650mW when operating, and 1mW in standby. The standby feature allows the device to remain powered down most of the time, but not completely turned off, such that it can be interrupted and powered up when needed without going through a complete boot cycle.

In addition to the favorable computational and power characteristics of the C5509, the interface characteristics of this device are also ideal for this application. It has an external sixteen bit I/O interface, which can be used for image data transfer with the CPLDs. It has a built-in USB port, a commonly used interface that can be used to connect to a variety of transceivers. There is also a built-in memory card interface, giving it expanded storage capabilities for large data volumes, such as those that result from imaging applications.

Transceiver. A transceiver is used to transmit image data to an operator terminal, as well as receive operator commands. The transceiver used is very application specific, and largely depends on the existing infrastructure available in the area of op-

eration. Possibilities for this device include a cellular phone, satellite phone, or other specialized communication device. Due to the wide range of transceiver possibilities, it is desirable to use a common interface so that the camera can be easily connected to different devices. Universal Serial Bus (USB) was chosen because it is a widely accepted standard with simple connection hardware, and has enough bandwidth to handle the data rate required for an imaging application.

Frame Grabber. The frame grabber interface interface allows image data to be viewed in real-time directly from CPLDs, making it possible to develop the low level algorithm without the DSP. The image data are transmitted to a frame grabber in a computer via LVDS, where the data are reconstructed and displayed as an image. This interface is powered with its own power supply to allow power measurements of the system independent of this development interface.

Power Supplies. The low power demands of this system require that the power system be as efficient as possible so that energy is not wasted in power conversion. The system components have many different power connections, requiring several voltages to be derived from the same battery source. This results in several different power planes, with isolated analog and digital grounds to mitigate the effect of digital noise on the analog image sensor electronics. Each of the positive voltage planes is powered by a separate low dropout regulator, which are used to allow the supply voltage to approach the regulator voltage as much as possible. The regulators are powered by two switching power supplies, one +3.6V, and one +1.8V. These power supplies are fed by the batteries, with approximately 75% efficiency. A summary of the power system is shown in Table 3.2.

3.2.2 Component Connections

The connection of the hardware components is the second determinant, in addition to the components themselves, of system functionality and power consumption. Connection of the components is done in the Lincoln Laboratory camera with a custom

Voltage Plane	Switcher	Devices
+1.6V Digital	+1.8V	DSP
+1.8V Digital	+1.8V	Xilinx Core,Xilinx I/O,SRAM
+2.5V Digital	+3.6V	Imager Digital,Xilinx I/O
+3.3V Digital	+3.6V	DSP
+3.3V Analog	+3.6V	Imager Analog
+3.3V Frame Grabber	N/A	Frame Grabber

Table 3.2: Power System

printed circuit board[1], shown in Figure 3-4. A block diagram showing the connections of all the major hardware components is shown in Figure 3-3.

The first CPLD is essentially the center of control for the camera. Its main function is to control the imager, collect image data, and interface those data to other system components. For this purpose, CPLD 1 has bus connections to all the major devices, as shown in Figure 3-3. The flow of data to each component is determined by the mode of the system, which is controlled by a finite state machine in CPLD 1. This will be discussed in detail in Section 3.4.

The second CPLD is responsible for computation of the TTLT. This requires a connection to CPLD 1 to receive data, and to the memory bank, which is used during motion detection processing. The data bus to CPLD 1 is sixteen bits to allow data communication, as well as control signals used to set motion parameters, and communicate the TTLT status.

Once a trigger event is determined by the TTLT, full resolution data must be sent to the DSP. The DSP's external data interface consists of sixteen data bits, eight address bits, and a number of control signals. This data bus is connected to CPLD 1, and is asynchronous, requiring the use of the memory bank as a rate buffer. The way this is done will be discussed in detail in the next section, but it is important to note that data are not simply fed through CPLD 1 from the imager to the DSP, as occurs when data are sent to CPLD 2 for the TTLT.

The frame grabber is connected to CPLD 1 and CPLD 2 via a common 16 bit

[1]Printed circuit board and schematic designed by Pete Lafaucci and Mike Dovidio at MIT Lincoln Laboratory

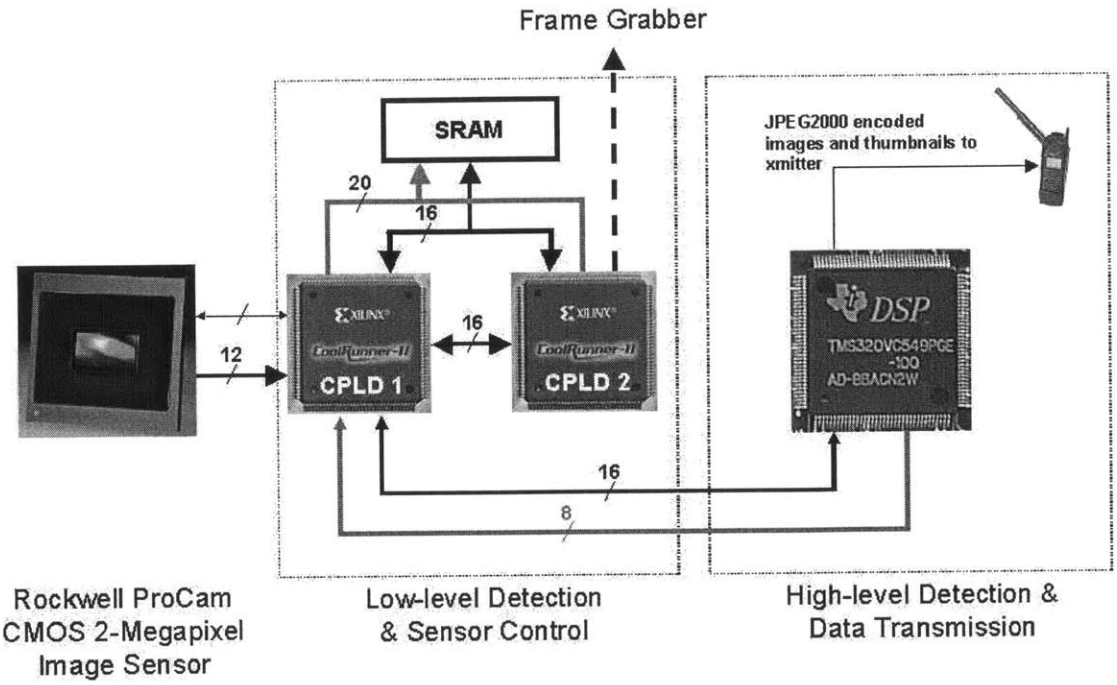

Figure 3-3: Component Connections

data bus. This allows both devices access to the frame grabber, which is very useful for debugging. In addition, using a common bus provides an extra sixteen bits of communication between the CPLDs, as the frame grabber is not part of the final system.

The last major device connection is the Memory bus. The memory bus has sixteen data bits, twenty address bits, four chip selects, and a number of control signals. This bus is shared by both CPLDs, allowing both devices access to the memory bank.

3.3 Hardware Operation

Hardware operation can be broken down into the three modes discussed earlier: Trigger, Collection, and Transmission. Under the layered functionality approach, the hardware that acts as part of the trigger mode is always active, while the collection and transmission hardware is only active when turned on by the preceding layer. Thus, the system operates in a way that keeps trigger mode power at an absolute

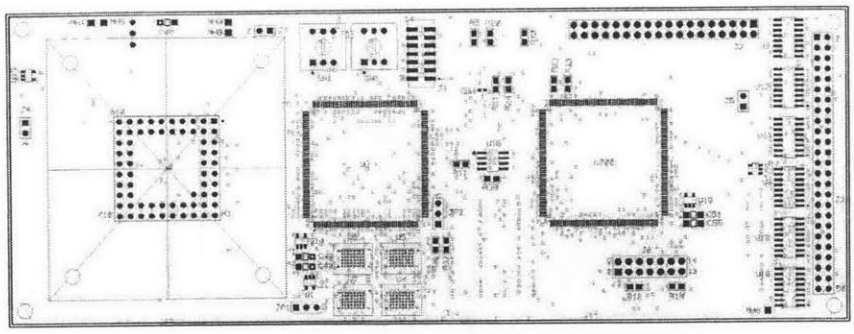

Figure 3-4: Printed Circuit Board

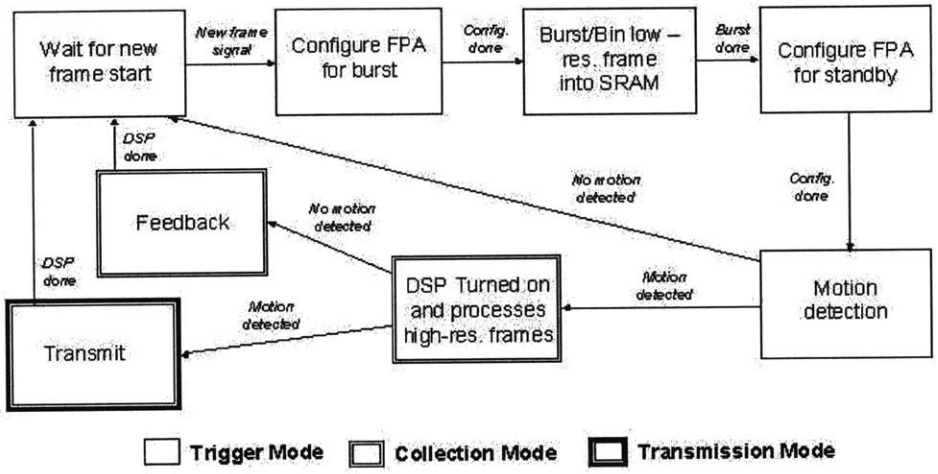

Figure 3-5: Operating Sequence

minimum, and power in the collection and transmission modes as low as possible. A diagram of the hardware operating sequence in shown in Figure 3-5.

3.3.1 Trigger Mode

The two main devices operating in this mode are the image sensor and the control and trigger electronics. The most power intensive of these components is the image sensor; therefore, lowest power is achieved by minimizing the amount of time this device is on.

Time spent with the imager on is minimized because imagery is only needed at a rate of approximately 1 fps, while the imager readout is capable of speeds up to 15

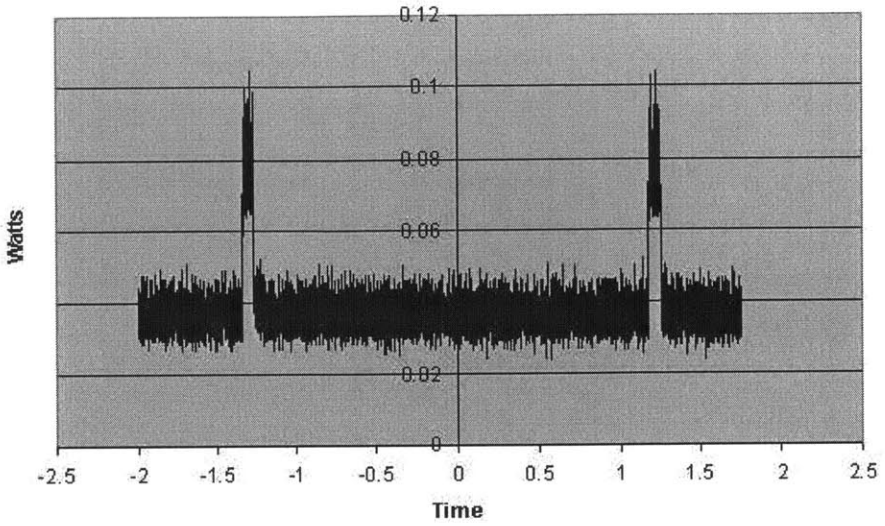

Figure 3-6: Imager Power Consumption in Trigger Mode

fps[2]. The imager is turned on to collect a single frame of data at this frame rate, then powered down for the remainder of the frame period. Thus, during most of the frame period, the imager consumes the standby power, 40 mW, with a spike to 80 mW during image collection. The result is a time average power consumption of 44 mW. A plot showing the image power consumption over a period of two frames is shown in Figure 3-6. This illustrates the power spike to 80 mW during image collection.

The image data are sent to the second CPLD, where the TTLT is performed. If a trigger is detected, the DSP is turned on and the system transitions to the Collection mode. If there is no trigger, the system remains in its low power state until it is time to power up the imager and collect a new frame.

The CPLDs consume approximately 4mW each, and the memories 2mW. It is estimated that a transceiver in a low power listening state would consume approximately 20mW. These values combined with the 44mW imager power result in a total average power dissipation in the trigger mode of 74mW. This result is summarized in Table 3.3.

[2]While the image sensor is capable of 30 fps, memory write speed and circuit board limitations reduce the maximum rate to 15 fps.

3.3.2 Collection Mode

Once a trigger event occurs, the DSP is turned on, and high resolution imagery is immediately collected and sent to the DSP. Once several frames are collected, these data are processed to better determine if the cause of the trigger was an interesting event. The algorithm to make this determination is still under consideration.

If it is determined that the imagery is not worthy of transmission, the DSP sends information back to the TTLT to aid in future false alarm mitigation; this idea of adaptive feedback will be discussed in Chapter 4. Once the information is sent, the DSP is powered down and the camera returns to the trigger mode. If the imagery is determined to be worthy of transmission, the imagery is compressed using the JPEG 2000 standard, and then transmitted in the transmission mode.

Power consumption in the collection mode is mostly driven by the DSP, which dissipates 650mW. The image sensor power consumption is 80mW, and the CPLDs use 4mW each. The memory power increases to 6mW due to the constant use as a frame buffer. The transceiver power is again estimated at 20mW. The combination of all these devices results in an average collection mode power consumption of 764mW. This result is summarized in Table 3.3.

3.3.3 Transmission Mode

The transmission layer is turned on if the DSP determines that a collection of high resolution images contains data that are interesting enough to send. The transmitter is interfaced to the DSP over USB, allowing commands and data to be exchanged. The data to be sent are already compressed using the JPEG 2000 standard when the transmitter is turned on. JPEG 2000 uses multi-resolution compression, whereby wavelet coefficients corresponding to very low resolution are sent first, followed by increasing resolution coefficients until full resolution is achieved. Thus, a low resolution image at the operator terminal is received quickly, allowing selection of a region of interest that can be sent back to the DSP. Once this region of interest is sent back, only those coefficients corresponding to that region are transmitted, making optimal

Component	Trigger	Collection	Transmission
Imager	44 mW	80 mW	40 mW
CPLDs	8 mW	8 mW	8 mW
SRAM	2 mW	6 mW	0 mW
DSP	0mW	650 mW	650 mW
Transmitter	20 mW	20 mW	2000 mW
Total	**74 mW**	**764 mW**	**2698 mW**

Table 3.3: Power Dissipation for Operational Modes

use of the limited transmission bandwidth and reducing the time the transceiver has to be turned on.

Power consumption for the transceiver is largely unknown, and varies depending on the device used. An estimate based on commercially available devices of this type suggests power dissipation on the order of 2000mW. In the transmission mode, the image sensor can be powered down, resulting in dissipation of 40mW. The CPLDs consume 8mW, and the memory power consumption is not significant. The DSP consumes 650mW as in the collection mode. The combination of all the devices results in average power dissipation of 2698mW in the transmission mode. This is summarized in Table 3.3.

3.4 Logic Implementation

System operation and data flow are controlled by programmable logic in CPLD 1. This section describes the logic implemented in this device to accomplish the functionality described in the previous sections. A diagram representing the logic blocks of this device is shown in Figure 3-7. All logic is written in VHDL, and compiled with Xilinx's ISE 5.0 software. The VHDL source code for this device is included in the file *CPLD1.vhd*, included in Appendix B, along with the associated modules.

3.4.1 Primary Controller

The center of control for the camera system is the Primary Controller. It has control over all of the hardware components either directly or indirectly through a connected process. It is implemented with a finite state machine that represents the sequence of operations shown in Figure 3-5. The state determines control signals which route data according to the current mode of operation, and start other processes. State transitions are determined by the current state, process status inputs, and input signals from other devices. The connections of control signals are represented by the thin arrows in figure 3-7.

3.4.2 Data Flow

One of the primary functions of the first CPLD is to control data flow through the different system devices. Data flow can be broken down into two data paths, corresponding to the Trigger and Collection modes; these are shown by the thick lines in Figure 3-7.

The black line in Figure 3-7 represents the data path that is used in the Trigger mode. Image data are sent to the TTLT in real time as they are read out from the sensor, where processing is performed. The background statistics, which are written into the memory bank by the second CPLD during TTLT processing, are read out to the frame grabber for debugging and analysis. The gray line in Figure 3-7 represents the data path used in the Collection Mode. Data coming from the imager are routed to the memory bank, which is used as a frame buffer. Once the data are stored in memory, they are read out and routed to the DSP.

3.4.3 Image Sensor Control

The image sensor has a fairly simple interface because it is such a highly integrated device. This is beneficial in that it greatly simplifies the control logic required to run it, but detrimental in that it makes it difficult to run in the different modes this application requires. The interface consists of three control signals, three register

programming signals, a twelve bit data output bus, and two output control signals.

The image sensor is programmed with a three signal serial interface, consisting of clock, strobe, and data. The programmed data words are sixteen bits long and contain both the register address and data. A register programming module as part of the image sensor control logic takes in 16-bit input words, and programs the imager using the sensor's serial protocol. To begin this process, a *start* signal is sent from the primary controller to the register programming module. The programming sequence sends eight 16-bit data words, which are hard-wired in the CPLD and dependent on the state of the primary controller. Once the programming sequence is complete, the programming module sends a *done* signal back to the primary controller to indicate that the imager is programmed.

The three sensor input control signals are frame, line, and clock. The image sensor has an "integrate while read" architecture, where one frame is read out while the next is being integrated. On the rising edge of the frame signal, integration of the previous frame stops, and readout begins. The start of integration of the next frame is determined by a programmable register that sets the time in rows after the rising edge of the frame signal before the next integration begins. This simple architecture is beneficial for continuous video applications, but in this application it results in an extremely long integration time, as frames are only collected once per second. In order to get around this problem, a "dummy" frame is taken before the actual data frame, such that the data frame follows directly after the dummy frame, resulting in an acceptably short integration time.

The frame signal is determined by an output from the primary controller. As data are read out, the sensor provides line and frame output signals that are synchronized with the readout data. The number and length of image data lines are determined by the programmed resolution. The line pulses are counted to determine when the readout is complete, as no signal indicating this is sent by the imager. Once all the lines in the frame are read out, the image control module sends a *frame done* signal to the primary controller to indicate that image collection is complete.

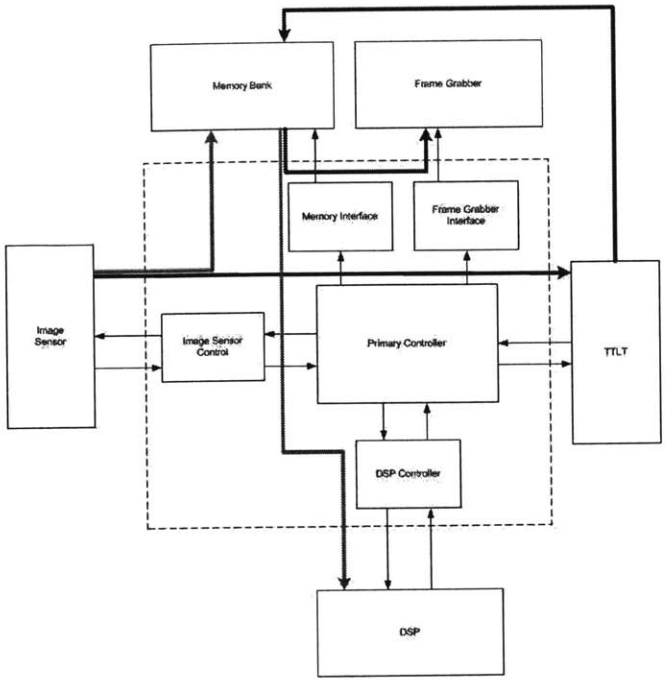

Figure 3-7: CPLD 1 Block Diagram

3.4.4 DSP Controller

The DSP controller is an independent state machine that handles all interfacing to the digital signal processor. Once the primary controller enters the DSP state, all functionality is handed off to the DSP controller. The interface between CPLD 1 and the DSP consists of a 16-bit bidirectional data bus, an 8-bit unidirectional address bus commanded by the DSP, write enable, chip enable, and read enable. Once control is handed off, the DSP is interrupted to power it up out of deep sleep, and the imaging sensor is programmed to collect full resolution imagery. The controller then waits for a command from the DSP.

