A Low-Cost Hybrid Vision System for Intelligent Cruise Control Applications

by

Mark Christian Spaeth

B.S. Electrical Engineering and Computer Science
The Ohio State University (1997)

B.S. Mathematics
The Ohio State University (1996)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

Author .....
Department of Electrical Engineering and Computer Science
May, 1999

Certified by .....
Dr. Hae-Seung Lee
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

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

A Low-Cost Hybrid Vision System for Intelligent Cruise Control Applications

by

Mark Christian Spaeth

Submitted to the Department of Electrical Engineering and Computer Science
on May, 1999, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

In recent years, automobiles have become increasingly computerized and varying degrees of intelligent control has been integrated into automotive systems. A natural extension of this trend is full intelligent and autonomous control of vehicle by onboard computer systems. This thesis presents the design, development, and construction of a low-cost, low-power vision system suitable for on-board automated vehicle systems such as intelligent cruise control. The apparatus leverages vision algorithms, simplified by a prescribed camera geometry, to compute depth maps in real-time, given the input from three imagers mounted on the vehicle. The early vision algorithms are implemented using Dr. David Martin's **ADAP** mixed signal array processor. The back-end algorithms are implemented in software on PC for simplicity, but could easily be implemented in hardware in a later design. The final apparatus was able to compute depth maps at a rate of 24 frames per second, limited only by the interrupt latency of the PC executing the algorithms.

Thesis Supervisor: Dr. Hae-Seung Lee
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Dr. Ichiro Masaki
Title: Visiting Researcher

Acknowledgments

I'd like to start by acknowledging my co-advisors Professor Hae-Seung Lee and Dr. Ichiro Masaki for inviting me to come to MIT for my graduate work. Both have aided my work immeasurably with their deep understanding of the field, keen insights into troubleshooting, and patience when I was hacking Linux kernel drivers, and none of us knew what I was doing.

Furthermore, I'd like to acknowledge the other teachers who have guided me no my way. We learn important things from everyone in our lives, so while my Ohio State professors and high school teachers may have had the most explicit impact on my technical education, friends, family, and acquaintances are no less important. Special thanks go, of course, to my family for their support and encouragement, and letting me get away with all I've gotten away with.

Though, I'm sure none of them will ever read this document, I'd like to acknowledge those people who have been my inspiration in pursuing electrical engineering, such as Steve Wozniak, Nolan Bushnell, Owen Rubin, and Eugene Jarvis. Kudos if you know who these people are, but I'm sure you've encountered their work.

Lastly, I'd like to thank the MIT MTL community in general, and my officemates in specific for allowing me to pursue my hobbies in the hallways of Building 39. Sure, it's messy, noisy, time consuming, and space consuming, but I think it makes MTL a better place. Have you played Atari today?

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past several years, there has been increasing research interest in intelligent vehicle systems. Looking towards the goal of autonomous vehicles, integrated sensor and control systems are needed for tasks such as lane following, obstacle detection and avoidance, and adaptive cruise control. While simple systems using radar for obstacle detection have appeared in high-end late-model vehicles, to date cost effective sensor systems have not reached commercial fruition.

This thesis describes a prototype for a vision-based system which computes depth maps in real-time for adaptive cruise control. In order to achieve a high frame rate and low power operation, the system employs Dr. David Martin's ADAP mixed signal array processor to perform the early vision processing. The late processing is performed on laptop computer, but the algorithms implemented are simple enough to facilitate a VLSI/ASIC realization. All of the integrated circuits used are off-the-shelf components to ensure a cost-effective solution, and are implemented in compatible technologies, so that multiple subsystems may be implemented on a single die in the future to lower the total cost of the integrated system.

1.1 Thesis Background

For the ultimate goal of intelligent and autonomous vehicles, knowledge of scene ahead is essential to the vehicle control algorithms. The most popular approaches for the acquisition of this data are scanning radar and vision systems.