The DSP is essentially a miniature computer with significant signal processing capabilities. Once this device is powered up, it runs a program to retrieve image data and process them. The DSP interfaces to the system by sending commands to the DSP controller. Commands are sent to the DSP using the address bus of the data interface, with addresses corresponding to specific commands. A command is issued

by pulsing the write enable signal of the interface to the DSP controller while the desired command is valid on the address bus. A description of the DSP commands is given below.

Frame Read The first step in DSP processing is collecting image data. The DSP is completely asynchronous to the CPLD, requiring a rate buffer to transmit data between CPLD 1 and the DSP. In most asynchronous applications similar to this, a FIFO would be used as a rate buffer. FIFOs can be implemented in several different ways, none of which work well for this implementation. This is primarily because data come out on every edge of the CPLD/imager clock, which is also very close to the minimum cycle time of the SRAM. A register based FIFO would require a lot of depth, as data would back up in the FIFO if a pixel were not read on any given cycle. There is not space in the CPLD to accommodate a large FIFO.

The requirements of a FIFO make it impossible to implement with the existing design. Instead of a FIFO, the memory bank is used as a frame buffer, where entire frames are written to memory, then read out by the DSP. Transmission takes on the order of a half a second, during which the memories are being read out, and cannot be written to; therefore, if multiple frames with close temporal spacing are required, all the desired frames must be written into memory before any are read out. In the current hardware, there is enough memory to store three frames if the data are reduced to eight bits per pixel. Once the data are stored in the frame buffer, they must be sent to the DSP. This is done by the data path discussed in Section 3.4.2, where the memory data bus is mapped to the DSP data bus. The DSP address bus is only eight bits wide, which is not wide enough to address the memory. Instead, an internal address counter is kept by CPLD 1, which is advanced through the memory space by the DSP by toggling the read enable signal.

Feedback Write An important part of making the triggering algorithm robust to false alarms is the ability to receive feedback from the DSP processing algorithm. This consists of sending threshold updates for specific pixels to the TTLT. The pixel

address and threshold value are sent to the DSP controller over the data and address buses after the feedback write command. Once data are received by the controller, it writes the updated values into the appropriate memory space for the specified pixel parameter.

DSP Done Once the DSP is finished with processing and data feedback, it must command the DSP controller to send the camera back into trigger mode. Once this command is sent, the DSP is powered down to its sleep state. Control is given back to the primary controller, and the system returns to trigger mode.

3.4.5 Memory Interface

The four Samsung memories have a standard SRAM interface. There are twenty address bits, sixteen data bits, chip select, write enable, and output enable. All the SRAM signals are shared between the two CPLDs, creating a potential bus contention issue if one of the CPLD interfaces is not set to a high impedance state at all times. To be certain contention never occurs, a register bit is set in CPLD 1, and shared with CPLD 2, that specifies which device has control of the bus, and automatically sets the non-controlling interface to high impedance.

3.4.6 Frame Grabber Interface

The frame grabber interface consists of three output control signals, a sixteen bit data bus, and four input serial programming signals. Data are sent to the DSP over the data bus, valid on the rising edge of an output pixel clock. In addition to the clock, frame and line pulses are generated to dictate how the data on the bus should be reconstructed to form an image. The four input signals allow a serial programming interface with data, clock, and strobe. This is used to set algorithm parameters dynamically from the frame grabber computer.

3.5 Testing and Debugging

There were two stages of hardware testing for this system. The first was general hardware testing to ensure that the devices were operating properly on the printed circuit board. The second form of testing was functional, to ensure that the completed system operates as described in Section 3.3.

The scope of this project was to develop the TTLT algorithm and Trigger mode hardware, with the appropriate interfaces for the Collection and Transmission modes. The role of the hardware and algorithms of these modes were discussed to better understand the role of the Trigger functionality, but have only been implemented to the extent that the interface to the first CPLD could be tested.

3.5.1 Device Testing

The first phase of device testing was checking the power to all devices to ensure that all power regulator connections were made and stable. The second step was to test device functionality and I/O. This was done by collecting full frame image data, sending the data to the second CPLD, storing it in memory, and reading it out to the frame grabber through the first CPLD. This data path utilizes all the I/O connections as well as the basic functionality of all the devices.

The second phase of device testing was checking the functionality of the image sensor in the unusual operation required by this application. The slow frame rate, resolution switching, and constant power cycles are not conventional ways of operating an imager. Thus, the results of this type of operation were not specified by the manufacturer. The effects of these operations had to be tested to ensure that the sensor would still function.

The first function tested was the effect of constantly changing the camera resolution. It was discovered from this testing that the frame immediately following the change of resolution was significantly brighter than other frames, using a constant integration time. The solution to this was coupled with the solution to the integration time problem that results from a slow frame rate, discussed in 3.4.3, where an extra

frame is always captured before the desired image frame.

The second function tested was the extent to which the image sensor could be powered down between image frames without a significant effect on the imagery. The first method explored was to turn off the image sensor's power regulator in order to make the sensor power dissipation go to zero. Due to a long settling time on startup, however, this mode of operation produced very poor results. The next method tried was to use the power saving functions of the device that allowed the A/D converters to be powered off, and the bias currents reduced. This proved to reduce power consumption by 50%, without any noticeable effect on image quality.

3.5.2 Operational Debugging and Testing

Once the hardware was tested to ensure proper operation, the next stage of development was to debug the logic in CPLD 1. During this stage of testing, interface signals with CPLD 2 were simulated using test switches, as the logic for CPLD 2 was developed after CPLD 1.

The first step in logic testing and debugging was simulation. The logic synthesized using the VHDL source was simulated in the ModelSim environment. This testing was useful for debugging the state machine sequencing and the resulting control outputs, but full system testing was not possible because of a lack of an HDL description of the camera sensor and SRAM. Several external test points connected to extra CPLD I/O were built into the printed circuit board for the purpose of in-circuit testing. Any CPLD signal can be mapped to these points and monitored on a logic analyzer. This interface was used to complete debugging of the VHDL source code.

3.6 Summary

This chapter has described in detail the system architecture of the Lincoln Laboratory camera. The low power design and operation make the hardware capable of extended operation when associated with an effective TTLT in CPLD 2. The next chapter will describe the theory and implementation of the TTLT.

Chapter 4

Through The Lens Trigger

The success of this camera system is dependent on the ability to run a through the lens trigger while still maintaining low power. This requires a triggering algorithm that is simple enough to be implemented with limited logic, but intelligent enough to be robust against false alarms. Without these two aspects, the system will consume significant power, either from the trigger algorithm, or by constant triggering of the higher power electronics due to false alarms.

4.1 Algorithm

The triggering algorithm is based on the concept of background subtraction, where each new image frame is compared with a stored background in order to detect significant deviations. Algorithms within the scope of background subtraction are differentiated by the method used to collect and maintain the background, and the criteria used to define a significant deviation. Several existing background subtraction algorithms were discussed in section 2.2.

The purpose of this camera system is to continuously monitor remote areas for extended periods of time. It is assumed that most of the time, no significant events are occurring in the camera field of view. The scene remains mostly static, except for changes that might occur due to natural events. The most significant natural events that can cause a trigger to occur are wind blowing around objects in the background,

and illumination changes caused by clouds or movement of the sun. Such events will be referred to as local motion effects and illumination effects, respectively. These types of variations pose the most significant challenge to developing a simple TTLT algorithm.

4.1.1 The Basic Algorithm

The basic algorithm is based on the assumption that the scene is static, with no local motion or illumination effects. Extensions of the basic algorithm to account for these factors will be discussed in section 4.2.

The first part of the algorithm is collecting and maintaining the background model based on a Gaussian distribution with mean μ_t, and standard deviation σ_t. The mean is calculated using a weighted average between μ_{t-1} and I_t, the current pixel intensity. The weighting between these two values are $(1-\alpha)$ and α, respectively, where α is the learning rate. The mean for each frame is then represented by

$$\mu_{t+1} = \alpha \cdot \mu_t + (1-\alpha) \cdot I_t \tag{4.1}$$

This describes the calculation computed each frame for each pixel. Thus, the background at time t, $\beta_t(n_1, n_2)$, is described by equation

$$\beta_{t+1}(n_1, n_2) = \alpha \cdot \beta_t(n_1, n_2) + (1-\alpha) \cdot \mathcal{I}_t(n_1, n_2) \tag{4.2}$$

where $\mathcal{I}_t(n_1, n_2)$ is the current image frame. The standard deviation, σ_t, is approximated by the image sensor noise, σ_{noise}.

4.1.2 Choosing α

The constant α must be determined in order to use equation 4.2 to calculate the background mean. The choice of α determines how quickly the background can adapt to scene changes, as well as how sensitive it is to sensor noise. The variance in intensity due to sensor noise will cause uncertainty in the calculated value of the running mean.

This causes variance in the intensity PDF in addition to σ_{noise}. Although this variance could be accounted for when choosing the detection threshold, α should be made small to keep the distribution as tight as possible. This additional variance can be neglected if it is significantly smaller than the noise variance. In addition to determining the mean variance, α also determines how quickly the system can adapt to changes in the background. Adaptation should be fast to incorporate changes in the background as quickly as possible. This requires a large value of α.

A tradeoff then exists on the choice of α. It should be small to minimize the effect of variance in the mean, and large to allow the background to adapt to changes quickly. Selection of α can be done by examining the relationship between mean variance and settling time. Equation 4.1 is essentially a low-pass filter, with transfer function

$$H_{\mu}(e^{j\omega}) = \frac{\alpha}{1-(1-\alpha)e^{-j\omega}} \tag{4.3}$$

This can be used to determine the response to an input unit step, and input Gaussian process with variance σ^2_{noise}. These results can be used to measure the settling time to adapt to an addition to the background, and the variance of the estimated mean. Figure 4-1 shows a plot of the curves that result from these two inputs.

The points shown on the variance curve represent the values of α that are realizable in the implemented system. Values of α must be of the set 2^n, where n is an integer less than zero; the reasoning for this constraint will be discussed in section 4.3. The point circled in Figure 4-1 is $\alpha = \frac{1}{16}$, and represents a good compromise between settling time and variance. The variance due to choice of α is $\frac{1}{20}$ of the noise variance, small enough that it can be neglected. The settling time is approximately 75 frames. This settling time is longer than desired, but as Figure 4-1 shows, the point chosen is on the corner of the curve, such that the settling time is not significantly improved without a large increase in α.

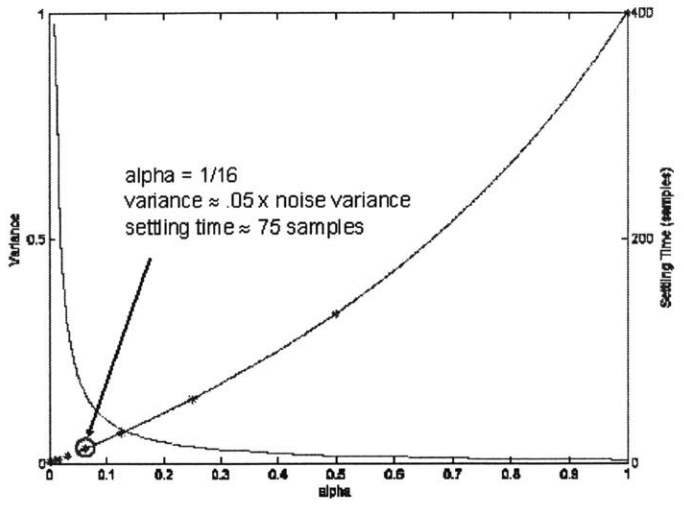

Figure 4-1: Choice of α on Noise Variance and Settling Time

4.1.3 Choosing τ

The next step in the basic background subtraction algorithm is comparing the current frame with the background model to detect significant deviations. To begin this comparison, the absolute error between the current frame and the background, $E_t(n_1, n_2)$, is calculated using

$$E_t(n_1, n_2) = |\mathcal{I}_t(n_1, n_2) - \beta_t(n_1, n_2)| \tag{4.4}$$

Once the error frame is calculated, foreground pixels can be detected by comparing this frame with a detection threshold, τ. Each pixel, (n_1, n_2), is modeled by a Gaussian distribution with mean $\beta_t(n_1, n_2)$, and standard deviation σ_{noise}. Using this distribution, the probability, ρ, of an absolute deviation occurring that is greater than τ, given that it is part of the background, can be calculated. Thus, the detection threshold can be set using

$$\tau = \left[ICDF(\frac{1+\rho}{2})\right] \cdot \sigma_{noise} \tag{4.5}$$

where $(1 - \rho)$ is the desired probability of false alarm, and $ICDF(\cdot)$ is the inverse

zero-mean normal cumulative distribution function, with a standard deviation of one. A binary foreground map, $F(n_1, n_2)$, can then be generated using

$$F_t(n_1, n_2) = \begin{cases} 1, & E_t(n_1, n_2) > \tau \\ 0, & E_t(n_1, n_2) \leq \tau \end{cases} \tag{4.6}$$

where 1's represent foreground pixels and 0's background.

Once the foreground detection map is computed for a given frame, the Through The Lens Trigger is easily determined. The simplest trigger would signal a detection if any of the pixels in the foreground map are set to one. More complicated triggers might require that a certain number of pixels in the foreground detection map be set to one, or that there be a grouping of a certain size. In this implementation, the single detection criterion is used in conjunction with enhancements to the basic algorithm.

4.2 Algorithm Enhancements

While the basic algorithm works well under the assumption of a static scene, the introduction of local motion or global illumination changes violate the static assumption, and make the algorithm ineffective in reliably detecting scene changes. In existing background subtraction methods, these effects are handled using multiple Gaussian mixture models and normalized intensity values. Using multiple Gaussian models is not possible in this implementation, as the power constraint severely limits the amount of logic that can be used for algorithm computation. An approach is then required that allows the model to perform effectively using a single-Gaussian model.

4.2.1 Pixel Binning

The idea of pixel binning is to divide the image into blocks of $m \times n$ pixels, and average the intensities in each block to form a single pixel. This is used in this system to mitigate the effect of local motion through spatial averaging. For example, take a tree branch against a uniform background that is 2×3 pixels in size, with a spatial variation of 4 pixels in each direction due to blowing wind. If the branch were placed

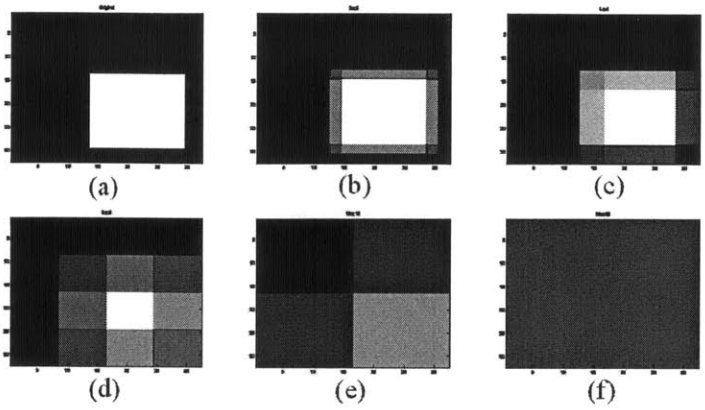

Figure 4-2: Effect of bin size on target/background contrast. Target is 16×16 in (a), bin sizes are (b)2×2, (c)4×4, (d)8×8, (e)16×16, (f)32×32.

in the middle of a 6×7 block, the average of the pixels would change very little, as long as it remained inside the bin.

The price paid for doing this averaging is a dulling of the contrast of the image as the resolution is reduced. The larger the bin size, the more the scene intensities will blend together. This is a problem for background subtraction, since it operates on the principal of differentiating foreground intensity from background intensity. Decreasing the contrast between foreground and background will make this determination much more difficult. If the approximate target size is known, and the bin size is chosen such that it is half this size in the horizontal and vertical directions, it is guaranteed that at least one pixel will have the maximum contrast between foreground and background. An example of this is shown in Figure 4-2(a-f), where in Figure 4-2(d) the bin size is set at half the horizontal and vertical target size.

In order to examine how the binned pixels behave, the probability distribution functions of their intensity must be determined. The PDFs of the binned pixels are the combination of the PDFs of the pixels in the bin. In a local region, pixels are assumed to have approximately the same PDF, with a mean, μ_{pix}, and variance, σ^2_{pix}. In many cases this distribution will be multi-modal, representing the different objects that appear in the pixel, weighted by their relative occurrence. An example of a distribution with two illuminations, possibly a tree and a road, is shown in Figure

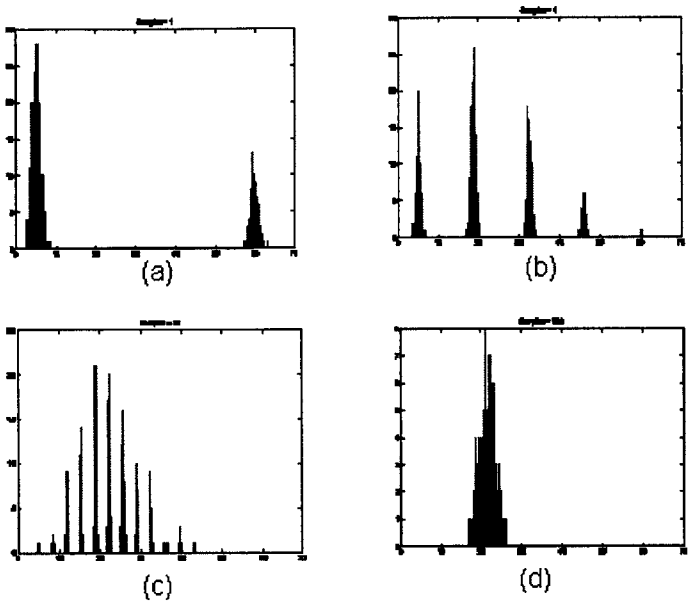

Figure 4-3: Sampled Bimodal Distribution - (a) Original Distribution, (b) $n = 2$, (c) $n = 16$, (d) $n = 256$

4-3(a).

Under the assumption that all the pixels in each bin have the same PDF, the values of each pixel in the bin are essentially samples of that distribution. Using the Central Limit Theorem [8], if the number of pixels in the bin is large enough, the PDF of the bin can be approximated by a Gaussian distribution with mean μ_{pix}, and variance $\frac{\sigma_{pix}^2}{n}$, where n is the number of pixels in the bin. The distribution that results by sampling 4-3(a) for different values of n is shown in Figure 4-3(b-d).

Binning can then be used to create a unimodal distribution out of a multi-modal distribution. The mean of a binned pixel frame, $\mu(b_1, b_2)$, can be calculated with a running average of the bin intensity, just as it was in the basic algorithm. Unlike the basic algorithm, the variance is not a constant value that can be measured a priori, and is different for each bin.

Variance can change significantly from bin to bin for two reasons. The first is that local motion will not always be contained within a single bin. In many cases a moving object will fall into several bins, adding variance from movement in and

out. The second is that the background behind moving objects will not always be uniform. As a result, the bin intensity can vary from an object within the bin moving and uncovering different parts of the area behind it. These problems suggest that an adaptive method of estimating the bin intensity variance is needed to be able to set individual detection thresholds for each bin.

The first method explored for estimating the variances uses the same type of running average that is used to approximate the background mean. The variance is represented by $\sigma^2_{bin}(b_1, b_2)$, where (b_1, b_2) are the binned pixels. To calculate the variance frame, the error frame is used as the input to a running average, calculating the average error for each bin at time t, $\epsilon_t(b_1, b_2)$. This is represented by

$$\epsilon_{t+1}(b_1, b_2) = \alpha \cdot \epsilon_t(b_1, b_2) + (1 - \alpha) \cdot E_t(n_1, n_2) \tag{4.7}$$

It follows from the properties of a Gaussian distribution that the average error is 0.6745 times the standard deviation. The standard deviation of the binned pixels are then given by

$$\sigma_{bin_t}(b_1, b_2) = \frac{\epsilon_t(b_1, b_2)}{0.6745} \tag{4.8}$$

With this estimate of the variance, a desired probability of false alarm can be used to set the threshold as some multiple, s, of the bin average error, based on a desired probability of false alarm $(1 - \rho)$. This is represented by

$$\tau_t(b_1, b_2) = s \cdot \epsilon_t(b_1, b_2) \tag{4.9}$$

where

$$s = \frac{ICDF(\frac{1+\rho}{2})}{0.6745} \tag{4.10}$$

In practice, a minimum fixed threshold value is also needed because quantization error causes the error for pixels with very little motion to be calculated as zero. Without a minimum threshold, these pixels cause frequent false alarms. Figure 4-4 shows the

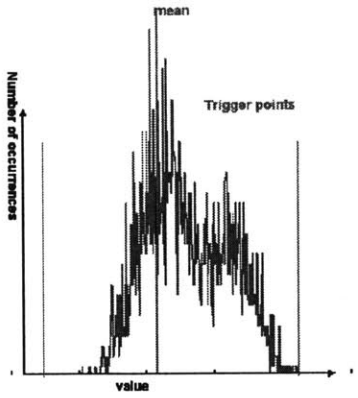

Figure 4-4: Example of Gaussian shaped histogram of pixel bin

histogram of a bin with local motion over several frames. This shows that the bins do follow an approximately normal distribution.

Using the running average method of estimating the binned pixel variance has advantages and disadvantages. The advantage of this method is that it provides an adaptive method of setting the detection threshold without the use of feedback from a higher-level processor. The disadvantage is that with the slow 1 fps frame rate, the variance is not very quick to adapt to changes in local motion. This requires that the motion be fairly constant in order to build up a good model.

4.2.2 Adaptive Feedback

The second method explored for setting the detection threshold uses the concept of adaptive feedback. The idea is to occasionally use computationally intensive processing that is much higher power to better determine scene complexities, and use this information to "teach" the TTLT to perform better. This is done using several frames of high resolution imagery following the trigger, as well as the binary pixel map. The exact method the DSP would use to process the imagery is still under consideration, but it would likely be some type of morphological processing that would take advantage of knowledge about target construction and movement.

Adaptive feedback forms the basis for the second method of variance estimation.

In this method, feedback from the DSP is used to adjust fixed thresholds for bins that cause frequent false triggers. The algorithm begins by using the same low threshold for every bin. When a trigger occurs, the DSP performs high-level processing to determine if the trigger was in fact real, and if not, feedback information to the TTLT to desensitize the pixels that caused the false detection.

An example of this method is illustrated in Figure 4-5. In this example, there is a tree in the background whose branches move around in the wind. When the algorithm is first started, this would likely cause a trigger, and send high resolution imagery to the DSP. The DSP could determine from the fact that the tree was simply waving back and forth, causing scattered detections, that this trigger was not caused by an event of interest. It would then send a command to the TTLT to increase the threshold for the bins containing the tree, such that a very large deviation would be needed to cause a detection. In another instance, a car could drive through the image, also causing a TTLT detection. The DSP would process the high resolution imagery and determine that a contiguous block of pixels caused the detection, which show a clear direction of motion over several high resolution frames. Based on this analysis, the DSP could determine that the detection was an event of interest, and transmit the imagery.

The advantage of this type of algorithm over the adaptive error estimation method is that the scene is permanently learned. Thus, if local motion dies down and then picks back up, there is a much lower probability that a false trigger will occur because that area is permanently desensitized. The disadvantage of this method is that the startup time in the beginning is very long, and relies very heavily on adaptive feedback. Another disadvantage is that if local motion in a certain region stops permanently, that area will still be desensitized.

Adaptive feedback can also be used with the first method described. In this case, DSP feedback would alter the multiplier used to calculate the detection threshold from the average error. This would allow feedback to desensitize pixels with a lot of local motion, while still allowing the TTLT to maintain a self-adaptive threshold.

The threshold values, $\tau_t(b_1, b_2)$, generated by the chosen method, can be used to

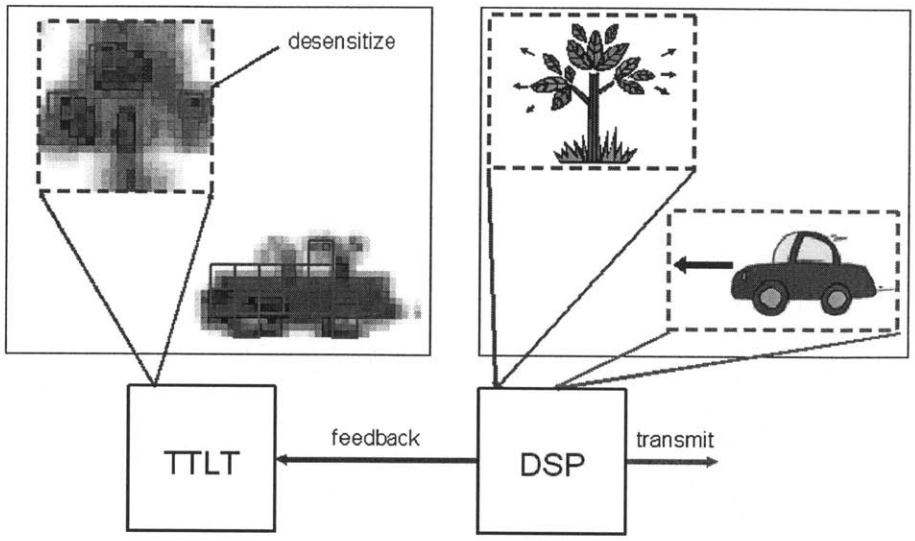

Figure 4-5: Example of Adaptive Thresholding Using DSP Feedback

generate a binary foreground map, $F_t(b_1, b_2)$, as in section 4.1. This is represented by

$$F_t(b_1, b_2) = \begin{cases} 1, & E_t(b_1, b_2) > \tau_t(b_1, b_2) \\ 0, & E_t(b_1, b_2) \leq \tau_t(b_1, b_2) \end{cases} \tag{4.11}$$

The bin size is set such that only one bin is guaranteed to have maximum contrast between the target and background. Thus, a trigger is generated if any bin is detected as foreground.

4.2.3 Choosing g

The binary pixel map can also be used to detect global illumination changes. Changes in global illumination would cause intensity values to change all over the image, resulting in a large number of detections, especially in bright areas. This is in contrast to an event of interest, which would likely be small compared to the size of the image, and cause only a small number of detections. Thus, a constant g could be set, such that if the number of detections were greater than g, the image would be determined to have undergone a global illumination change. To ensure that an object of interest is not mistaken for a global illumination change, g should be set at least twice as

Symbol	Parameter	Source
α	Learning Coefficient	preset
m	Horizontal Bin Size	preset
n	Vertical Bin Size	preset
g	Global Illumination Threshold	preset
$\tau(n_1, n_2)$	Trigger Threshold	preset or adaptive

Table 4.1: Trigger Algorithm Parameters

large as the number of pixels the desired target is expected to occupy.

If a global illumination change is detected, no trigger occurs, and the background model must be adjusted. To make this adjustment, the background mean is set to the current image intensity in order to immediately acquire the new background illumination. The adaptive mean then proceeds under normal operation.

4.3 Algorithm Summary

The sections above have outlined a method for through the lens detection using extremely power constrained processing. The basic algorithm uses a running mean based on the learning coefficient α to maintain an adaptive model of the background intensity. Pixel binning is used with the basic algorithm to average out local motion in order to combine multi-modal distributions into unimodal distributions . The threshold, $\tau(b_1, b_2)$, for each bin can be set using either adaptive error estimation or fixed thresholds that can be altered through adaptive feedback. The determination of which method should be used depends on scene dynamics, and their relevance to the advantages and disadvantages of each method. The computation of the enhanced algorithm results in a binary foreground map, which is used in trigger determination and detection of global illumination changes. The algorithm parameters are summarized in Table 4.1.

4.4 Logic Implementation

This chapter has developed a single-Gaussian based background subtraction algorithm with the ability to adapt to background changes, local motion, and global illumination effects. In order for this algorithm to be used for the Through The Lens Trigger, it must operate in real time, and be computed with the logic in a single CoolRunner CPLD. This section will describe the logic implementation of this algorithm in CPLD 2. All logic for this device was written in VHDL, and synthesized with the Xilinx ISE software. The top level source code for this device is *CPLD2.vhd*, included in Appendix B with all associated modules.

Within the second CPLD there are two main logic blocks, binning logic and background subtraction logic. In addition to these two blocks there is also a memory interface and a parameter file. The parameter file stores the parameters for the background subtraction algorithm, and is programmed using a standard 3-wire interface, either from CPLD 1 or the frame grabber. The external connections to CPLD 2 are a bus to CPLD 1, a frame grabber bus, and a memory bus. This architecture is shown in Figure 4-6, where the thick arrows are data buses, and the thin arrows control signals.

4.4.1 Binning

The first step in the computation of the TTLT is pixel binning. Image data are sent to CPLD 2 from CPLD 1, along with frame and line signals. The data are binned as they comes in, then stored in the memory bank for background subtraction processing. As described in section 4.2, the idea of binning is to divide the image into blocks of $m \times n$ pixels, then sum the pixels in the block together to form bins. Two of the major restrictions on this system are bus widths and memory capacity. These require that binned pixels take as few bits as possible. For this reason, binning is done by averaging the pixels in each block, rather than just summing them. This keeps the binned pixel intensities to a maximum of twelve bits, just as in the original image data. Another restriction imposed by the limited logic available is the impossibility

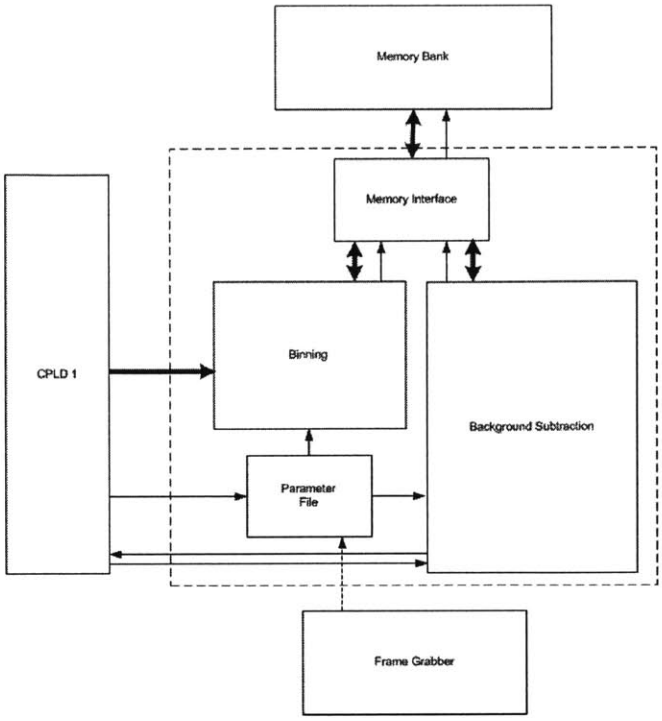

Figure 4-6: CPLD 2 Block Diagram

of implementing a multiplier/divider. The division necessary for averaging must be accomplished by bit down-shifting, allowing only divisors of 2^n, where n is an integer. Divisors that are not of the set 2^n are rounded up to the next value in the set.

The image data are sent by CPLD 1 in a line scan order. This means that the entire frame is sent line by line and each line is sent a pixel at a time. The background subtraction algorithm involves binning in both the horizontal and vertical directions. This requires that blocks of m pixels in each line be combined over n lines. In order to do this, intermediate summations must be stored to collect blocks over multiple lines. This can be done with in-place computation, where only a single memory location is needed for each bin. Intermediate values are read from that location, summed with the incoming pixel values for that bin, then stored back in the same location.

A diagram of the binning logic is shown in Figure 4-7. The inputs to the logic block are the video data and memory bus. The output is also connected to the memory bus. For line 1 of n in each block, the signal *zero_sel* is set to '1' so that a zero input

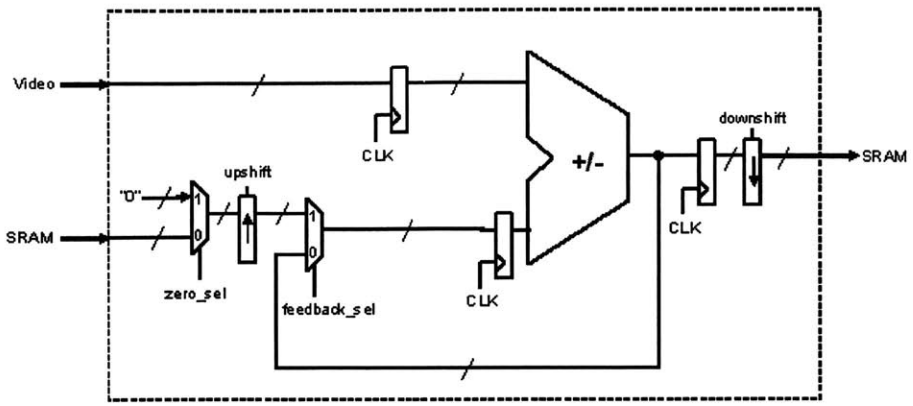

Figure 4-7: Binning Logic

is selected instead of the memory bus, because there are no intermediate values yet. For the lines $2 : n$ of each block, *zero_sel* is set to '0' so that the intermediate values are read in from the memory. The data are then up-shifted to account for the fact that previous summations were down-shifted. This is necessary to make the relative magnitudes of the video data and intermediate values correct.

For pixel 1 of m for each block within the line, the signal *feedback_sel* is set to '1' so that the intermediate summation is sent through. For pixels $2 : m$, *feedback_sel* is set to '0' so that the incoming video values are summed with all the previous values. Once all m pixels for that block are summed, the data are down-shifted and then written back to the memory location for that pixel. This sequence of operation shows how in place computation is performed to calculate each bin.

Memory addressing and select signal generation are done using four counters, two for intra-block counting, and two for inter-block counting. The intra-block counters are set to count up to m and n, then roll over and count again. The values from these counters are used to generate the signals *zero_sel* and *feedback_sel*. Every time the counters roll over, they increment their respective inter-block counters. The values of the inter-block counters are used to generate the row and column memory addresses. The column counter is reset by the line signal, and the row counter by the frame signal.

4.4.2 Background Subtraction

Once the image data are binned, the next step in TTLT processing is background subtraction. TTLT processing consists of thresholding the current error frame and updating the background parameters. This is done using a microprocessor type architecture, whereby control signals are generated by a state machine to run a simple program that performs the threshold operation and background model update for each pixel.

Background subtraction is started when a 'start' signal is received from the primary controller in CPLD 1. A state machine then runs the background subtraction processing cycle for each bin.

Figure 4-8 shows the logic that makes up the background subtraction processor. The inputs to the processor are the binned frame and background model frames, which are accessed through the memory bus. The outputs are the updated background frames to write back to the memory, and a detect signal that is set to one for each pixel that is detected as foreground, in addition to writing the value into the binary pixel map.

The learning coefficient, α, is programmed into the parameter file. This value is limited to elements of 2^n, where n is an integer less than zero. This is because of the restriction that all divides must be accomplished by bit shifting. The value set for α determines the number of bit shifts that are used in the calculation of the running mean. The nominal value for this parameter is $\frac{1}{16}$, as described in section 4.1.2, but is also programmable.

The current bin frame and background parameter frames are all stored in a single memory. The maximum number of bins, using $m = n = 1$, is the input image size of 480×270. Thus, a maximum of 18 bits of address space is needed to store a frame of data. Each of the four memories has 20 bits of address space, so a single memory can provide four frames of storage. This allows the bin intensity frame, background mean, background variance, and binary pixel map to all be stored in the same memory. Each bin has an 18-bit address, with the final two bits corresponding to each of its four

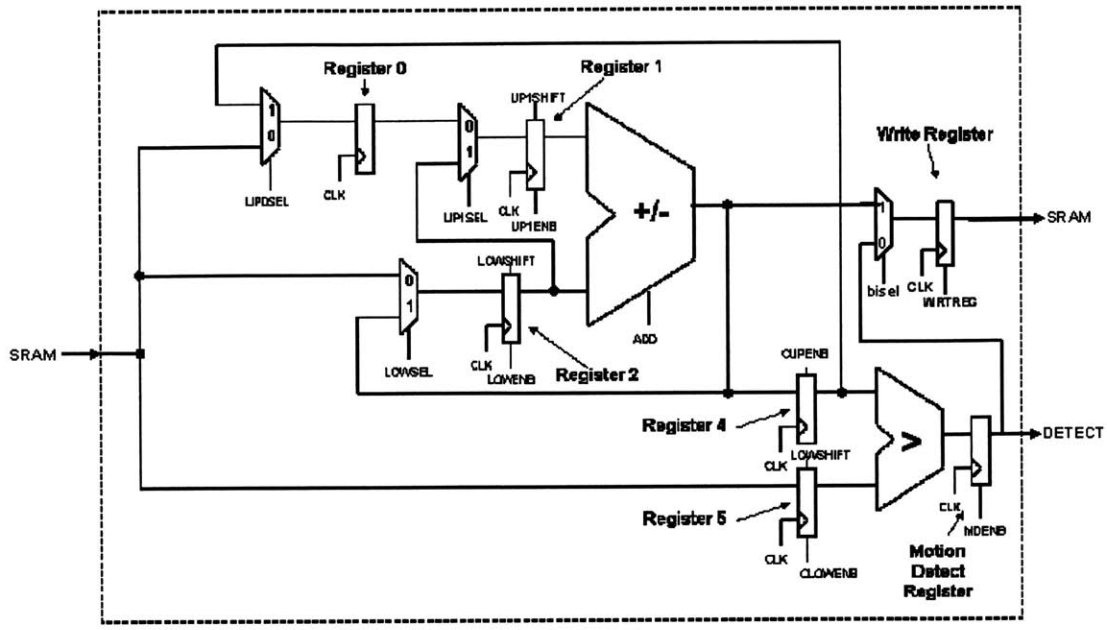

Figure 4-8: Background Subtraction Logic

associated frames. The eighteen bit address is generated by a counter that moves sequentially through all the bins. The two frame selection bits are set by program control signals.

The background variance frame is either used to store the average error or fixed threshold value, depending on which method of thresholding is used. These parameters are updated by adaptive feedback through CPLD 1, which writes parameter updates from the DSP directly into the corresponding memory location.

During background subtraction processing, a count is kept of the number of foreground pixels detected. This is done using the *detect* output of the processor. At the end of processing, a trigger is sent to the primary controller in CPLD 1 if the count is greater than zero and less than g. This simple counting system works just as well as using the binary pixel map because the detection parameters rely on total counts, not spatial location. Once all the pixels are processed, and the trigger is determined, the state machine returns to a wait state, and sends a *done* signal back to the primary controller in CPLD 1.

4.4.3 Testing and Debugging

The TTLT hardware consists only of CPLD 2 and the memory bank. These devices were operationally tested as part of the testing and debugging described in Chapter 3. The only testing and debugging that had to be done on the TTLT processor was debugging the VHDL source code.

The first step in debugging was simulation in the ModelSim environment. This was done to ensure proper operation of the state machine and counters. The logic was then implemented in the CPLD, and further debugging was done using built in test points and a logic analyzer.

4.5 Summary

This section has described the theory and implementation of the Through The Tens Trigger. The algorithm is based on the idea of using binning and adaptive feedback to allow a single-Gaussian background subtraction method to perform well under varying background conditions. It is implemented in logic using in-place computation of pixel bins, and a simple microprocessor architecture for background subtraction processing.

Chapter 5

Results and Conclusion

The previous two chapters have described in detail the implementation of the camera hardware and the TTLT algorithm. These two pieces combine to form the Lincoln Laboratory camera, the main component of the TTLT demonstration system. The first part of this chapter will describe the demonstration system and results. The second part will provide some concluding comments and discuss future extensions of the work in this paper.