Radar systems scan the space in front of the vehicle for reflections. The position of obstacles is determined by the angle of the transmitter and the delay between the transmission and receipt of the signal. Some inaccuracies are inherent in the system, as granularity in the delay time corresponds to uncertainty in the computed distance. For a single radar-equipped autonomous vehicle in a simple environment, such a system is quite effective, but interference from multiple radar sources and indirect reflection paths in complex environments complicate the interpretation of the radar range data and limit the ultimate utility of radar systems. Additionally, since on standard roads lane lines are not discernible from the road surface using radar, lane following cannot be implemented.

Vision systems use one or more cameras to image the space in front of the vehicle. Using various machine vision algorithms, data such as object positions and relative velocities can be extracted from sequences or sets of images. Standard visible-light vision systems have may have trouble distinguishing features in low-light or adverse atmospheric conditions, but since human eyesight suffers under the same conditions, the user is more apt to

understand when the system might fail. In order to extract three-dimensional information such as depth, closing speed, or focus of expansion from sets of images, it is necessary to correlate features between the input images. For generalized camera geometries, the requisite coordinate transformations are quite computationally expensive, but by constraining the camera geometry, the algorithms may be greatly simplified and easily implemented in real-time software or in digital circuitry. Finally, since vision systems are inherently passive, there is no problem with interference from nearby sensors.

1.2 Thesis Motivation

Existing vision systems normally use an NTSC video camera and a frame buffer to capture images, and a complex digital signal processing (DSP) system to process sequences of images and extract information about the scene. While this approach is effective in a research environment, the cost, power consumption, and size of the requisite DSP preclude its use in on-board or embedded commercial applications.

Conventionally, integrated circuit imager chips were constructed using CCD technologies which had an inherent pixel serial output. This serial data format requires that the images be buffered in RAM before most algorithms can execute. Recent advances in VLSI technology have produced imagers fabricated in standard CMOS processes with random access pixel addressing, allowing algorithms to directly access data. Additionally, since the imager is implemented a standard CMOS process, the early vision processing can be implemented on the same die, reducing the system chip count and easing interfacing concerns.

In order to simplify the algorithms to extract information from the scene multiple cameras may be employed. Using standard machine vision algorithms, features may be correlated between images generated simultaneously from cameras at different positions in space. By placing the cameras in specific geometries, the computational complexity of the correlation routines may be simplified to a one dimensional search. For adaptive cruise control, the desired information is the distance to objects in the frame. This data may be derived by correlating edge positions between multiple images and using the disparities between the edge positions to compute the distance to the edge. To decrease the granularity of the distance measurements it is necessary to increase the resolution with which edge positions are discerned. Thus, a horizontal edge detection operator is followed by a sub-pixel edge resolution operator which locates edges at up to $\frac{1}{9}$ pixel accuracy [1].

This thesis will demonstrate a prototype of the system described above using existing components. Images will be acquired using a Photobit PB159-DM monochrome CMOS camera. The early vision processing algorithms described later will be implemented both in software and using Dr. David Martin's ADAP mixed signal array processor. The ADAP is a multiple instruction multiple data array processor, allowing arithmetic operations to be pipelined to increase the computational throughput. The ADAP employs mixed signal processing to effect power savings over pure digital hardware for multiplication and division operations. To avoid fabricating a custom digital processor for the back end data processing, the feature correlation routines will be implemented in software on a PC. The complete system computes depth maps in real time at approximately 24 frames per second using 64 by 64 pixel images. The speed of this prototype system is constrained by the interfaces between the disparate off-the-shelf components, so a future fully integrated implementation

[1] Due to limitations it the implementation, only $\frac{1}{4}$ pixel accuracy is achieved

should operate at much higher frame rates.

1.3 Thesis Organization

This chapter has introduced the motivation behind this thesis. Chapter 2 discusses the basic design of the system and motivates the selection of the components used in the implementation. Chapter 3 develops the machine vision algorithms used in this work, starting with basic vision theory. Chapter 4 describes the actual implementation of the hardware system. Chapter 5 details the integration of the hardware apparatus and the significant software component. Chapter 6 discusses the simulation and testing of the system and the associated results.