5.1 TTLT Demonstration System

The final product of the work described in this paper is an operational system that demonstrates the feasibility of through the lens triggering. This system contains all the hardware described, except for the transmitter, which is replaced by a computer that is connected to the DSP over USB. An application running on the computer receives and displays the full resolution imagery captured when a trigger is detected. The computer application also displays a power monitor that displays the system power over time. This feature is used to demonstrate the significant reduction in power achieved during trigger operation.[1]

The frame grabber interface is used to demonstrate the operation of the TTLT algorithm by displaying the adaptive background model in real-time. This is done by

[1]Computer application and DSP software developed by Jim Sopchack at MIT Lincoln Laboratory

reading out the memory space in which the background model is stored through the frame grabber. This allows the background data frames to be viewed, including the binned image, background mean, average error, and binary pixel map.

The operation of the TTLT algorithm can be illustrated using imagery of background data, taken through the frame grabber interface. Figure 5-1(a) shows the stored background over time as an apple is adaptively incorporated. The last image in this sequence is the binary foreground map generated when the apple was first placed in the field of view. Figure 5-1(b) shows the same type of image sequence as 5-1(a), but in this case 16×16 binning is used and the apple placed such that the binned pixels are a quarter of the apple's size. The incorporation of the apple into the background is more difficult to see in this case than in the previous case, but the binary foreground map in the final image clearly shows that the apple was detected when first placed in the scene. This demonstrates that although pixel binning does not always provide imagery with enough resolution for a good picture by a person's standards, it does provide imagery sufficient for background subtraction.

5.1.1 Performance

Testing was done on the TTLT to determine its performance in different background conditions. The backgrounds used were a static indoor lab scene and a dynamic outdoor scene. In addition to varying background conditions, the thresholding method used was also varied, using both the fixed and variable methods described in Chapter 4. Implementation of adaptive feedback is beyond the scope of this paper, and has not yet been done.

The first scenario tested was the static indoor lab scene. The fixed and variable threshold methods performed very well, as expected, with virtually no false triggers and perfect detection. The two thresholding methods were essentially the same, as the minimum threshold for the variable threshold method was set equal to the fixed threshold value.

The second scenario tested was detection in an outdoor scene using a fixed threshold. The scene used was the view from a window, looking out on a parking lot

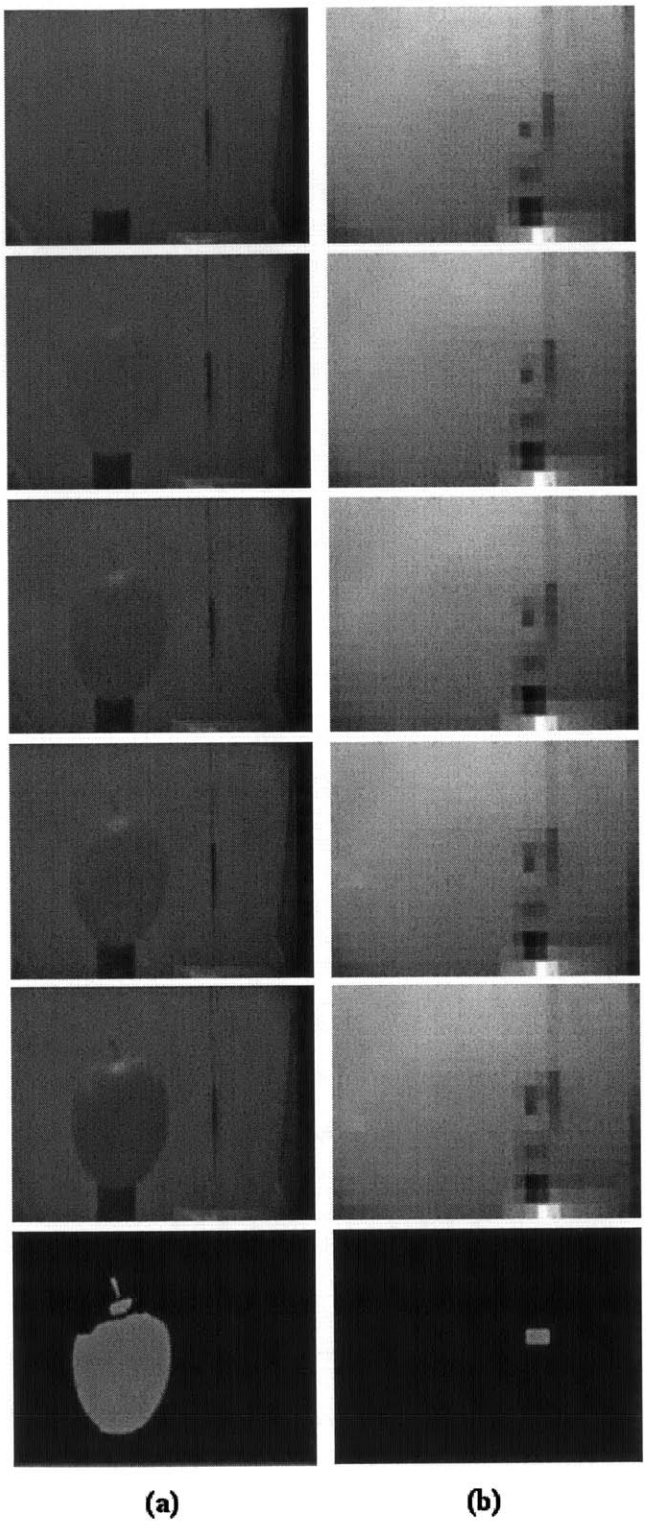

Figure 5-1: Adaptive Background Model Example

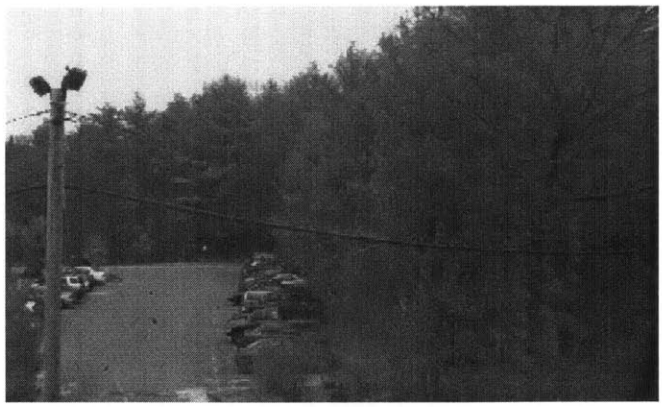

Figure 5-2: Background used for dynamic scene testing

surrounded by trees on a windy day, shown in Figure 5-2. This provided a good dynamic background with trees waving in the wind. It also provided several targets, with people walking by and cars pulling in and out of the parking lot. Cars in the lot corresponded to approximately 16×50 pixels in size with the optics used, and people approximately 18×6 pixels. This provided good variation in target size to judge the effect of bin size on detection.

A range of bin sizes using both a low and high threshold value were tested, and the number of false pixel detections per frame recorded. A plot of false alarms versus bin size for both threshold values is shown in Figure 5-3. The false alarms were normalized by multiplying the number of recorded false bin detections by the number of pixels per bin. A *log* scale is used for both axes in order to better display the shapes of the curves. This graph clearly shows that the number of false foreground detections per frame is reduced with increasing bin size. As discussed in Chapter 4, the tradeoff for this is a loss in contrast between targets and the background. Although there were not enough targets present to take accurate measurements, it was observed that cars were always detected for all bin sizes, but people were not reliably detected for larger bin sizes, especially in the 16×16 case. This is the expected result, as contrast is lost when the pixels become larger than a quarter the size of the target.

Figure 5-3 also shows that the number of false alarms is decreased as the threshold is increased. Examination of the data in the low threshold case reveals that most of

the detections were concentrated in areas of frequent motion. The number of false detections in these areas was decreased with the higher threshold, and reduced to zero in the 16×16 case. The downside to the increase in threshold was a noticeable decrease in the likelihood of people being detected. This shows that adaptive feedback could significantly reduce the number of false detections without significantly affecting the positive detection rate by desensitizing only those areas in which a significant number of false triggers occur.

The third scenario tested was similar to the one just discussed, but was done using an adaptive threshold based on running error estimation. The same outdoor scene was used as for the fixed threshold, on the same day and approximate time. The algorithm was tested with several different bin sizes, using both a low and high average error multiple to determine the threshold. The results for detections and false alarms are plotted in Figure 5-3, along with the results from the fixed threshold testing.

These results showed a similar trend to the fixed threshold in that the number of false detections per frame decreased with increasing bin size. It was observed that cars were always detected at larger bin sizes, while people became unreliably detectable, just as in the fixed threshold case. The higher multiple showed improved performance over the lower multiple, suggesting that using adaptive feedback to increase the threshold error multiple in highly varying areas could be used to significantly improve false alarm performance.

The experimental results overall show that the TTLT performs very well in static environments, and moderately well in dynamic environments. In the static background case the false alarm rate is effectively zero, corresponding to a number of false triggers well below the 75 maximum per day rate estimated in Chapter 2. The system performed moderately well with a dynamic background, but in most cases had a false alarm rate of over a pixel per frame, corresponding to a false trigger every frame. The one case in which this did not occur was the largest bin size with a fixed threshold. This suggests that while the variable threshold method works better in general, it is difficult to get down to zero false alarms. For both the fixed and variable threshold

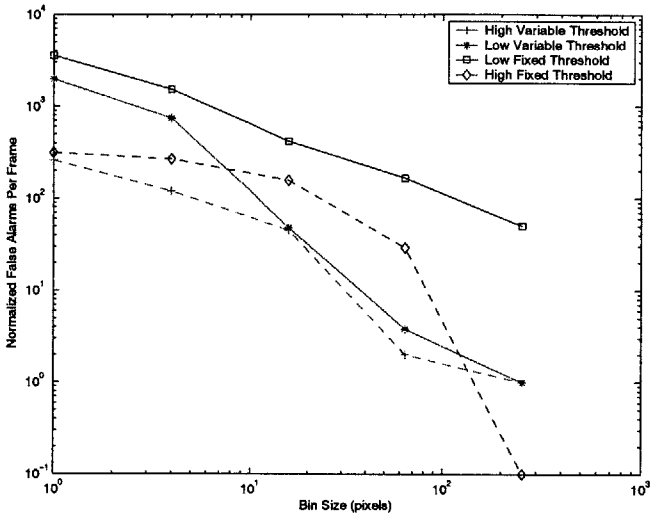

Figure 5-3: False Alarm Performance

cases, the improved performance with increasing thresholds suggests that with adaptive feedback to better determine individual pixel thresholds, those few pixels that caused consistent false alarms could be desensitized even further, allowing the false alarm rate to approach zero in order to achieve fewer than 75 false triggers per day.

5.2 Future Work

The first piece of future work is completing the current system beyond the TTLT. Adaptive feedback and transmitter interfacing still need to be implemented to complete the system outlined in this paper. The completed system also needs to be packaged in a weather resistant chassis so that it can be placed outside and field tested.

In addition to completing the current system, there are simple modifications that could be made to expand the capabilities of the Lincoln Laboratory camera for specific applications. One of the applications for which this type of camera system could become very useful is border monitoring. Several TTLT cameras could be set up at intervals, pointing down a border to detect and image anything that crosses.

A test bed for this type of system has already been conceived at MIT Lincoln Lab-

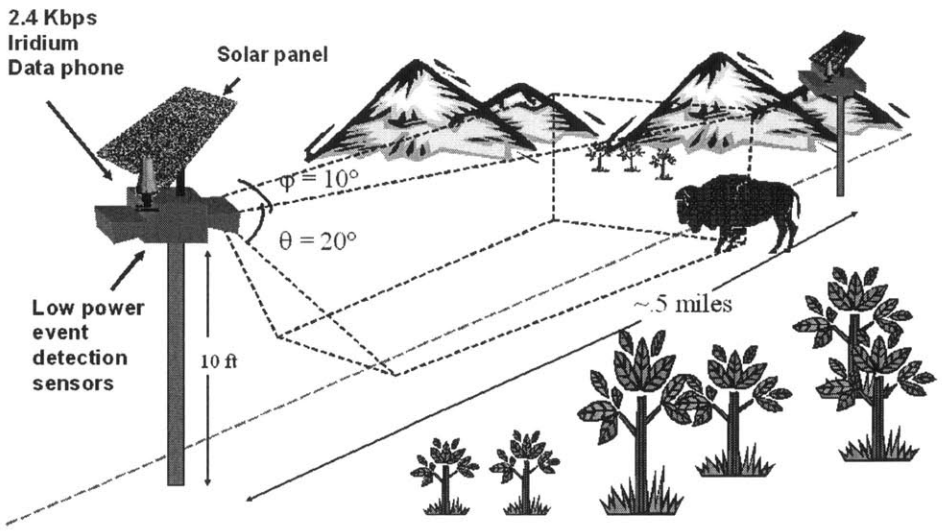

Figure 5-4: Proposed Border Monitoring System

oratory. The proposed system is designed to monitor a five mile border in Yellowstone National Park, and detect bison that attempt to leave the park in the winter in search of food. The system would be set up as shown in Figure 5-4[2], whereby cameras are placed on poles, with two cameras per pole pointing in each direction down the border. The cameras are spaced such that there is redundant coverage, with two cameras covering the space between each pole. With this type of redundant system, false alarm mitigation could be improved by correlated detection through inter-camera communication. This is an example of an application specific modification that expands system capabilities.

Another possible extension of the current TTLT system is the use of color in trigger processing. Many background subtraction techniques in current research use color information as a key aspect of their detection methodologies [6],[10]. This type of information could be especially useful in the Lincoln Laboratory system if the color characteristics of the object of interest were known. For example, if the object were known to have a large red component, the detection algorithm could be run on only the red component of the RGB data, effectively filtering out objects in other colors

[2]Proposal and graphic by Dr. William Ross at MIT Lincoln Laboratory

such as trees or sky. If a trigger was determined, full-color high-resolution imagery could then be transmitted.

These are only a few of the possible extensions of the work presented in this paper. This new area of research offers many opportunities for future research in signal processing for low power applications.

5.3 Conclusion

This paper has described work done at MIT Lincoln Laboratory to develop a low power surveillance system using image based triggering. This work has specifically focused on the Through The Lens Triggering aspect of the system. This is a new area of research, but is related to previous research in externally triggered low power cameras, and background subtraction algorithms without power constrained processing.

The low power hardware developed for the TTLT uses a combination of low power devices and power conscious operation to achieve the lowest power possible and still maintain necessary functionality. The TTLT algorithm uses background subtraction based on a single-Gaussian model for each pixel. A single-Gaussian approximation is achieved in dynamic backgrounds by using binning to reduce local motion effects through averaging. These effects are further mitigated using adaptive thresholding based on average error, and adaptive feedback from high-level processing in a DSP.

The experimental results show that the TTLT works very well against static backgrounds, and moderately well in dynamic outdoor backgrounds when the bin size is appropriately chosen. It is hoped that future work with adaptive feedback from a DSP will help mitigate dynamic background effects, and allow the system to operate with fewer than 75 false alarms per day to achieve the desired power consumption.

The research presented in this paper provides a solid first step towards the goal of producing a fieldable system that can be used for continuous video surveillance without human intervention for long periods of time. This type of system would provide the capability for video monitoring with higher resolution, higher flexibility, and less risk than any system currently available.

Bibliography

[1] BECTA. Recent trends in digital imaging. Technical report, British Educational Communications and Technology Agency, March 2004.

[2] N. Friedmand and S. Russell. Image segmentation in video sequences: A probabalistic approach. In *Proc. of the Thirteenth Conference on Uncertainty in Artificial Intelligence(UAI)*, Aug. 1-3 1997.

[3] A. El Gamal. Trends in cmos image sensor technology and design. *IEDM*, December 2002.

[4] X. Gao, T. Boult, F. Coetzee, and V. Ramesh. Error analysis of background adaptation. *CVPR'00*, June 2000.

[5] GM Battery. *Thionyl Chloride Lithium Battery (ER34615).*

[6] G. Gordon, T. Darrell, M. Harville, and J. Woodfill. Background estimation and removal based on range and color. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, June 1999.

[7] M. Harville. A framework for high-level feedback to adaptive, per-pixel, mixture-of-gaussian background models. *EECV*, pages 543–560, 2002.

[8] S. Kachigan. *Statistical Analysis.* Radius Press: New York, 1986.

[9] T. Matsuyama, T. Ohya, and H. Habe. *Background Subtraction for Non-Stationary Scenes.* Department of Electronics and Communications, Graduate School of Engineering, Kyoto University: Sakyo,Kyoto, Japan, 1999.

[10] C. Stauffer and W.E.L. Grimson. Adaptive background mixture models for real-time tracking. *CVPR'99*, 2:255–261, June 1999.

[11] K. Toyama, J. Krumm, B. Brumitt, and B. Meyers. Wallflower: Principles and practice of background maintenance. *ICCV'99*, pages 255–261, Sept. 1999.

[12] C. Wren, A. Azarbayejani, T. Darrell, and A. Pentland. Pfinder: Real-time tracking of the human body. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:780–785, July 1997.

Appendix A

Hardware Parts List

#	QTY	Part Number	Value	Description
1	53	YAG 04022F10427B20D	.1UF	CAP0402
2	16	PAN ECJ-3YF104Z7A106Z	10UF	CAP1206
3	1	PAN ECJ-0EF1H103Z	.01UF	CAP0402
4	3	SAM FTMH-125-02-F-DV-ES-A-P		SAM 2X25
5	3			ETCH
6	1	MOLEX 87332-1420		JTAG 2X7 2MM
7	1			HDR2X18
8	1			HDR2X25
9	3	BERG 68024-136 2 FROM		HEADER2X1
10	1	SAMTEC TSW-107-07-S-D		JTAG 2X7 1MM
11	1	TUCHEL ELECTRONICS		MMC RECEPTICAL
12	2			JMPR3
13	3	BLM31P500S	80 AT	IND EMI
14	8			MTG VIA
15	3	LL115-100	10K	RES0805
16	1	LL114-200	2K	RES0805
17	2	LL112-332	33.2	RES0805
18	11	LL114-475	4.75K	RES0805
19	3	LL116-100	100K	RES0805
20	3	LL113-332	332	RES0805
21	3	PAN ERJ-6GEY0R00V	0	RES0805
22	2	EECO 230057GB		HEX SWITCH
23	1	LINEAR TECH LTC6900CS5		LTC6900
24	2	XILINX XC2C512-6PQ208		XC2C PQ208
25	1	NATIONAL DS90C032TM		DS90C032TM
26	4	SAMSUNG K6F1616R6A-EF70		K6F1616R6A
27	1	ROCKWELL PROCAM HD		PROCAM 65
28	5	NATIONAL DS90C031TM		DS90C031TM
29	1	NATIONAL LP2981IM5-2.5		LP2981 2V5
30	2	NATIONAL LP2981IM5-3.3		LP2981 3V3
31	1	AT25128N-10SI-1.8		AT25128
32	1	NATIONAL LP2992IM5-1.5		LP2992 1V5

Appendix B

VHDL Source Code

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


entity CPLD1 is
    Port (clk : in std_logic;
                clk_out : out std_logic;
                --ProCam Interface
                pclk : out std_logic;
                video : in std_logic_vector(11 downto 0);
                framesync, linesync : out std_logic;
                framestart, linestart : in std_logic;
                ser_clk,ser_stb,ser_dat : out std_logic;
                ponresetb,resetb : out std_logic;
                ret_dat : in std_logic;
                --Memory Interface
                mem_data : inout std_logic_vector(15 downto 0);
                mem_addr : out std_logic_vector(19 downto 0);
                mem_we,mem_oe : out std_logic;
                mem_cs : out std_logic_vector(3 downto 0);
                --DSP Interface
                dsp_data : inout std_logic_vector(15 downto 0);
                dsp_addr : in std_logic_vector(7 downto 0);
                interupt : out std_logic;
                a_re : in std_logic;
                a_oe : in std_logic;
                a_we : in std_logic;
                a_ce : in std_logic;
                ardy : out std_logic;
                --Frame Grabber(over interconnect bus)
                fg_data : out std_logic_vector(15 downto 0);
                fg_line,fg_frame,fg_clk : out std_logic;
                fg_in : out std_logic_vector(2 downto 0);
                --CPLD interconnects
                motion : in std_logic;
                fg_con,sram_con : out std_logic;
                md_st : out std_logic;
                md_rdy : in std_logic;
                bin_en : out std_logic;
                fg_en : out std_logic;
                --Test Points
                tp1,tp2,tp3,tp4,tp5,tp6,tp7,tp8 : out std_logic;
                mode1,mode2 : in std_logic_vector(3 downto 0);
                --Power Enables
                dsp33_enb,dsp15_enb : out std_logic);
end CPLD1;

architecture Behavioral of CPLD1 is

        signal reset : std_logic;
        signal cycle : std_logic_vector(23 downto 0);