Appendix A describes how to set up, program, and operate the system. Appendix B contains source code for *img2edge*, *edg2spr*, and *spr2map* which were used to simulate the operation of the ADAP for algorithm testing. Appendix C describes the Linux kernel driver for the PCMCIA data acquisition card and how to interface to it. Appendix D contains the source code for the *progadap* and *progcam*, the ADAP and Camera programming routines. Appendix E contains the source for *xdepth* which computes the distance maps given the hardware generated sub-pixel resolved edge data. Appendix F includes schematics and layouts for the various PCBs used in the final implementation, including the schematic for the controller implemented in an Altera PLD.

Chapter 2

System Design

For a complete vision processing systems, the main components are an imager, image processors, and an output device. The block diagram for a simple vision system is shown in Figure 2-1.

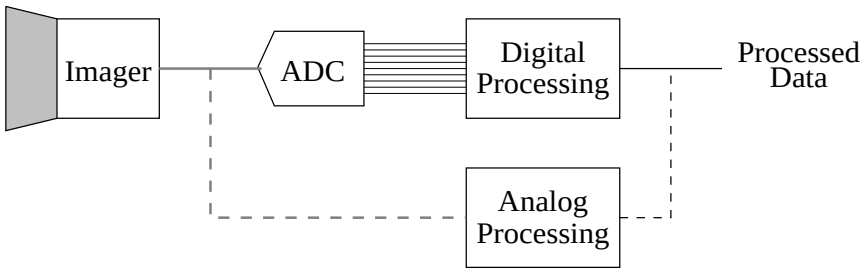

Figure 2-1: A simple machine vision system

Integrated circuit imager structures natively output analog data as a voltage, current, or charge packet. This value is normally passed through an analog-to-digital converter (ADC) to minimize the impact of noise when the data is passed off-chip to a processing unit. Alternatively, processing may be integrated onto the same chip as the imager, eliminating the need for the ADC. Traditionally, only dedicated special purpose chips have integrated the imager and processors on the same chip [1, 2]. This architecture allows the processor to access image data directly, avoiding interface circuitry that lowers the system bandwidth.

For a more generalized image processing system, it would be best to integrate a programmable processor onto the imager die. The fabrication of custom integrated circuits was beyond the scope of this work, so the design uses existing components that are designed in compatible processes. A simple block diagram for the system implemented for this thesis is given in Figure 2-2.

In order to simplify the design of the system, a laptop PC was chosen as the system and data output device. Through a data acquisition card and the parallel port, software the PC communicates with a digital controller board which generates the clocking and control signals necessary to configure and run the camera ICs and the early vision processing boards. The raw camera pixel data from the cameras may be routed to either the early vision processing PCB or back to the laptop, depend on how the algorithms are implemented. This chapter motivates the selection of the components used for the camera, digital controller, and early vision subsystems.

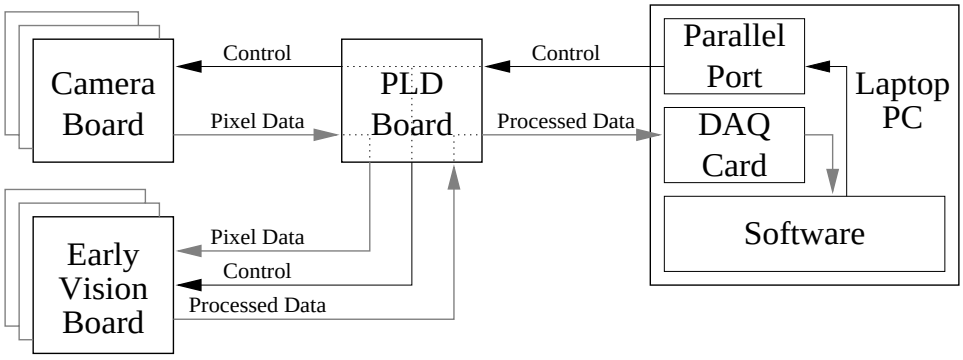

Figure 2-2: Vision processing system implementation

2.1 Imager Selection

While many commercial imagers are are implemented using charge-coupled devices (CCDs), recent research has resulted in imagers utilizing standard CMOS processing.