        --SYTSTEM CONTROLLER
        --control
        signal syscontrol_ps, syscontrol_ns : std_logic_vector(4 downto 0);
        constant standby : std_logic_vector := "00000";
        constant fpa_powerup1 : std_logic_vector := "00001";
        constant fpa_powerup2 : std_logic_vector := "00010";
        constant binner1 : std_logic_vector := "00011";
        constant binner2 : std_logic_vector := "00100";
        constant fpa_powerdown1 : std_logic_vector := "00101";
        constant fpa_powerdown2 : std_logic_vector := "00110";
        constant motion_detection1 : std_logic_vector := "00111";
        constant motion_detection2 : std_logic_vector := "01000";
        constant dsp_handoff : std_logic_vector := "01001";
```



```
        constant dsp_handoff_wait : std_logic_vector := "01010";
        constant xframe1 : std_logic_vector := "01011";
        constant xframe2 : std_logic_vector := "01100";
        constant mem_readout1 : std_logic_vector := "01101";
        constant mem_readout2 : std_logic_vector := "01110";
        constant full_pgm1 : std_logic_vector := "01111";
        constant full_pgm2 : std_logic_vector := "10000";

        --signals
        signal cycle_start : std_logic;
        signal frm_st : std_logic;
        signal bin_enb,fg_enb,sram_cont,fg_cont : std_logic;

        signal a_re_int : std_logic;
        signal a_oe_int : std_logic;
        signal a_we_int : std_logic;
        signal a_ce_int : std_logic;
        signal ardy_int : std_logic;


        --PROCAM
        signal fst_pulse,line_pulse,frm_done : std_logic;
        signal bin_frm_cnt : std_logic_vector(19 downto 0);
        signal frame : std_logic;
        signal reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7 : std_logic_vector(15 do
wnto 0);

        --SERIAL PROGRAMMER
        component pgmer is
                Port (s_clk,reset : in std_logic;
                        pgm_st : in std_logic;
                        pgm_done : out std_logic;
                        ser_clk : out std_logic;
                        ser_stb : out std_logic;
                        ser_dat : out std_logic;
                        reg0 : in std_logic_vector(15 downto 0);
                        reg1 : in std_logic_vector(15 downto 0);
                        reg2 : in std_logic_vector(15 downto 0);
                        reg3 : in std_logic_vector(15 downto 0);
                        reg4 : in std_logic_vector(15 downto 0);
                        reg5 : in std_logic_vector(15 downto 0);
                        reg6 : in std_logic_vector(15 downto 0);
                        reg7 : in std_logic_vector(15 downto 0));
        end component;

        signal pgm_strt,pgm_done,pgm_clk,pgm_stb,pgm_dat : std_logic;

        --EDGE
        component edge is
                Port (clk : in std_logic;
                        sig : in std_logic;
                        pulse : out std_logic );
        end component;

        --MOTION DETECTION
        signal md_strt : std_logic;
        signal mot : std_logic;
        signal trigger : std_logic;
        signal trigger_cnt : std_logic_vector(2 downto 0);

        --DSP
        signal dsp_on,dsp_strt : std_logic;
        signal dsp_rdy : std_logic;
        signal dsp_pgm_strt : std_logic;
        signal dsp_frm_strt : std_logic;
        signal dsp_command : std_logic;
        signal dsp_mem_addr : std_logic_vector(20 downto 0);
```

```
signal dsp_mem_data_in : std_logic_vector(11 downto 0);
signal dsp_mem_data_out : std_logic_vector(15 downto 0);
signal dsp_mem_we,dsp_mem_oe : std_logic;
signal dsp_state : std_logic_vector(2 downto 0);
signal interupt_int : std_logic;
signal dsp_cont_req : std_logic;

component dsp_cont is
Port (clk : in std_logic;
        start : in std_logic;
        ready : out std_logic;
        pgm_strt : out std_logic;
        pgm_done : in std_logic;
        frm_strt : out std_logic;
        switch : in std_logic;
        control_request : out std_logic;
        --Sensor
        video : in std_logic_vector(11 downto 0);
        frame_edge : std_logic;
        line_edge : std_logic;
        --SRAM Interface
        mem_addr : out std_logic_vector(20 downto 0);
        mem_data_out : out std_logic_vector(15 downto 0);
        mem_data_in : in std_logic_vector(11 downto 0);
        mem_we : out std_logic;
        mem_oe : out std_logic;
        --DSP Interface
        dsp_addr : in std_logic_vector(7 downto 0);
        dsp_data : inout std_logic_vector(15 downto 0);
        interupt : out std_logic;
        a_re : in std_logic;
        a_oe : in std_logic;
        a_we : in std_logic;
        a_ce : in std_logic;
        ardy : out std_logic;
        state_out : out std_logic_vector(2 downto 0));
end component;

--MEMORY READOUT
signal memd0,memd1,memd_pulse : std_logic;
signal memd_cnt : std_logic_vector(19 downto 0);
signal memd_done,memd_line,memd_frame : std_logic;
signal memd_addr : std_logic_vector(19 downto 0);
signal memrd_strt : std_logic;
signal sram_cs,sram_cs1 : std_logic;
signal mem_addr_int : std_logic_vector(19 downto 0);
signal mem_we_int,mem_oe_int : std_logic;
signal data_enb : std_logic;

--METERING
signal low_level,high_level : std_logic;
signal met_row_cnt : std_logic_vector(8 downto 0);
signal met_row_tc : std_logic;
signal hlev_cnt : std_logic_vector(4 downto 0);
signal hlev_tc : std_logic;
signal llev_cnt : std_logic_vector(4 downto 0);
signal llev_tc : std_logic;
signal inc_cnt,dec_cnt : std_logic_vector(1 downto 0);
signal inc_tc,dec_tc : std_logic;
signal integ_time : std_logic_vector(3 downto 0);
signal tm_tc,tm_ntc : std_logic;
signal dec_int,inc_int : std_logic;

--TEST PATTERNS

signal decode : std_logic_vector(3 downto 0);
```



```
        signal memd_data : std_logic_vector(15 downto 0);
        signal test_pat1 : std_logic_vector(15 downto 0);

begin

        clk_out <= clk;

        reset <= '0';


--_____________________________________________
--OPERATING MODE
--_____________________________________________

        --"00" No Handoff
        --"01" Triggered Operation
        --"10" Always Handoff


--_____________________________________________________
--CYCLE (FRAME) TIMER
--_____________________________________________________

        process(clk)
        begin
                if rising_edge(clk) then
                        --if reset = '1' then
                        --      cycle <= (others => '0');
                        --else
                                cycle <= cycle + 1;
                        --end if;
                end if;
        end process;

        cycle_start <= '1' when (cycle = "00000000000000000000010") else '0';


--_____________________________________________________________________________
--PRIMARY CONTROLLER
--_____________________________________________________________________________


        --State Register
        process(clk)
        begin
                if rising_edge(clk) then
                        syscontrol_ps <= syscontrol_ns;
                end if;
        end process;


        --tp7 <= clk;

        --State Transitions

        process(cycle_start,pgm_done, syscontrol_ps, frm_done,mode1,md_rdy, tri
gger, memd_done, dsp_rdy, dsp_pgm_strt, dsp_frm_strt)
        begin
                case syscontrol_ps is

                        when standby =>
                                --control
                                if cycle_start = '1' then
                                        syscontrol_ns <= fpa_powerup1;
                                else
                                        syscontrol_ns <= standby;
                                end if;
```

```
		--signals
		pgm_strt <= '0';
		frm_st <= '0';
		md_strt <= '0';
		dsp_strt <= '0';
		fg_cont <= '0';
		sram_cont <= '1';
		bin_enb <= '0';
		fg_enb <= '0';
		memrd_strt <= '0';
		sram_cs <= '0';

	when fpa_powerup1 =>
		if pgm_done = '0' then
			syscontrol_ns <= fpa_powerup2;
		else
			syscontrol_ns <= fpa_powerup1;
		end if;
		--signals
		pgm_strt <= '1';
		frm_st <= '0';
		md_strt <= '0';
		dsp_strt <= '0';
		fg_cont <= '0';
		sram_cont <= '1';
		bin_enb <= '0';
		fg_enb <= '0';
		bin_enb <= '1';
		fg_enb <= '0';
		memrd_strt <= '0';
		sram_cs <= '0';

	when fpa_powerup2 =>
		--control
		if pgm_done = '1' then
			syscontrol_ns <= xframe1;--binner1;
		else
			syscontrol_ns <= fpa_powerup2;
		end if;
		--signals
		pgm_strt <= '0';
		frm_st <= '0';
		md_strt <= '0';
		dsp_strt <= '0';
		fg_cont <= '0';
		sram_cont <= '1';
		bin_enb <= '1';
		fg_enb <= '0';
		memrd_strt <= '0';
		sram_cs <= '0';

	when xframe1 =>
		--control
		if frm_done = '0' then
			syscontrol_ns <= xframe2;
		else
			syscontrol_ns <= xframe1;
		end if;
		--signals
		pgm_strt <= '0';
		frm_st <= '1';
		md_strt <= '0';
		dsp_strt <= '0';
		fg_cont <= '0';
		sram_cont <= '1';
		bin_enb <= '0';
		fg_enb <= '0';
```



```
		memrd_strt <= '0';
		sram_cs <= '0';

	when xframe2 =>
		--control
		if frm_done = '1' then
			syscontrol_ns <= binner1;
		else
			syscontrol_ns <= xframe2;
		end if;
		pgm_strt <= '0';
		frm_st <= '0';
		md_strt <= '0';
		dsp_strt <= '0';
		fg_cont <= '0';
		sram_cont <= '1';
		bin_enb <= '0';
		fg_enb <= '0';
		memrd_strt <= '0';
		sram_cs <= '0';

	when binner1 =>
		--control
		if frm_done = '0' then
			syscontrol_ns <= binner2;
		else
			syscontrol_ns <= binner1;
		end if;
		--signals
		pgm_strt <= '0';
		frm_st <= '1';
		md_strt <= '0';
		dsp_strt <= '0';
		fg_cont <= '0';
		sram_cont <= '1';
		bin_enb <= '1';
		fg_enb <= '0';
		memrd_strt <= '0';
		sram_cs <= '0';

	when binner2 =>
		--control
		if frm_done = '1' then
			syscontrol_ns <= fpa_powerdown1;
		else
			syscontrol_ns <= binner2;
		end if;
		pgm_strt <= '0';
		frm_st <= '0';
		md_strt <= '0';
		dsp_strt <= '0';
		fg_cont <= '0';
		sram_cont <= '1';
		bin_enb <= '1';
		fg_enb <= '0';
		memrd_strt <= '0';
		sram_cs <= '0';

	when fpa_powerdown1 =>
		if pgm_done = '0' then
			syscontrol_ns <= fpa_powerdown2;
		else
			syscontrol_ns <= fpa_powerdown1;
		end if;
		--signals
		pgm_strt <= '1';
		frm_st <= '0';
```

```
                md_strt <= '0';
                dsp_strt <= '0';
                fg_cont <= '0';
                sram_cont <= '1';
                bin_enb <= '0';
                fg_enb <= '0';
                memrd_strt <= '0';
                sram_cs <= '0';

        when fpa_powerdown2 =>
                --control
                if pgm_done = '1' then
                        syscontrol_ns <= motion_detection1;
                else
                        syscontrol_ns <= fpa_powerdown2;
                end if;
                --signals
                pgm_strt <= '0';
                frm_st <= '0';
                md_strt <= '0';
                dsp_strt <= '0';
                fg_cont <= '0';
                sram_cont <= '1';
                bin_enb <= '0';
                fg_enb <= '0';
                memrd_strt <= '0';
                sram_cs <= '0';

        when motion_detection1 =>
                --control
                if md_rdy = '0' then
                        syscontrol_ns <= motion_detection2;
                else
                        syscontrol_ns <= motion_detection1;
                end if;
                --signals
                pgm_strt <= '0';
                frm_st <= '0';
                md_strt <= '1';
                dsp_strt <= '0';
                fg_cont <= '0';
                sram_cont <= '1';
                bin_enb <= '0';
                fg_enb <= '0';
                memrd_strt <= '0';
                sram_cs <= '0';

        when motion_detection2 =>
                --control
                if md_rdy = '1' then
                        if mode1(1 downto 0) = "00" then
                                syscontrol_ns <= mem_readout1;
                        elsif mode1(1 downto 0) = "01" then
                                syscontrol_ns <= dsp_handoff;
                        elsif mode1(1 downto 0) = "11" then
                                if trigger = '1' then
                                        syscontrol_ns <= dsp_ha
ndoff;

                                else
                                        syscontrol_ns <= mem_re
adout1;

                                end if;
                        else
                                syscontrol_ns <= standby;
                        end if;
                else
                        syscontrol_ns <= motion_detection2;
```



```
                end if;
                --signals
                pgm_strt <= '0';
                frm_st <= '0';
                md_strt <= '0';
                dsp_strt <= '0';
                fg_cont <= '0';
                sram_cont <= '1';
                bin_enb <= '0';
                fg_enb <= '0';
                memrd_strt <= '0';
                sram_cs <= '0';

        when mem_readout1 =>
                --control
                if memd_done = '0' then-- dsp_on = '1' then

                        syscontrol_ns <= mem_readout2;
                else
                        syscontrol_ns <= mem_readout1;
                end if;
                --signals
                pgm_strt <= '0';
                frm_st <= '0';
                md_strt <= '0';
                dsp_strt <= '0';
                fg_cont <= '0';
                sram_cont <= '0';
                bin_enb <= '0';
                fg_enb <= '1';
                memrd_strt <= '1';
                sram_cs <= '0';

        when mem_readout2 =>
                --control
                if memd_done = '1' then
                        syscontrol_ns <= standby;
                else
                        syscontrol_ns <= mem_readout2;
                end if;
                --signals
                pgm_strt <= '0';
                frm_st <= '0';
                md_strt <= '0';
                dsp_strt <= '0';
                fg_cont <= '0';
                sram_cont <= '0';
                bin_enb <= '0';
                fg_enb <= '1';
                memrd_strt <= '0';
                sram_cs <= '0';

        when dsp_handoff =>
                --Control
                if dsp_rdy = '0' then
                        syscontrol_ns <= dsp_handoff_wait;
                else
                        syscontrol_ns <= dsp_handoff;
                end if;
                --Signals
                pgm_strt <= dsp_pgm_strt;
                frm_st <= dsp_frm_strt;
                md_strt <= '0';
                dsp_strt <= '1';
                fg_cont <= '0';
                sram_cont <= '0';
                bin_enb <= '0';
```

```
                        fg_enb <= '0';
                        memrd_strt <= '0';
                        sram_cs <= '1';


                when dsp_handoff_wait =>
                        --Control
                        if dsp_rdy = '1' then
                                syscontrol_ns <= mem_readout1;--standby;
                        else
                                syscontrol_ns <= dsp_handoff_wait;
                        end if;
                        --Signals
                        pgm_strt <= dsp_pgm_strt;
                        frm_st <= dsp_frm_strt;
                        md_strt <= '0';
                        dsp_strt <= '0';
                        fg_cont <= '0';
                        sram_cont <= '0';
                        bin_enb <= '0';
                        fg_enb <= '0';
                        memrd_strt <= '0';
                        sram_cs <= '1';

                when others =>
                        --control
                        syscontrol_ns <= standby;
                        --signals
                        pgm_strt <= '0';
                        frm_st <= '0';
                        md_strt <= '0';
                        dsp_strt <= '0';
                        fg_cont <= '0';
                        sram_cont <= '1';
                        bin_enb <= '0';
                        fg_enb <= '0';
                        memrd_strt <= '0';
                        sram_cs <= '0';

        end case;

end process;

md_st <= md_strt;
bin_en <= bin_enb;
fg_en <= fg_enb;
sram_con <= sram_cont;
fg_con <= fg_cont;


--_____________________________________________________________________
--DSP CONTROLLER
--_____________________________________________________________________


ardy <= ardy_int;
interupt <= interupt_int;

dsp_command <= mode1(0);
dsp_mem_data_in <= mem_data(11 downto 0);

dsp_controller : dsp_cont port map(clk,dsp_strt,dsp_rdy,dsp_pgm_strt,pg
m_done,dsp_frm_strt,dsp_command,dsp_cont_req,
                                   video,fst_pulse,line_pulse,dsp_
mem_addr,dsp_mem_data_out,dsp_mem_data_in,
                                   dsp_mem_we,dsp_mem_oe,dsp_addr,
dsp_data,interupt_int,a_re,a_oe,a_we,a_ce,ardy_int,dsp_state);
```



```
--_____________________________________________________________________
--ProCam Control
--_____________________________________________________________________

        ponresetb <= '1';
        resetb <= '1';

        --Frame Counter for Binning
        process(clk)
        begin
                if rising_edge(clk) then
                        --if reset = '1' then
                        --      bin_frm_cnt <= "1111111111111111111";
                        if fst_pulse = '1' then
                                bin_frm_cnt <= "01000000000000000000" ;--"111111
11110000000000";
                        elsif frm_done = '0'then
                                bin_frm_cnt <= bin_frm_cnt + 1;
                        end if;
                end if;
        end process;

        frm_done <= '1' when (bin_frm_cnt = "11111111111111111111") else '0';

        frmedge : edge port map(clk,framestart,fst_pulse);
        lineedge : edge port map(clk,linestart,line_pulse);


        --CONTROL SIGNALS
        linesync <= '1';
        framesync <= frame;
        pclk <= cycle(0);

        frame <= frm_st;

        --Register Values
        process(syscontrol_ps,integ_time)
        begin
                case syscontrol_ps is

                when fpa_powerup1 =>
                        reg0 <= x"00"&"00000000";
                        reg1 <= x"FB"&"11111001";--NO INTERLACE
                        reg2 <= x"F3"&"01000100";--subsample
                        reg3 <= x"FD"&"00000000";--powerup ADC
                        reg4 <= x"F4"&"00010111";--set ADC clock
                        reg5 <= x"80"&"10000000";--Integration Time
                        reg6 <= x"00"&"00000000";
                        reg7 <= x"00"&"00000000";

                when fpa_powerup2 =>
                        reg0 <= x"00"&"00000000";
                        reg1 <= x"FB"&"10001000";--NO INTERLACE
                        reg2 <= x"F3"&"01000100";--subsample
                        reg3 <= x"FD"&"00000000";--powerup ADC
                        reg4 <= x"F4"&"00010100";--set ADC clock
                        reg5 <= x"80"&"11111111";--integ_time&"0000";--"1111111
1";--Integration Time
                        reg6 <= x"00"&"00000000";
                        reg7 <= x"00"&"00000000";

                when fpa_powerdown1 =>
                        reg0 <= x"00"&"00000000";
                        reg1 <= x"FB"&"10001000";--NO INTERLACE
                        reg2 <= x"F3"&"01000100";--subsample
                        reg3 <= x"00"&"00000000";
                        reg4 <= x"00"&"00000000";
```

```
                        reg5 <= x"00"&"00000000";
                        reg6 <= x"00"&"00000000";
                        reg7 <= x"00"&"00000000";

                when fpa_powerdown2 =>
                        reg0 <= x"00"&"00000000";
                        reg1 <= x"E1"&"00000000";--bias current
                        reg2 <= x"E3"&"00000000";--bias current
                        reg3 <= x"FD"&"11111111";--powerdown ADC
                        reg4 <= x"E0"&"00000000";--main bias current
                        reg5 <= x"E4"&"00000000";--bias current
                        reg6 <= x"E8"&"00000000";--bias current
                        reg7 <= x"EA"&"00000000";--bias current

                when dsp_handoff_wait =>
                        reg0 <= x"00"&"00000000";
                        reg1 <= x"FB"&"10001000";--NO INTERLACE
                        reg2 <= x"F3"&"00000000";--subsample
                        reg3 <= x"FD"&"00000000";--powerup ADC
                        reg4 <= x"00"&"00000000";
                        reg5 <= x"00"&"00000000";
                        reg6 <= x"00"&"00000000";
                        reg7 <= x"00"&"00000000";


                when others =>
                        reg0 <= x"00"&"00000000";
                        reg1 <= x"00"&"00000000";
                        reg2 <= x"00"&"00000000";
                        reg3 <= x"00"&"00000000";
                        reg4 <= x"00"&"00000000";
                        reg5 <= x"00"&"00000000";
                        reg6 <= x"00"&"00000000";
                        reg7 <= x"00"&"00000000";

                end case;
        end process;
--_____________________________________
--SERIAL PROGRAMMER
--_____________________________________

        ser_pgmer : pgmer port map(cycle(0),reset,pgm_strt,pgm_done,pgm_clk,pgm
_stb,pgm_dat,reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7);

        ser_clk <= pgm_clk;
        ser_stb <= pgm_stb;
        ser_dat <= pgm_dat;