2.1.1 CCD Imagers

The enabling principle behind CCD imagers is the ability to transfer charge completely and efficiently between adjacent MOS capacitors by applying overlapping clocks to their gates [3]. A CCD imager array is shown in Figure 2-3. Photogates integrate charge at each pixel. The charge is transferred into vertical CCD shift registers through a transmission gate, and is shifted down into a horizontal CCD register to the output. At the CCD output, the charge packets are converted to a voltage or current quantity for output or further processing.

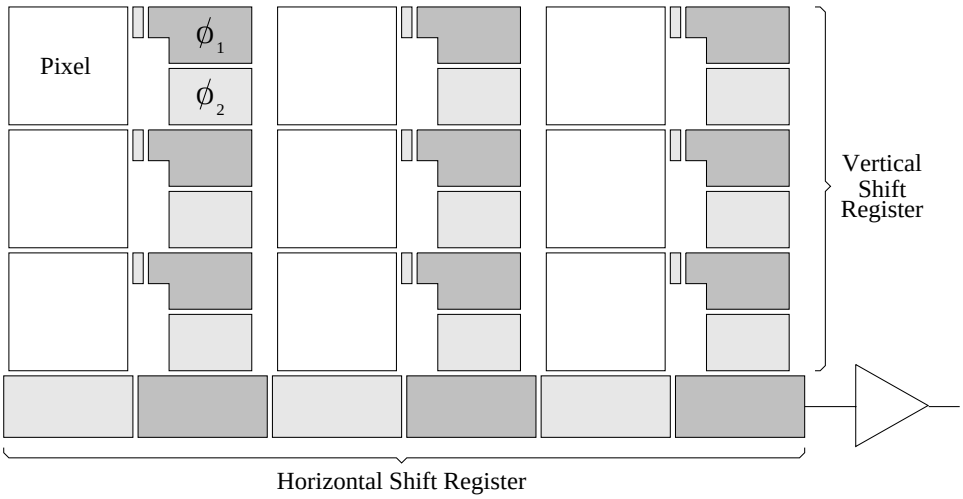

Figure 2-3: A CCD imager array

The CCD array imager has been dominant for many years due to their low read noise, small pixel size, high pixel fill factor, and extremely low fixed pattern noise. Due to the shift-register structure inherent in the design of a CCD array, the output is normally pixel-

serial, so buffer memory is needed to use a CCD imager with a processing algorithm which requires non-serial access to pixel data. While CCD signal processing circuits have been demonstrated [4], CMOS circuits are generally more flexible. CMOS circuits may be implemented along with CCDs, but the structures used to generated the CCDs require special semiconductor processing, making them more expensive than corresponding standard CMOS circuits.

2.1.2 CMOS Imagers

A typical CMOS imager array is shown in Figure 2-4. CMOS imagers use either a photogate or photodiode pixel structure, but the readout circuitry is similar to a ROM. When the row select line is enabled, the pixel charge is shared onto the column line capacitance, converting the charge from the pixel into a voltage on the column line. The column output logic then senses the line voltage, amplifies it, and transforms it into a voltage or current for output.

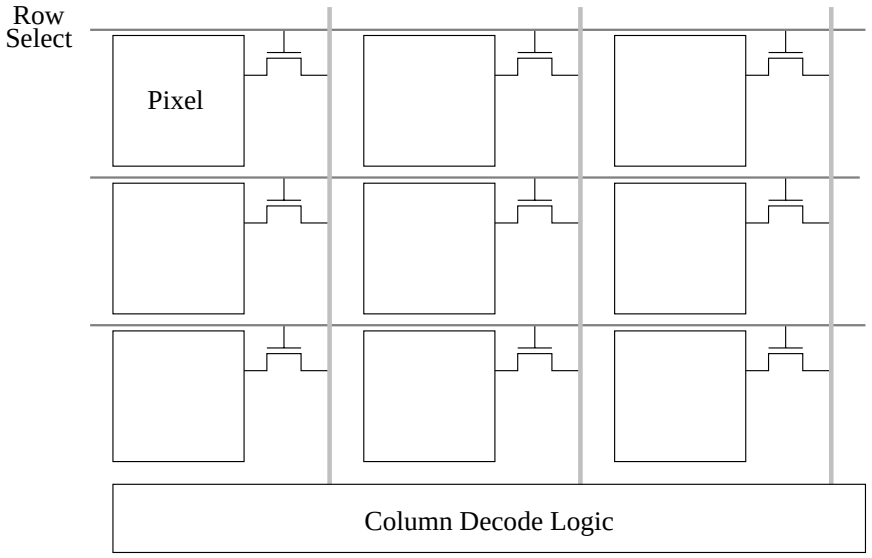

Figure 2-4: A CMOS imager array

MOS imagers normally have higher read nose due to the pass transistor and long column lines, but since they can be built in standard CMOS processes, the logic and interconnect portions of the pixels scale well with improved technologies. Additionally, both analog and digital processing circuitry can be inexpensively integrated onto the same die as the imager. Finally, adding an input column decoder, and adding an additional series transistor in each cell allows random access addressing of the pixels, allowing for more complex processing schemes without the need to buffer the image data.

2.1.3 PhotoBit PB159-DM

For this design, the PhotoBit PB159-DM monochrome CMOS imager was chosen. The complete specifications for this chip including the interface timing requirements can be obtained from the PhotoBit website at **http://www.photobit.com**. The imager is capable of output 30 frames per second at 512 x 384 resolution, while drawing less than 200mW

of power from a single 5V supply [5]. During the design phase of this research, the PhotoBit part was the only widely available general-purpose commercial part[1], but it has many features that are desirable for this application.

The PB159 has a rolling electronic shutter, which allows the pixel exposure to be synchronized to read-out circuitry, such that no additional exposure time is needed between frames. On-chip automatic exposure control, and programmable shutter widths allow the imager to perform well under all lighting conditions. The sensor is packaged in a 44-pin PLCC and requires only one supply, one clock, and six bias voltages, so it is simple to physically lay out on a printed circuit board.

Since the image processing cannot be integrated onto the sensor, the system has to accommodate the native imager output format. The PB159 has an on-chip ADC, to produce an 8-bit digital output. While the array itself is random access, the surrounding read circuitry outputs the pixel data serially. The output window is programmable, and the imager provides frame valid, line valid, and pixel valid signals to synchronize the external interface circuitry to the imager. Unfortunately, since the imager outputs pixels at a fixed 14.3MHz, so the data must be buffered before processing.

2.2 Vision Processing

The vision processing task can be broadly split into two parts: early vision processing and late processing. Early vision refers to front-end algorithms that process raw data using simple equations, while late processing involves more intelligence and decision-making.

Early vision algorithms are most efficiently implemented in dedicated custom circuitry, but programmable processors provide more flexibility and allow different algorithms to be tested on a the same hardware. For this thesis, the early vision algorithms are implemented in software on the PC for evaluation, but are implemented on a power efficient programmable mixed-signal array processor in the completed apparatus. For simplicity late processing is combined with the data output routines in software on the host PC.

2.2.1 Early Vision Processing – Software

The early vision algorithms were first written in C code on a PC for testing and evaluation. The evaluations were performed using synthesized calibrated images such as the road scene in Figure 2-5. Since the precise spatial positions of the rendered objects in the images are known, the output of the early vision routines can be compared with expected results. The rendered images were preferred over real images, as it is much simpler to create images with blur, noise, and imager misalignments than through rendering than by traditional photographic means. The software implementations of the algorithms were also incorporated into the final *xdepth* software, and tested using live images from the PhotoBit imagers using a digital controller board that bypassed the hardware early vision system.

[1] Other CMOS imagers, primarily aimed at markets, were available from other manufacturers, but they output standard video formats such as NTSC and CCIR, so much additional circuitry would have been necessary to extract the raw pixel values.

Figure 2-5: A calibrated test image

2.2.2 Early Vision Processing – ADAP

For the hardware implementation of the early vision algorithms, Dr. David Martin's ADAP programmable mixed-signal array processor was used [6]. The ADAP processor consists of a 5 by 5 array of locally interconnected functional cell. The ADAP is classified as a multiple-instruction multiple-data (MIMD) processor, as each cell operates independently on different data. The array was fabricated in a $0.8\mu m$ triple metal CMOS process, occupies $3mm^2$ of silicon, and consumes 53mW of power when operating at 3.63 MIPS.