--_____________________________________________________________
--FRAME GRABBER
--_____________________________________________________________
        fg_data <= "0000"&video when (bin_enb = '1') else "0000"&memd_data(11 d
ownto 0);
        fg_line <= linestart when (bin_enb = '1') else memd_line;
        fg_frame <= frm_done when (bin_enb = '1') else memd_frame;
        fg_clk <= not clk;
        fg_in <= (others => 'Z');


        dsp33_enb <= '1';
        dsp15_enb <= '1';


                --Readout Memory
        process(clk)
```



```
        begin
                if rising_edge(clk) then
                        memd0 <= memrd_strt;
                        memd1 <= memd0;
                end if;
        end process;
        memd_pulse <= memd0 and (not memd1);

        process(clk)
        begin
                if rising_edge(clk) then
                        -- reset = '1' then
                        --fgdisp_cnt <= "11111111111111111111";
                        if memd_pulse = '1' then
                                memd_cnt <= "00000000000000000000";
                        elsif memd_done = '0' then
                                memd_cnt <= memd_cnt + 1;
                        --elsif tp_on = '1' then
                        --      fgdisp_cnt <= fgdisp_cnt + 1;
                        end if;
                end if;
        end process;

        memd_done <= '1' when (memd_cnt = "11111111111111111111") else '0';
        memd_addr <= memd_cnt(19 downto 0);
        memd_line <= '1' when (memd_cnt(8 downto 4) = "11111") else '0';
        memd_frame <= '1' when (memd_cnt(19 downto 13) = "1111111")else '0';

        --MEMORY INTERFACE
        process(clk)
        begin
                if rising_edge(clk) then
                        sram_cs1 <= sram_cs;
                end if;
        end process;


        mem_addr_int <= memd_addr when (sram_cs1 = '0') else dsp_mem_addr(19 do
wnto 0);
        mem_we_int <= '1' when (sram_cs1 = '0') else dsp_mem_we;
        mem_oe_int <= '0' when (sram_cs1 = '0') else dsp_mem_oe;
        mem_cs <= "1110" when (sram_cs1 = '0') else "11"&dsp_mem_addr(20)&(not
dsp_mem_addr(20));

        mem_addr <= mem_addr_int when (sram_cont = '0') else (others => 'Z');
        mem_we <= mem_we_int when (sram_cont = '0') else 'Z';
        mem_oe <= mem_oe_int when (sram_cont = '0') else 'Z';
        mem_data <= dsp_mem_data_out when (data_enb = '1') else (others => 'Z')
;

        data_enb <= dsp_mem_oe and (not sram_cont) and sram_cs1;



        --Test Pattern
        process(clk)
        begin
                if rising_edge(clk) then
                        if cycle_start = '1' then
                                if test_pat1 = x"AAAA" then
                                        test_pat1 <= x"5555";
                                else
                                        test_pat1 <= x"AAAA";
                                end if;
                        end if;
                end if;
```

```
	end process;



	memd_data <= "0000"&mem_data(11 downto 0);



	--Row Counter
	process(clk)
	begin
		if rising_edge(clk) then
			if line_pulse = '1' then
				met_row_cnt <= "000100000";
			elsif met_row_tc = '0' then
				met_row_cnt <= met_row_cnt + 1;
			end if;
		end if;
	end process;

	met_row_tc <= '1' when (met_row_cnt = "111111111") else '0';

	high_level <= video(11) and video(10) and video(9);
	low_level <= (not video(11)) and not(video(10)) and not(video(9));

	--High Level Counter
	process(clk)
	begin
		if rising_edge(clk) then
			if fst_pulse = '1' then
				hlev_cnt <= "00000";
			elsif high_level = '1' and met_row_tc = '0' and hlev_tc
 = '0' then
				hlev_cnt <= hlev_cnt + 1;
			end if;
		end if;
	end process;

	hlev_tc <= '1' when (hlev_cnt = "11111") else '0';

	--Low Level Counter
	process(clk)
	begin
		if rising_edge(clk) then
			if fst_pulse = '1' then
				llev_cnt <= "00000";
			elsif low_level = '1' and met_row_tc = '0' and llev_tc
= '0' then
				llev_cnt <= llev_cnt + 1;
			end if;
		end if;
	end process;

	llev_tc <= '1' when (llev_cnt = "11111") else '0';


	dec_int <= hlev_tc and (not llev_tc);
	inc_int <= llev_tc and (not hlev_tc);

	--Total Counters
	process(clk)
	begin
		if rising_edge(clk) then
			if fst_pulse = '1' then
				if dec_int = '0' then
					dec_cnt <= "00";
```



```
				elsif dec_tc = '0' then
					dec_cnt <= dec_cnt + 1;
				end if;
			end if;
		end if;
	end process;

	dec_tc <= '1' when (dec_cnt = "11") else '0';

	process(clk)
	begin
		if rising_edge(clk) then
			if fst_pulse = '1' then
				if inc_int = '0' then
					inc_cnt <= "00";
				elsif inc_tc = '0' then
					inc_cnt <= inc_cnt + 1;
				end if;
			end if;
		end if;
	end process;
	inc_tc <= '1' when (inc_cnt = "11") else '0';

	process(clk)
	begin
		if rising_edge(clk) then
			if fst_pulse = '1' then
				if dec_tc = '1' and tm_tc = '0' then
					integ_time <= integ_time + 1;
				elsif inc_tc = '1' and tm_ntc = '0' then
					integ_time <= integ_time - 1;
				end if;
			end if;
		end if;
	end process;

	tm_tc <= '1' when (integ_time = "1111") else '0';
	tm_ntc <= '1' when (integ_time = "0000") else '0';



--_____________________________________________________________
--MOTION
--_____________________________________________________________

	process(clk)
	begin
		if rising_edge(clk) then
			if md_strt = '1' then
				trigger_cnt <= "000";
			elsif motion = '1' then
				if trigger = '0' then
					trigger_cnt <= trigger_cnt + 1;
				end if;
			end if;
		end if;
	end process;

	trigger <= (trigger_cnt(0) and trigger_cnt(1) and trigger_cnt(2)) or ds
p_cont_req;

	--Detect Global Illumination Change


--_____________________________________________
```

```
-- TEST POINTS
--_____________________________________

        tp1 <= syscontrol_ps(0);
        tp2 <= syscontrol_ps(1);
        tp3 <= syscontrol_ps(2);
        tp4 <= syscontrol_ps(3);
        tp5 <= interupt_int;
        tp6 <= dsp_state(2);
        tp7 <= dsp_state(1);
        tp8 <= dsp_state(0);--dsp_mem_addr(0);

end Behavioral;
```

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity dspcont is
    Port (clk : in std_logic;
                dsp_strt : in std_logic;
                dsp_on : out std_logic;
                dsp_frame : out std_logic;
                dsp_fg : out std_logic;
                dsp_pgm_strt : out std_logic;
                pgm_done : in std_logic;
                state_out : out std_logic_vector(3 downto 0));
end dspcont;

architecture Behavioral of dspcont is


        signal dsp_sm_ps,dsp_sm_ns : std_logic_vector(3 downto 0);
        constant dsp_sm_standby : std_logic_vector(3 downto 0) := "0000";
        constant dsp_sm_imageconfig1 : std_logic_vector(3 downto 0) := "0001";
        constant dsp_sm_imageconfig2 : std_logic_vector(3 downto 0) := "0010";
        constant dsp_sm_video1 : std_logic_vector(3 downto 0) := "0011";
        constant dsp_sm_video2 : std_logic_vector(3 downto 0) := "0100";

        signal dsp_frm_reset,dsp_frm_tc,dsp_frm_inc : std_logic;
        signal dsp_frm_cnt : std_logic_vector(21 downto 0);
        signal dsp_totfrm_tc,dsp_totfrm_reset,dsp_totfrm_inc : std_logic;
        signal dsp_totfrm_cnt : std_logic_vector(4 downto 0);

        signal dsp_image_on : std_logic;
        signal dsp_video_strt : std_logic;
        --signal dsp_pgm_strt : std_logic;
        --signal pgm_done : std_logic;

        signal reset : std_logic;

        --SERIAL PROGRAMMER
        --component pgmer is
        --      Port (s_clk,reset : in std_logic;
        --              pgm_st : in std_logic;
        --              pgm_done : out std_logic;
        --              ser_clk : out std_logic;
        --              ser_stb : out std_logic;
        --              ser_dat : out std_logic;
        --              reg0 : in std_logic_vector(15 downto 0);
        --              reg1 : in std_logic_vector(15 downto 0);
        --              reg2 : in std_logic_vector(15 downto 0);
        --              reg3 : in std_logic_vector(15 downto 0);
        --              reg4 : in std_logic_vector(15 downto 0);
        --              reg5 : in std_logic_vector(15 downto 0);
        --                      reg6 : in std_logic_vector(15 downto 0);
        --              reg7 : in std_logic_vector(15 downto 0));
        --end component;

        --signal pgm_clk,pgm_stb,pgm_dat : std_logic;
        --signal reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7 : std_logic_vector(15
downto 0);


begin
```



```
state_out <= dsp_sm_ps;

 reset <= '0';

--State Register
process(clk)
begin
        if rising_edge(clk) then
                dsp_sm_ps <= dsp_sm_ns;
        end if;
end process;

process(dsp_sm_ns,dsp_strt,pgm_done,dsp_image_on)
begin
        case dsp_sm_ps is

        when dsp_sm_standby =>
                --control
                if dsp_strt = '1' then
                        dsp_sm_ns <= dsp_sm_imageconfig1;
                else
                        dsp_sm_ns <= dsp_sm_standby;
                end if;
                --signals
                dsp_on <= '0';
                dsp_pgm_strt <= '0';
                dsp_video_strt <= '0';

        when dsp_sm_imageconfig1 =>
                if pgm_done = '0' then
                        dsp_sm_ns <= dsp_sm_imageconfig2;
                else
                        dsp_sm_ns <= dsp_sm_imageconfig1;
                end if;
                --signals
                dsp_on <= '1';
                dsp_pgm_strt <= '1';
                dsp_video_strt <= '0';


        when dsp_sm_imageconfig2 =>
                --control
                if pgm_done = '1' then
                        dsp_sm_ns <= dsp_sm_video1;
                else
                        dsp_sm_ns <= dsp_sm_imageconfig2;
                end if;
                --signals
                dsp_on <= '1';
                dsp_pgm_strt <= '0';
                dsp_video_strt <= '0';

        when dsp_sm_video1 =>
                --control
                if dsp_image_on = '1' then
                        dsp_sm_ns <= dsp_sm_video2;
                else
                        dsp_sm_ns <= dsp_sm_video1;
                end if;
                --signals
                dsp_on <= '1';
                dsp_pgm_strt <= '0';
                dsp_video_strt <= '1';

        when dsp_sm_video2 =>
                --control
```

```
                        if dsp_image_on = '0' then
                                dsp_sm_ns <= dsp_sm_standby;
                        else
                                dsp_sm_ns <= dsp_sm_video2;
                        end if;
                        --signals
                        dsp_on <= '1';
                        dsp_pgm_strt <= '0';
                        dsp_video_strt <= '0';

                when others =>
                        dsp_sm_ns <= dsp_sm_standby;
                        dsp_on <= '0';
                        dsp_pgm_strt <= '0';
                        dsp_video_strt <= '0';

                end case;
        end process;



                --VIDEO COUNTERS
                --frame pulse generator
                process(clk)
                begin
                        if rising_edge(clk) then
                                if dsp_frm_reset = '1' then
                                        dsp_frm_cnt <= "00000000000000000000";
                                elsif dsp_frm_inc = '1' then
                                        dsp_frm_cnt <= dsp_frm_cnt +1;
                                end if;
                        end if;
                end process;

                dsp_frm_tc <= '1' when (dsp_frm_cnt = "11111111111111111111")
else '0';
                dsp_frm_reset <= (dsp_frm_tc and (not dsp_totfrm_tc)) or dsp_vid
eo_strt;
                dsp_frm_inc <= not dsp_frm_tc;

                dsp_totfrm_tc <= '1' when (dsp_totfrm_cnt = "11111") else '0';
                dsp_totfrm_reset <=     dsp_video_strt;
                dsp_totfrm_inc <= dsp_frm_tc and (not dsp_totfrm_tc);

                dsp_fg <= '1' when (dsp_frm_cnt(21 downto 19) = "111") else '0';
                dsp_frame <= '1' when (dsp_frm_cnt(21 downto 4) = "1111111111111
11111") else '0';
                dsp_image_on <= not dsp_totfrm_tc;

                --total frame counter
                process(clk)
                begin
                        if rising_edge(clk) then
                                if dsp_totfrm_reset = '1' then
                                        dsp_totfrm_cnt <= "10000";
                                elsif dsp_totfrm_inc = '1' then
                                        dsp_totfrm_cnt <= dsp_totfrm_cnt + 1;
                                end if;
                        end if;
                end process;


                --reg0 <= x"AAAA";
                --reg1 <= x"5555";
                --reg2 <= x"1234";
                --reg3 <= x"4567";
```



```
                --reg4 <= x"FFFF";
                --reg5 <= x"FAFA";
                --reg6 <= reg5;
                --reg7 <= reg4;
                --SERIAL TESTING
                --ser_pgmer : pgmer port map(clk,reset,dsp_pgm_strt,pgm_done,pgm
_clk,pgm_stb,pgm_dat,reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7);

                --ser_clk <= pgm_clk;
                --ser_stb <= pgm_stb;
                --ser_dat <= pgm_dat;


end Behavioral;
```

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity edge is
    Port (clk : in std_logic;
                sig : in std_logic;
                pulse : out std_logic );
end edge;

architecture Behavioral of edge is

        signal s0 : std_logic;
        signal s1 : std_logic;

begin

        process(clk)
        begin
                if rising_edge(clk) then
                        s0 <= sig;
                        s1 <= s0;
                end if;
        end process;

        pulse <= s0 and (not s1);


end Behavioral;
```

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity pgmer is
    Port ( s_clk,reset : in std_logic;
                pgm_st : in std_logic;
                pgm_done : out std_logic;
                ser_clk : out std_logic;
                ser_stb : out std_logic;
                ser_dat : out std_logic;
                reg0 : in std_logic_vector(15 downto 0);
                reg1 : in std_logic_vector(15 downto 0);
                reg2 : in std_logic_vector(15 downto 0);
                reg3 : in std_logic_vector(15 downto 0);
                reg4 : in std_logic_vector(15 downto 0);
                reg5 : in std_logic_vector(15 downto 0);
                reg6 : in std_logic_vector(15 downto 0);
                reg7 : in std_logic_vector(15 downto 0));
end pgmer;

architecture Behavioral of pgmer is

        --counter for testing
        signal test_count : std_logic_vector(9 downto 0);

        --PROCAM SERIAL PROGRAMMING
        signal pgm_term,pgm_stop,ld_pgm_reg : std_logic;
        signal pgm_data,data_strb,pgm_strt : std_logic;
        signal pgm_cnt : std_logic_vector(7 downto 0);
        signal pgm_reg : std_logic_vector(15 downto 0);
        signal pgm_val : std_logic_vector(15 downto 0);
        --signal reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7 : std_logic_vector(15
downto 0);
        signal st0 : std_logic;

begin


        ------------------------------------------------------------------------
------------------------
        -- ProCam Control
        ------------------------------------------------------------------------
------------------------

        process(s_clk)
        begin
                if rising_edge(s_clk) then
                        st0 <= pgm_st;
                end if;
        end process;
        pgm_strt <= pgm_st and (not st0);


        --SERIAL PROGRAMMING


        process(s_clk)
        begin
                if rising_edge(s_clk) then
                        if reset = '1' then
```



```
                                pgm_cnt <= "11111111";
                        elsif pgm_strt = '1' then
                                pgm_cnt <= "00000000";
                        elsif pgm_stop = '0' then
                                pgm_cnt <= pgm_cnt + 1;
                        end if;
                end if;
        end process;

        process(s_clk)
        begin
                if rising_edge(s_clk) then
                        if ld_pgm_reg = '1' then
                                pgm_reg <= pgm_val;
                        elsif pgm_term = '0' then
                                pgm_reg(15) <= pgm_reg(14);
                                pgm_reg(14) <= pgm_reg(13);
                                pgm_reg(13) <= pgm_reg(12);
                                pgm_reg(12) <= pgm_reg(11);
                                pgm_reg(11) <= pgm_reg(10);
                                pgm_reg(10) <= pgm_reg(9);
                                pgm_reg(9) <= pgm_reg(8);
                                pgm_reg(8) <= pgm_reg(7);
                                pgm_reg(7) <= pgm_reg(6);
                                pgm_reg(6) <= pgm_reg(5);
                                pgm_reg(5) <= pgm_reg(4);
                                pgm_reg(4) <= pgm_reg(3);
                                pgm_reg(3) <= pgm_reg(2);
                                pgm_reg(2) <= pgm_reg(1);
                                pgm_reg(1) <= pgm_reg(0);
                                pgm_reg(0) <= '0';
                        end if;
                end if;
        end process;

        pgm_data <= pgm_reg(15);

        process(pgm_cnt,reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7)
        begin
                case pgm_cnt(7 downto 5) is
                        when "000" => pgm_val <= reg0;
                        when "001" => pgm_val <= reg1;
                        when "010" => pgm_val <= reg2;
                        when "011" => pgm_val <= reg3;
                        when "100" => pgm_val <= reg4;
                        when "101" => pgm_val <= reg5;
                        when "110" => pgm_val <= reg6;
                        when "111" => pgm_val <= reg7;
                        when others => pgm_val <= reg0;
                end case;
        end process;

        pgm_term <= pgm_cnt(4);
        pgm_stop <= pgm_cnt(0) and pgm_cnt(1) and pgm_cnt(2) and pgm_cnt(3) and
 pgm_cnt(4) and pgm_cnt(5) and pgm_cnt(6) and pgm_cnt(7);
        ld_pgm_reg <= pgm_cnt(4) and pgm_cnt(3) and pgm_cnt(2) and pgm_cnt(1) a
nd pgm_cnt(0);
        pgm_done <= pgm_stop;

        ser_clk <= not s_clk;

        ser_stb <= pgm_term;

        process(s_clk)
        begin
                if rising_edge(s_clk) then
                        ser_dat <= pgm_data;
```

```
                end if;
        end process;

        --END ProCam SERIAL PROGRAMMING

end Behavioral;
```

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity CPLD2 is
    Port (clk : in std_logic;
                --Memory Interface
                sram_data : inout std_logic_vector(11 downto 0);
                sram_addr : out std_logic_vector(19 downto 0);
                sram_we,sram_oe : out std_logic;
                sram_cs : out std_logic_vector(3 downto 0);
                --Frame Grabber Interface (over interconnect bus)
                fg_data : out std_logic_vector(15 downto 0);
                fg_line,fg_frame,fg_clk : out std_logic;
                fg_enb : out std_logic;
                par_clk,par_stb,par_dat : in std_logic;
                --CPLD interconnects
                md_strt : in std_logic;
                md_rdy : out std_logic;
                motion : out std_logic;
                sram_cont : in std_logic;
                fg_cont : in std_logic;
                bin_enb : in std_logic;
                fg_en : in std_logic;
                line : in std_logic;
                frame : in std_logic;
                video : in std_logic_vector(11 downto 0);
                --Power Enables
                fpa_enb,fpd_enb : out std_logic);
end CPLD2;

architecture Behavioral of CPLD2 is

        --BINNER
        component binner is
        Port (clk,enb : in std_logic;
                data : std_logic_vector(11 downto 0);
                frame, line : in std_logic;
                h_size, v_size : in std_logic_vector(5 downto 0);
                mem_data_in : in std_logic_vector(11 downto 0);
                mem_data_out : out std_logic_vector(11 downto 0);
                mem_addr : out std_logic_vector(19 downto 0);
                oe, we : out std_logic);
                --tp : out std_logic;
                --
                --add : out std_logic;
                --add_in1,add_in2 : out std_logic_vector(23 downto 0);
                --add_out : in std_logic_vector(23 downto 0);
                --cout : in std_logic);
        end component;

        signal bin_data : std_logic_vector(11 downto 0);
        signal b_frame, b_line : std_logic;
        signal h_size,v_size : std_logic_vector(5 downto 0);
        signal b_sram_data_in,b_sram_data_out : std_logic_vector(11 downto 0);
        signal b_sram_addr : std_logic_vector(19 downto 0);
        signal b_oe, b_we : std_logic;

        --MOTION DETECTOR
        component mot is
        Port (clk,reset : in std_logic;
```



```
                --Memory Interface
                sram_data_in : in std_logic_vector(11 downto 0);
                sram_data_out : out std_logic_vector(11 downto 0);
                sram_addr : out std_logic_vector(19 downto 0);
                sram_we,sram_oe : out std_logic;
                --CPLD interconnects
                md_strt : in std_logic;
                md_rdy : out std_logic;
                motion : out std_logic;
                col_max, row_max : in std_logic_vector(8 downto 0);
                shift_amt : in std_logic_vector(1 downto 0);
                thresh : in std_logic_vector(8 downto 0));
                --
                --add : out std_logic;
                --up_a,low_a : out std_logic_vector(16 downto 0);
                --output : in std_logic_vector(16 downto 0);
                --add_cout : in std_logic);
end component;

signal m_sram_data_in,m_sram_data_out : std_logic_vector(11 downto 0);
signal m_sram_addr : std_logic_vector(19 downto 0);
signal m_we,m_oe,m_rdy : std_logic;
signal col_max,row_max : std_logic_vector(8 downto 0);
signal shift_amt : std_logic_vector(1 downto 0);
signal motion_int : std_logic;
signal thresh : std_logic_vector(8 downto 0);

--Parameter programming
signal decode : std_logic_vector(7 downto 0);
signal pd : std_logic_vector(15 downto 0);
signal par_shift : std_logic;
signal valid1,valid2,valid3,valid : std_logic;
signal dval : std_logic_vector(1 downto 0);
signal par_clk_sync,par_stb_sync,par_dat_sync : std_logic;
signal pgm_stb1,pgm_stb2,pgm_stb3 : std_logic;
signal pgm_enb : std_logic;

--Connections
signal addr : std_logic_vector(19 downto 0);
signal oe,we : std_logic;
signal dat_in, dat_out : std_logic_vector(11 downto 0);
signal reset : std_logic;

--fpulse
signal fpulse0,fpulse1,fpulse : std_logic;

--adder
signal mot_add_in1,mot_add_in2 : std_logic_vector(16 downto 0);
signal bin_add_in1,bin_add_in2 : std_logic_vector(23 downto 0);
signal mot_add,bin_add : std_logic;
signal add_in1,add_in2 : std_logic_vector(23 downto 0);
signal add_output : std_logic_vector(23 downto 0);
signal add_add,add_cout : std_logic;

component ripp_add_20 is
Port (add : in std_logic;
    a : in std_logic_vector(23 downto 0);
        b : in std_logic_vector(23 downto 0);
        sum : out std_logic_vector(23 downto 0);
        cout : out std_logic);
end component;


--TEST PATTERN
signal testp_flip : std_logic;
signal video_in : std_logic_vector(11 downto 0);
signal test_pat1 : std_logic_vector(11 downto 0);
```

```
begin

        reset <= '0';


        --ADDER
        --adder : ripp_add_20 port map(add_add,add_in1,add_in2,add_output,add_c
out);

        --add_add <= mot_add when (bin_enb = '0') else bin_add;
        --add_in1 <= "0000000"&mot_add_in1 when (bin_enb = '0') else bin_add_in
1;
        --add_in2 <= "0000000"&mot_add_in2 when (bin_enb = '0') else bin_add_in
2;



        --BINNER

        bin : binner port map(clk,bin_enb,bin_data,b_frame,b_line,h_size,v_size
,b_sram_data_in,b_sram_data_out,b_sram_addr,b_oe,b_we);

        bin_data <= video;-- when (bin_enb = '1') else "000000000000";
        b_line <= line;-- when (bin_enb = '1') else '1';
        b_frame <= frame;-- when (bin_enb = '1') else '1';

        --MOTION DETECTOR

        md : mot port map(clk,fpulse,m_sram_data_in,m_sram_data_out,m_sram_addr
,m_we,m_oe,md_strt,m_rdy,motion_int,col_max,row_max,shift_amt,thresh);

        md_rdy <= m_rdy;

                --reset motion detector
                process(clk)
                begin
                        if rising_edge(clk) then
                                fpulse0 <= frame;
                                fpulse1 <= fpulse0;
                        end if;
                end process;
                fpulse <= (not fpulse0 and fpulse1);

        --Only allow motion detect while motion detector is on
        motion <= motion_int and (not m_rdy);



        --MEMORY CONNECTIONS
        addr <= b_sram_addr when (m_rdy = '1') else m_sram_addr;
        sram_addr <= addr when (sram_cont = '1') else (others => 'Z');

        sram_cs <= (others => 'Z');--"0111";-- when (sram_cont = '1') else (oth
ers => 'Z');

        oe <= b_oe when (m_rdy = '1') else m_oe;
        we <= b_we when (m_rdy = '1') else m_we;
        sram_oe <= oe when (sram_cont = '1') else 'Z';
        sram_we <= we when (sram_cont = '1') else 'Z';

        dat_in <= sram_data;
        m_sram_data_in <= dat_in;
        b_sram_data_in <= dat_in;

        dat_out <= b_sram_data_out when (m_rdy = '1') else m_sram_data_out; --b
_sram_data_ou
```



```
        sram_data <= dat_out when (sram_cont = '1' and oe = '1') else (others =
> 'Z');


        --ENABLES
        fpa_enb <= '1';
        fpd_enb <= '1';
        fg_enb <= not fg_en;

        --DATA PARAMETER LATCHES
        --Decoder

        with pd(11 downto 9) select
                decode <= "00000001" when "000", --hsize
                                "00000010" when "001", --vsize
                                "00000100" when "010", --col_max
                                "00001000" when "011", --row_max
                                "00010000" when "100", --shift_amt
                                "00100000" when "101", --threshold
                                --"01000000" when "110",
                                --"10000000" when "111",
                                "00000000" when others;


        --PARAMETER PROGRAMMING RECEIVER
        process(clk)
        begin
        if rising_edge(clk) then
            if  par_clk_sync = '1' then
                dval <= "00";
            elsif dval = "00" then
                dval <= "11";
            elsif dval = "11" then
                dval <= "01";
            else
                dval <= "01";
            end if;
        end if;
        end process;

        valid1 <= dval(0) and dval(1);

        process(clk)
        begin
                if rising_edge(clk) then
                        valid2 <= valid1;
                        valid3 <= valid2;
                end if;
        end process;

        valid <= valid3 and (not par_clk_sync);



        process(clk)
        begin
                if rising_edge(clk) then
                        par_dat_sync <= par_dat;
                        par_stb_sync <= par_stb;
                        par_clk_sync <= par_clk;
                end if;
        end process;


        process(clk)
```

```
begin
        if rising_edge(clk) then
                if valid = '1' then
                        pd(0) <= par_dat_sync;
                        pd(1) <= pd(0);
                        pd(2) <= pd(1);
                        pd(3) <= pd(2);
                        pd(4) <= pd(3);
                        pd(5) <= pd(4);
                        pd(6) <= pd(5);
                        pd(7) <= pd(6);
                        pd(8) <= pd(7);
                        pd(9) <= pd(8);
                        pd(10) <= pd(9);
                        pd(11) <= pd(10);
                end if;
        end if;
end process;

--CATCH Strobe to program registers
process(clk)
begin
        if rising_edge(clk) then
                pgm_stb2<= pgm_stb1;
                pgm_enb <= pgm_stb2;
        end if;
end process;
pgm_stb1 <= valid and par_stb_sync;



process(clk)
begin
        if rising_edge(clk) then
                if pgm_enb = '1' and decode(0) = '1' then
                        h_size <= "11"&pd(3 downto 0);--"1100";
                end if;
        end if;
end process;

--process(clk)
--begin
--      if rising_edge(clk) then
--              if pgm_enb = '1' and decode(1) = '1' then
                        v_size <= h_size;--"11"&pd(3 downto 0);--"1100";
--              end if;
--      end if;
--end process;

--process(clk)
--begin
--      if rising_edge(clk) then
--              if pgm_enb = '1' and decode(2) = '1' then
                        col_max <= "111111111";--pgm_data(8 downto 0);
        --      end if;
--      end if;
--end process;

--process(clk)
--begin
--      if rising_edge(clk) then
--              if pgm_enb = '1' and decode(3) = '1' then
                        row_max <= "100000000";--pgm_data(8 downto 0);
--              end if;
--      end if;
--end process;
```



```
        --process(clk)
        --begin
        --      if rising_edge(clk) then
        --              if pgm_enb = '1' and decode(5) = '1' then
                                thresh <= "010000000";--pd(8 downto 0);
        --              end if;
        --      end if;
        --end process;

        process(clk)
        begin
                if rising_edge(clk) then
                        if pgm_enb = '1' and decode(4) = '1' then
                                shift_amt <= pd(1 downto 0);
                        end if;
                end if;
        end process;



end;
```

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity binner is
    Port (clk,enb : in std_logic;
                data : std_logic_vector(11 downto 0);
                frame, line : in std_logic;
                h_size, v_size : in std_logic_vector(5 downto 0);
                mem_data_in : in std_logic_vector(11 downto 0);
                mem_data_out : out std_logic_vector(11 downto 0);
                mem_addr : out std_logic_vector(19 downto 0);
                oe, we : out std_logic);
                --tp : out std_logic;
                --
                --add : out std_logic;
                --add_in1,add_in2 : out std_logic_vector(23 downto 0);
                --add_out : in std_logic_vector(23 downto 0);
                --cout : in std_logic);
end binner;

architecture Behavioral of binner is

        --DATA FLOW
        signal input_mux : std_logic_vector(23 downto 0);
        signal memd_h : std_logic_vector(17 downto 0);
        signal memd_v : std_logic_vector(23 downto 0);
        signal mem_h : std_logic_vector(17 downto 0);
        signal mem_v : std_logic_vector(11 downto 0);
        signal out_data : std_logic_vector(11 downto 0);
        signal out_addr,out_addr1 : std_logic_vector(19 downto 0);
        signal sel1, sel2 : std_logic;
        signal bin_stop : std_logic;

        --ADDER
        component ripp_add_20 is
        Port (add : in std_logic;
            a : in std_logic_vector(23 downto 0);
                b : in std_logic_vector(23 downto 0);
                sum : out std_logic_vector(23 downto 0);
                cout : out std_logic);
        end component;

        signal add_in1, add_in2 : std_logic_vector(23 downto 0);
        signal add_out : std_logic_vector(23 downto 0);
        signal add : std_logic;
        signal cout : std_logic;
        signal add_in2_p : std_logic_vector(23 downto 0);


        --MEMORY
        signal address : std_logic_vector(19 downto 0);

        --TIMING
        signal pre_col_tc,col_tc,row_tc,wrt: std_logic;
        signal col_load,row_load : std_logic;
        signal frame_pulse,line_pulse : std_logic;
        signal col_cnt,row_cnt : std_logic_vector(5 downto 0);
        signal f0,f1,l0,l1 : std_logic;
        signal tc_delay : std_logic;
        signal col_addr : std_logic_vector(8 downto 0);
```



```
        signal row_addr : std_logic_vector(8 downto 0);
        signal row_beg : std_logic;
        signal col_stop : std_logic;
        signal tcount : std_logic_vector(11 downto 0);--TESTING
        signal tdata : std_logic_vector(11 downto 0);--TESTING

begin


        --EDGE DETECTORS
        process(clk)
        begin
                if rising_edge(clk) then
                        f0 <= frame;
                        f1 <= f0;
                end if;
        end process;
        frame_pulse <= (not f0) and f1;
        --tp <= line_pulse;

        process(clk)
        begin
                if rising_edge(clk) then
                        l0 <= line;
                        l1 <= l0;
                end if;
        end process;
        line_pulse <= (not l0) and l1;

        --TIMING

        pre_col_tc <= '1' when (col_cnt = "111111") else '0';

        process(clk)
        begin
                if rising_edge(clk) then
                        col_tc <= pre_col_tc;
                        wrt <= col_tc;
                end if;
        end process;

        row_tc <= '1' when (row_cnt = "111111") else '0';
        row_beg <= '1' when (row_cnt = v_size) else '0';

        sel1 <= row_beg;
        sel2 <= col_tc;

        col_load <= line_pulse or pre_col_tc;
        row_load <= frame_pulse or (row_tc and line_pulse);


        process(clk)
        begin
                if rising_edge(clk) then
                        if col_load = '1' then
                                col_cnt <= h_size;
                        else
                                col_cnt <= col_cnt + 1;
                        end if;
                end if;
        end process;

        process(clk)
        begin
                if rising_edge(clk) then
                        if row_load = '1' then
                                row_cnt <= v_size;
```

```
                        elsif line_pulse = '1' then
                                row_cnt <= row_cnt + 1;
                        end if;
                end if;
        end process;


        process(clk)
        begin
                if rising_edge(clk) then
                        if line_pulse = '1' then
                                col_addr <= "000000000";
                        elsif pre_col_tc = '1' and col_stop = '0' then
                                col_addr <= col_addr + 1;
                        end if;
                end if;
        end process;

        col_stop <= '1' when (col_addr = "111111111") else '0';

        process(clk)
        begin
                if rising_edge(clk) then
                        if frame_pulse = '1' then
                                row_addr <= "000000000";
                        elsif row_tc = '1' and line_pulse = '1' and bin_stop =
'0' then
                                row_addr <= row_addr + 1;
                        end if;
                end if;
        end process;

        address <= "00"&row_addr&col_addr;
        bin_stop <= '1' when (row_addr = "111111111") else '0';



        --MEMORY INPUT DATA
        process(h_size,mem_data_in)
        begin
                --if h_size(7)= '0' then
                --      memd_h <= mem_data_in&"00000000";
                --elsif h_size(7 downto 6) = "10" then
                --      memd_h <= '0'&mem_data_in&"0000000";
                if h_size(5) = '0' then
                        memd_h <= mem_data_in&"000000";
                elsif h_size(5 downto 4) = "10" then
                        memd_h <= '0'&mem_data_in&"00000";
                elsif h_size(5 downto 3) = "110" then
                        memd_h <= "00"&mem_data_in&"0000";
                elsif h_size(5 downto 2) = "1110" then
                        memd_h <= "000"&mem_data_in&"000";
                elsif h_size(5 downto 1) = "11110" then
                        memd_h <= "0000"&mem_data_in&"00";
                elsif h_size(5 downto 0) = "111110" then
                        memd_h <= "00000"&mem_data_in&'0';
                else
                        memd_h <= "000000"&mem_data_in;
                end if;
        end process;

        process(v_size,memd_h)
        begin
                --if v_size(7)= '0' then
                --      memd_v <= memd_h"00000000";
                --elsif v_size(7 downto 6) = "10" then
                --      memd_v <= '0'&memd_h&"0000000";
```



```
                if v_size(5) = '0' then
                        memd_v <= memd_h&"000000";
                elsif v_size(5 downto 4) = "10" then
                        memd_v <= '0'&memd_h&"00000";
                elsif v_size(5 downto 3) = "110" then
                        memd_v <= "00"&memd_h&"0000";
                elsif v_size(5 downto 2) = "1110" then
                        memd_v <= "000"&memd_h&"000";
                elsif v_size(5 downto 1) = "11110" then
                        memd_v <= "0000"&memd_h&"00";
                elsif v_size(5 downto 0) = "111110" then
                        memd_v <= "00000"&memd_h&'0';
                else
                        memd_v <= "000000"&memd_h;
                end if;
        end process;

        with sel1 select
                input_mux <= memd_v when '0',
                                "00000000000000000000000000" when '1',
                                memd_v when others;

        with sel2 select
                add_in2_p <= add_out when '0',
                                input_mux when '1',
                                add_out when others;

        process(clk)
        begin
                if rising_edge(clk) then
                        add_in2 <= add_in2_p;
                end if;
        end process;

        --DATA FLOW
        add_in1 <= "000000000000"&data;



        --ADDER
        add <= '1';
        adder : ripp_add_20 port map(add,add_in1,add_in2,add_out,cout);



        --WRITE REGISTER
        process(h_size,add_out)
        begin
                if h_size(5) = '0' then
                        mem_h <= add_out(23 downto 6);
                elsif h_size(5 downto 4) = "10" then
                        mem_h <= add_out(22 downto 5);
                elsif h_size(5 downto 3) = "110" then
                        mem_h <= add_out(21 downto 4);
                elsif h_size(5 downto 2) = "1110" then
                        mem_h <= add_out(20 downto 3);
                elsif h_size(5 downto 1) = "11110" then
                        mem_h <= add_out(19 downto 2);
                elsif h_size(5 downto 0) = "111110" then
                        mem_h <= add_out(18 downto 1);
                else
                        mem_h <= add_out(17 downto 0);
                end if;
        end process;

        process(v_size,mem_h)
```

```
	begin
		if v_size(5) = '0' then
			mem_v <= mem_h(17 downto 6);
		elsif v_size(5 downto 4) = "10" then
			mem_v <= mem_h(16 downto 5);
		elsif v_size(5 downto 3) = "110" then
			mem_v <= mem_h(15 downto 4);
		elsif v_size(5 downto 2) = "1110" then
			mem_v <= mem_h(14 downto 3);
		elsif v_size(5 downto 1) = "11110" then
			mem_v <= mem_h(13 downto 2);
		elsif v_size(5 downto 0) = "111110" then
			mem_v <= mem_h(12 downto 1);
		else
			mem_v <= mem_h(11 downto 0);
		end if;
	end process;

	process(clk)
	begin
		if rising_edge(clk) then
			out_data <= mem_v;
			out_addr1 <= address;
			out_addr <= out_addr1;
		end if;
	end process;

	--MEMORY
	mem_data_out <= out_data;

	oe <= wrt;
	we <= ((not (wrt and (not clk))) or bin_stop) or (not enb);
	mem_addr <= out_addr when (wrt = '1') else address;


	--TESTING
	process(clk)
	begin
		if rising_edge(clk) then
			if frame_pulse = '1' then
				tcount <= tcount + 1;
			end if;
		end if;
	end process;

end Behavioral;
```

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity mot is
    Port (clk,reset : in std_logic;
                --Memory Interface
                sram_data_in : in std_logic_vector(11 downto 0);
                sram_data_out : out std_logic_vector(11 downto 0);
                sram_addr : out std_logic_vector(19 downto 0);
                sram_we,sram_oe : out std_logic;
                --CPLD interconnects
                md_strt : in std_logic;
                md_rdy : out std_logic;
                motion : out std_logic;
                col_max, row_max : in std_logic_vector(8 downto 0);
                shift_amt : in std_logic_vector(1 downto 0);
                thresh : in std_logic_vector(8 downto 0));
                --
                --add : out std_logic;
                --up_a,low_a : out std_logic_vector(16 downto 0);
                --output : in std_logic_vector(16 downto 0);
                --add_cout : in std_logic);
end mot;

architecture Behavioral of mot is

        --Motion Detection
        signal mot_cnt : std_logic_vector(2 downto 0);
        signal add_md,md_on : std_logic;
        signal md_we,md_oe : std_logic;
        signal md_addr : std_logic_vector(19 downto 0);
        signal md_data : std_logic_vector(11 downto 0);
                --state machine

        signal control : std_logic_vector(17 downto 0);
        --signal shift_amt : std_logic_vector(1 downto 0);
        signal present_state,next_state : std_logic_vector(3 downto 0);
        constant state0 : std_logic_vector(3 downto 0) := "0000";
        constant state1 : std_logic_vector(3 downto 0) := "0001";
        constant state2 : std_logic_vector(3 downto 0) := "0010";
        constant state3 : std_logic_vector(3 downto 0) := "0011";
        constant state4 : std_logic_vector(3 downto 0) := "0100";
        constant state5 : std_logic_vector(3 downto 0) := "0101";
        constant state6 : std_logic_vector(3 downto 0) := "0110";
        constant state7 : std_logic_vector(3 downto 0) := "0111";
        constant state8 : std_logic_vector(3 downto 0) := "1000";
        constant state9 : std_logic_vector(3 downto 0) := "1001";
        constant state10 : std_logic_vector(3 downto 0) := "1010";
        constant state11 : std_logic_vector(3 downto 0) := "1011";
        constant state12 : std_logic_vector(3 downto 0) := "1100";
        constant state13 : std_logic_vector(3 downto 0) := "1101";
        constant state14 : std_logic_vector(3 downto 0) := "1110";
        constant state15 : std_logic_vector(3 downto 0) := "1111";
                --control signals
        signal up0sel : std_logic;
        signal up0enb : std_logic;
        signal up1sel : std_logic;
        signal up1enb : std_logic;
        signal upshift : std_logic;
        signal lowsel : std_logic;
```