Cell Architecture

As shown in Figure 2-6, each cell consists of an arithmetic unit, analog storage unit, switch matrix, and control unit. The ADAP uses a unique architecture, where the signal lines into the array and between array cells carry signals encoded as analog voltage which represents an 8-bit digital value.

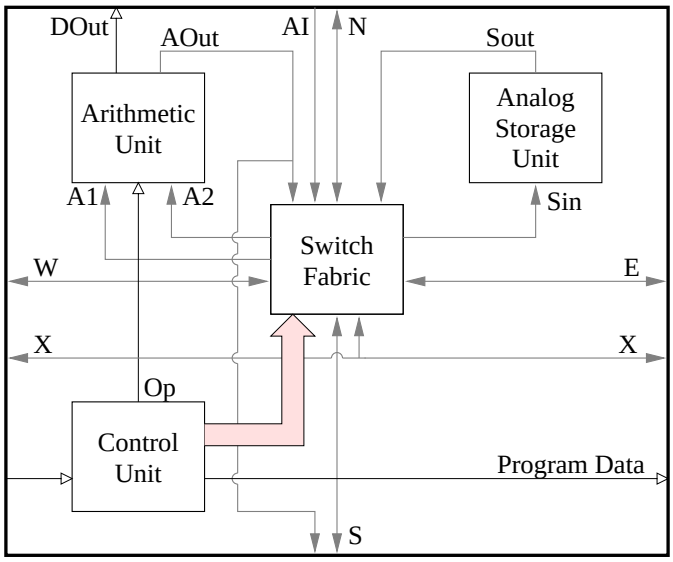

Figure 2-6: ADAP cell interconnections

Arithmetic Unit

The analog data format is very efficient, as it saves the extraneous interconnect needed to carry parallel digital signals or the additional clocking and registers needed to shift serial data. More importantly, analog signaling effects substantial power savings in the arithmetic unit. Digital circuitry can perform additions and subtractions efficiently, but the algorithmic computations required for multiplication and division require additional clock cycles. To achieve a desired data throughput, the system clock frequency must be raised. The ADAP utilizes the arithmetic unit diagrammed in Figure 2-7. Analog additions and subtractions can be efficiently performed using switched-capacitor circuits in the input sample-hold. The multiplication and division operations are performed by manipulating the reference voltages used in the analog to digital converter (ADC) and digital to analog converter (DAC). The ADC and DAC are implemented using cyclic topologies to minimize the silicon area for for given speed and power requirements. The D_{out} output from the shift register gives a serial digital output for the cells along the top of the array. These bit streams are the only viable array outputs, as it would be inefficient and impractical to have the analog drivers within the cells be capable of driving output pad capacitances at high speeds.

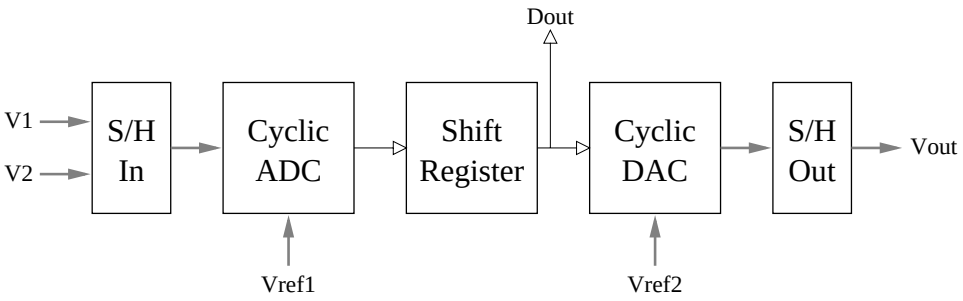

Figure 2-7: ADAP arithmetic unit

Storage Unit

The analog storage unit in each cell uses a sample-hold circuit to buffer data for one instruction cycle. Any arithmetic equation can be implemented using the ADAP, but the efficiency is very dependent on how the array is programmed. These buffers are paramount for efficiently implementing complex functions, as they allow the operation to be fully pipelined, so that new output values can be computed every clock cycle, with a finite latency.