```
        signal lowenb : std_logic;
        signal lowshift : std_logic;
        signal cupenb : std_logic;
        signal clowenb : std_logic;
        signal cshift : std_logic;
        signal mdenb : std_logic;
        signal memsel : std_logic_vector(1 downto 0);
        signal wrtreg : std_logic;
        signal wrtenb : std_logic;
        signal nextpix: std_logic;
                --registers/muxes
        signal up0_in, up0_out : std_logic_vector(16 downto 0);
        signal up1_in, up1_out : std_logic_vector(16 downto 0);
        signal low_in, low_out,lshift : std_logic_vector(16 downto 0);
        signal cup_out : std_logic_vector(16 downto 0);
        signal add_out : std_logic_vector(16 downto 0);
                --comparator
        signal compup, complow : std_logic_vector(11 downto 0);
        signal mot,mot_reg : std_logic;

        signal cpld2_ps,cpld2_ns : std_logic_vector(1 downto 0);
        constant start : std_logic_vector(1 downto 0) := "00";
        constant load_counters : std_logic_vector(1 downto 0) := "01";
        constant motion_detection : std_logic_vector(1 downto 0) := "10";
        signal md_begin: std_logic;
        --signal col_max,row_max : std_logic_vector(8 downto 0);
        signal col_count,row_count : std_logic_vector(8 downto 0);

        signal col_reset, col_limit : std_logic;
        signal row_reset,row_inc,row_limit : std_logic;
        signal md_done : std_logic;

        --adder
        component ripp_add_21 is
                Port (add : in std_logic;
                a : in std_logic_vector(16 downto 0);
                        b : in std_logic_vector(16 downto 0);
                        sum : out std_logic_vector(16 downto 0);
                        cout : out std_logic);
        end component;
                --signals
        signal output,up_a,low_a : std_logic_vector(16 downto 0);
        signal add,add_cout : std_logic;
        signal upper,lower : std_logic_vector(16 downto 0);

        --SRAM
        signal mem_data_in,mem_data_out : std_logic_vector(11 downto 0);
        signal mem_oe,mem_we : std_logic;
        signal mem_addr : std_logic_vector(19 downto 0);


begin

-------------------------------------------------------------------------------
--
-- MOTION CONTROLLER
-------------------------------------------------------------------------------
--

        --State Register
        process(clk)
        begin
                if rising_edge(clk) then
                        --if reset = '1' then
                        --      cpld2_ps <= start;
                        --else
                                cpld2_ps <= cpld2_ns;
```

```
                --end if;
        end if;
end process;

--State Machine
process(cpld2_ps,md_strt,md_done)
begin
        case cpld2_ps is

                when start =>
                        --control
                        if md_strt = '1' then
                                cpld2_ns <= load_counters;
                        else
                                cpld2_ns <= start;
                        end if;
                        --signals
                        md_begin <= '0';
                        md_rdy <= '1';

                when load_counters =>
                        --control
                        cpld2_ns <= motion_detection;
                        --signals
                        md_begin <= '1';
                        md_rdy <= '1';

                when motion_detection =>
                        --control
                        if md_done = '1' then
                                cpld2_ns <= start;
                        else
                                cpld2_ns <= motion_detection;
                        end if;
                        --signals
                        md_begin <= '0';
                        md_rdy <= '0';

                when others =>
                        --control
                        cpld2_ns <= start;
                        --signals
                        md_begin <= '0';
                        md_rdy <= '0';
        end case;
end process;

------------------------------------------------------------------------------
--
-- INTERCONNECT BUS
------------------------------------------------------------------------------
--

------------------------------------------------------------------------------
------------------------
-- MOTION DETECTION
------------------------------------------------------------------------------
------------------------

--CPLD 1 INTERFACE
        --_____________________________________________________________________
________
        -- * MDSTRT is input from CPLD 1
        -- * MDRDY is output to CPLD 1
        -- * MDRDY is high when waiting to start, MDSTRT sets high to start, wi
th row and
        --      column bin count on the bus.
```



```
        -- * MDRDY goes low during motion detection
        --_____________________________________________________________________
________


--MOTION DETECTION
        --_____________________________________
        -- DETECTION PARAMETERS
        -- * Can be hardwired or programmable
        -- * Column and Row counts 9 bits each
        --_____________________________________

        --col_max <= "111111111"; -- determined by bin size
        --row_max <= "111111111"; -- determined by bin size
        --shift_amt <= "11"; -- sets learning coefficient

        --_____________________________
        --COUNTERS
        -- * Row counter and Column counter
        -- * Started by md_start signal
        -- * incremented by nextpix control signal from state machine
        --_____________________________


        --COLUMN
        process(clk)
        begin
                if rising_edge(clk) then
                        if col_reset = '1' then
                                col_count <= "000000000";
                        elsif nextpix = '1' then
                                col_count <= col_count + 1;
                        end if;
                end if;
        end process;

        col_reset <= md_begin or col_limit;

        col_limit <= not ((col_count(0) xor col_max(0)) or (col_count(1) xor co
l_max(1)) or
                          (col_count(2) xor col_max(2)) or (col_count(3) xor col
_max(3)) or
                          (col_count(4) xor col_max(4)) or (col_count(5) xor col
_max(5)) or
                          (col_count(6) xor col_max(6)) or (col_count(7) xor col
_max(7)) or
                          (col_count(8) xor col_max(8)));

        --ROW
        process(clk)
        begin
                if rising_edge(clk) then
                        if row_reset = '1' then
                                row_count <= "000000001";
                        elsif row_inc = '1' then
                                row_count <= row_count + 1;
                        end if;
                end if;
        end process;

        row_reset <= md_begin;

        row_inc <= col_limit and (not row_limit);

        row_limit <= not ((row_count(0) xor row_max(0)) or (row_count(1) xor ro
w_max(1)) or
                          (row_count(2) xor row_max(2)) or (row_count(3) xor r
```

```
ow_max(3)) or
                        (row_count(4) xor row_max(4)) or (row_count(5) xor r
ow_max(5)) or
                        (row_count(6) xor row_max(6)) or (row_count(7) xor r
ow_max(7)) or
                        (row_count(8) xor row_max(8)));

        md_done <= row_limit and col_limit;

        md_addr <= memsel&row_count&col_count;

        --_____________________________________________________________
        --STATE MACHINE
        -- *    State Machine to Control Motion Detection
        --_____________________________________________________________

        --STATE REGISTER
        process(clk)
                begin
                if rising_edge(clk)     then
                        present_state <= next_state;
                end if;
        end process;

        --STATE ASSIGNMENTS
        process(present_state,md_begin,md_done)
        begin
                case present_state is
                        when state0 =>
                                if md_begin = '1' then
                                        next_state <= state1;
                                else
                                        next_state <= state0;
                                end if;
                        control <= "0000000000000000";

                        when state1 =>
                                next_state <= state2;
                                control <= "0100000000000000";
                        when state2 =>
                                next_state <= state3;
                                control <= "0000000000000000";
                        when state3 =>
                                next_state <= state4;
                                control <= "0001001000000010000";
                        when state4 =>
                                next_state <= state5;
                                control <= "0000000001000000";
                        when state5 =>
                                next_state <= state6;
                                control <= "0011000100000000";
                        when state6 =>
                                next_state <= state7;
                                control <= "0000011000001000000";
                        when state7 =>
                                next_state <= state8;
                                control <= "0001000000000000";
                        when state8 =>
                                next_state <= state9;
                                control <= "0000000010000000100";
                        when state9 =>
                                next_state <= state10;
                                control <= "0000001000100110000";
                        when state10 =>
                                next_state <= state11;
                                control <= "1100000000000000";
                        when state11 =>
```



```
                                next_state <= state12;
                                control <= "0011000100010100010";
                        when state12 =>
                                next_state <= state13;
                                control <= "0000011000000010100";
                        when state13 =>
                                next_state <= state14;
                                control <= "0001000000000000000";
                        when state14 =>
                                next_state <= state15;
                                control <= "0000000010000000100";
                        when state15 =>
                                if md_done = '1' then
                                        next_state <= state0;
                                else
                                        next_state <= state1;
                                end if;
                                control <= "0000000000000011011";
                        when others =>
                                next_state <= state0;
                                control <= "0000000000000000000";
                        end case;
                end process;

        --CONTROL SIGNALS
        up0sel <= control(17);
        up0enb <= control(16);
        up1sel <= control(15);
        up1enb <= control(14);
        upshift <= control(13);
        lowsel <= control(12);
        lowenb <= control(11);
        lowshift <= control(10);
        add <= control(9);
        cupenb <= control(8);
        clowenb <= control(7);
        cshift <= control(6);
        mdenb <= control(5);
        memsel <= control (4 downto 3);
        wrtreg <= control(2);
        wrtenb <= control(1);
        nextpix <= control(0);

        md_we <= not (wrtenb and (not clk));
        md_oe <= wrtenb;


        --_____________________________________________________
        -- COMPUTATION
        --_____________________________________________________

        --ADDER
        --instantiation
        adder: ripp_add_21 port map(add,up_a,low_a,output,add_cout);

        up_a <= upper when (upper > lower) else lower;
        low_a <= lower when (upper > lower) else upper;

        --MUXES
        --mux upper0, make 21 bits
        up0_in <= "00000"&mem_data_in when (up0sel = '0') else "00000"&compup(1
1 downto 0);
        --mux upper1
        up1_in <= up0_out when (up1sel = '0') else low_out;
        --mux lower, make 21 bits
        low_in <= "00000"&mem_data_in when (lowsel = '0') else output;
```

```
--REGISTERS
--register upper0 (21 bits, non-shifting)
process(clk)
begin
	if rising_edge(clk) then
		if up0enb = '1' then
			up0_out <= up0_in;
		end if;
	end if;
end process;


--register upper1 (21 bits, shifting)
process(clk)
begin
	if rising_edge(clk) then
		if up1enb = '1' then
			up1_out <= up1_in;
		elsif upshift = '1' then
			if shift_amt = "00" then
				up1_out <= up1_out(15 downto 0)&'0';
			elsif shift_amt = "01" then
				up1_out <= up1_out(14 downto 0)&"00";
			elsif shift_amt = "10" then
				up1_out <= up1_out(13 downto 0)&"000";
			else
				up1_out <= up1_out(11 downto 0)&"00000";
			end if;
		end if;
	end if;
end process;

--set output as input to adder
upper <= up1_out;

--register lower (21 bits, shifting)

process(clk)
begin
	if rising_edge(clk) then
		if lowenb = '1' then
			low_out <= low_in;
		elsif lowshift = '1' then
			if shift_amt = "00" then
				low_out <= low_out(15 downto 0)&'0';
			elsif shift_amt = "01" then
				low_out <= low_out(14 downto 0)&"00";
			elsif shift_amt = "10" then
				low_out <= low_out(13 downto 0)&"000";
			else
				low_out <= low_out(11 downto 0)&"00000";
			end if;
		end if;
	end if;
end process;

--set output as input to adder
lower <= low_out;


--WRITE REGISTER
process(clk)
begin
	if rising_edge(clk) then
		if wrtreg = '1' then
			if shift_amt = "00" then
				md_data <= output(12 downto 1);
```



```
			elsif shift_amt = "01" then
				md_data <= output(13 downto 2);
			elsif shift_amt = "10" then
				md_data <= output(14 downto 3);
			elsif shift_amt = "11" then
				md_data <= output(16 downto 5);
			end if;
		end if;
	end if;
end process;


--COMPARATOR

--register on top input of comparator, shift down adder value
process(clk)
begin
	if rising_edge(clk) then
		if cupenb = '1' then
			--if shift_amt = "11" then
			--	compup <= output(16 downto 5);
			--elsif shift_amt = "10" then
			--	compup <= output(14 downto 3);
			--elsif shift_amt = "01" then
			--	compup <= output(13 downto 2);
			--elsif shift_amt = "00" then
			--	compup <= output(12 downto 1);
			--end if;
			compup <= output(11 downto 0);
		end if;
	end if;
end process;


--register on bottom input of comparator
process(clk)
begin
	if rising_edge(clk) then
		if clowenb = '1' then
			complow <= mem_data_in(10 downto 7)&"11111111";-
-"00"&thresh&'0';
		end if;
	end if;
end process;


--comparator
mot <= '1' when (compup > complow) else '0';


process(nextpix,clk)
begin
	if rising_edge(clk) then
		if md_begin = '1' then
			mot_cnt <= "000";
		elsif mot_cnt = "111" then
			mot_cnt <= mot_cnt;
		elsif nextpix = '1' then
			if mot = '1' then
				mot_cnt <= mot_cnt + 1;
			else
				mot_cnt <= mot_cnt;
			end if;
		else
			mot_cnt <= mot_cnt;
		end if;
	end if;
```

```
        end process;


        motion <= mot;

        process(clk)
        begin
                if rising_edge(clk) then
                        if md_begin = '1' then
                                mot_reg <= '0';
                        else
                                mot_reg <= mot_cnt(0) and mot_cnt(1) and mot_cnt
(2);
                        end if;
                end if;
        end process;

--_____________________________________________________________________
--MEMORY INTERFACE
--_____________________________________________________________________


        mem_we <= md_we;
        mem_oe <= md_oe;
        mem_addr <= md_addr;

        sram_oe <= mem_oe;
        sram_we <= mem_we;


        mem_data_out <= mot&"00000000000" when (memsel = "01") else md_data;
        mem_data_in <= sram_data_in;
        sram_data_out <= mem_data_out;

        sram_addr <= mem_addr;

end;
```

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ripp_add_20 is
    Port (add : in std_logic;
             a : in std_logic_vector(23 downto 0);
                b : in std_logic_vector(23 downto 0);
                sum : out std_logic_vector(23 downto 0);
                cout : out std_logic);
end ripp_add_20;

architecture Behavioral of ripp_add_20 is

        component full_adder
                port (func,ci : in std_logic;
                         a,b : in std_logic;
                         sum, co : out std_logic);
        end component;

        signal c : std_logic_vector(23 downto 1);
        signal nadd: std_logic;
begin

        nadd <= (not add);

        u0: full_adder port map(add,nadd,a(0),b(0),sum(0),c(1));
        u1: full_adder port map(add,c(1),a(1),b(1),sum(1),c(2));
        u2: full_adder port map(add,c(2),a(2),b(2),sum(2),c(3));
        u3: full_adder port map(add,c(3),a(3),b(3),sum(3),c(4));
        u4: full_adder port map(add,c(4),a(4),b(4),sum(4),c(5));
        u5: full_adder port map(add,c(5),a(5),b(5),sum(5),c(6));
        u6: full_adder port map(add,c(6),a(6),b(6),sum(6),c(7));
        u7: full_adder port map(add,c(7),a(7),b(7),sum(7),c(8));
        u8: full_adder port map(add,c(8),a(8),b(8),sum(8),c(9));
        u9: full_adder port map(add,c(9),a(9),b(9),sum(9),c(10));
        u10: full_adder port map(add,c(10),a(10),b(10),sum(10),c(11));
        u11: full_adder port map(add,c(11),a(11),b(11),sum(11),c(12));
        u12: full_adder port map(add,c(12),a(12),b(12),sum(12),c(13));
        u13: full_adder port map(add,c(13),a(13),b(13),sum(13),c(14));
        u14: full_adder port map(add,c(14),a(14),b(14),sum(14),c(15));
        u15: full_adder port map(add,c(15),a(15),b(15),sum(15),c(16));
        u16: full_adder port map(add,c(16),a(16),b(16),sum(16),c(17));
        u17: full_adder port map(add,c(17),a(17),b(17),sum(17),c(18));
        u18: full_adder port map(add,c(18),a(18),b(18),sum(18),c(19));
        u19: full_adder port map(add,c(19),a(19),b(19),sum(19),c(20));
        --WAS TO u19
        u20: full_adder port map(add,c(20),a(20),b(20),sum(20),c(21));
        u21: full_adder port map(add,c(21),a(21),b(21),sum(21),c(22));
        u22: full_adder port map(add,c(22),a(22),b(22),sum(22),c(23));
        u23: full_adder port map(add,c(23),a(23),b(23),sum(23),cout);

        --u20: full_adder port map(add,c(20),a(20),b(20),sum(20),cout);

end Behavioral;
```

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ripp_add_21 is
    Port (add : in std_logic;
              a : in std_logic_vector(16 downto 0);
                 b : in std_logic_vector(16 downto 0);
                 sum : out std_logic_vector(16 downto 0);
                 cout : out std_logic);
end ripp_add_21;

architecture Behavioral of ripp_add_21 is

        component full_adder
                port (func,ci : in std_logic;
                         a,b : in std_logic;
                         sum, co : out std_logic);
        end component;

        signal c : std_logic_vector(16 downto 1);
        signal nadd: std_logic;
begin

        nadd <= (not add);

        u0: full_adder port map(add,nadd,a(0),b(0),sum(0),c(1));
        u1: full_adder port map(add,c(1),a(1),b(1),sum(1),c(2));
        u2: full_adder port map(add,c(2),a(2),b(2),sum(2),c(3));
        u3: full_adder port map(add,c(3),a(3),b(3),sum(3),c(4));
        u4: full_adder port map(add,c(4),a(4),b(4),sum(4),c(5));
        u5: full_adder port map(add,c(5),a(5),b(5),sum(5),c(6));
        u6: full_adder port map(add,c(6),a(6),b(6),sum(6),c(7));
        u7: full_adder port map(add,c(7),a(7),b(7),sum(7),c(8));
        u8: full_adder port map(add,c(8),a(8),b(8),sum(8),c(9));
        u9: full_adder port map(add,c(9),a(9),b(9),sum(9),c(10));
        u10: full_adder port map(add,c(10),a(10),b(10),sum(10),c(11));
        u11: full_adder port map(add,c(11),a(11),b(11),sum(11),c(12));
        u12: full_adder port map(add,c(12),a(12),b(12),sum(12),c(13));
        u13: full_adder port map(add,c(13),a(13),b(13),sum(13),c(14));
        u14: full_adder port map(add,c(14),a(14),b(14),sum(14),c(15));
        u15: full_adder port map(add,c(15),a(15),b(15),sum(15),c(16));
        u16: full_adder port map(add,c(16),a(16),b(16),sum(16),cout);--c(17));
        --u17: full_adder port map(add,c(17),a(17),b(17),sum(17),c(18));
        --u18: full_adder port map(add,c(18),a(18),b(18),sum(18),c(19));
        --u19: full_adder port map(add,c(19),a(19),b(19),sum(19),c(20));
        --u20: full_adder port map(add,c(20),a(20),b(20),sum(20),cout);

end Behavioral;
```

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity full_adder is
    Port (func,ci : in std_logic;
                a,b : in std_logic;
                sum, co: out std_logic);
end full_adder;

architecture Behavioral of full_adder is

signal b1,x1 : std_logic;

begin

        b1 <= b when (func = '1') else (not b);
        x1 <= a xor b1;
        sum <= x1 xor ci;
        co <= (a and b1) or (ci and x1);


end Behavioral;
```