Cell Interconnection

Data is transferred between cells via abutting signal lines (i.e. **S** in one cell and **N** in the cell immediately below it in the array). Data is sent into the array through the signal lines connected to the array cells at the extrema of the chip. Within the cell, the various cell input and output lines may be connected each other and with the structures within the cell though the switch matrix according to the rules in Table 2.1. After eliminating the redundant switches, 31 unique switches, implemented as NMOS pass transistors, are needed to construct the matrix.

Table 2.1: Available ADAP switch matrix connections

Signal	N	S	E	W	X	AOut	SOut	AI
N	-	X	X	X	-	X	X	-
S	X	-	X	X	-	X	X	-
E	X	X	-	X	-	X	X	-
W	X	X	X	-	-	X	X	-
SIn	X	X	X	X	-	X	-	-
A1	X	X	X	X	X	-	X	X
A2	X	X	X	X	-	-	X	-

Control Register

The control register is simply a 35-bit long shift register. The array is programmed by shifting bits into this register. The four outputs from the shift register are used to choose the operation performed by the arithmetic unit, and the other 31 are connected to the gates of the transistors in the switch matrix. The output of one control register is connected to the input of the register in the cell to its right, so a total of 165 bits are required to program each row of the array.

2.2.3 Back-End Processing

For this application, the output of the vision algorithm is a depth map image of the scene in front of the vehicle. On an integrated system, this image might be projected as part of a heads-up display, but this system uses a laptop PC to display the data. The back-end vision processing is incorporated into the PC display software for simplicity and to ease the design of the hardware system.

2.3 System Controller

Both the photobit imager and the ADAP arrays need precise clocking for programming and proper operation. While the PC is certainly powerful enough to generate these signals, it is more efficient to implement the clock generation in hardware. Due to the large number of clock signals and other interface logic needed to tie the imagers, ADAP arrays, and the PC together, programmable logic is used. The controller board utilizes an Altera EPM7128SQC100-10 programmable logic device. This device comes in a compact PQFP package, and is in-system programmable, allowing the clocking sequences to be easily changed and debugged for testing.

Chapter 3

Stereo Vision Algorithm

The hardware apparatus described in this thesis is suitable for many different vision processing tasks, but the intended application is intelligent cruise control. For this task, the key information that must be extracted from the image data is the distance to objects in the scene. Coupled with a lane tracking algorithm, this range information allows a controller to accurately discern the distance to objects directly in the intended path of the automobile, and an engine controller can modulate the vehicle's speed to keep this distance constant.

This chapter describes the basic algorithm used to extract the range data and the formulae which support the algorithm. Additionally, methods of increasing the robustness and resolution of the main algorithm will be discussed. For simplicity, these analyses assume that the camera optics may be reasonably modeled by a pinhole.

The complexity of the algorithms is constrained by the desire to produce a compact, low-power system. The use of the ADAP mixed-signal array processor, as motivated in the previous chapter, requires that the early-vision algorithms be implementable using only elementary arithmetic operations [6]. The integer arithmetic operations used in these algorithms also facilitates a low-power digital or an efficient software implementation. The back-end processing is not as constrained as it is performed in software, but is designed to be sufficiently simple to allow for a straightforward VLSI implementation.

3.1 Distance Calculations For Arbitrary Geometries

Given two cameras mounted with known geometries, and the position of a correlated feature in each image, the algorithm to discern the distance to the object from either camera is relatively simple. In Figure 3-1, suppose point **a** in image **A** is correlated with point **b** in image **B**. Projecting the points in the image planes through lens centers of the respective cameras produce the rays shown as dashed lines. The position of the feature in space is determined to be the point where the two projection rays intersect. Once this point is determined the distance from each camera's lens center to the object may be calculated.

Analytically, the coordinates of the feature in the image plane and the lens center of the camera are first transformed into an absolute coordinate space. These points are then used to parameterize the rays. Applying a least-squares minimization to the rays gives the position of the object in absolute coordinates. These procedures, while trivial to implement in software, are too computationally complex to execute for every feature correlation in the images when real-time performance is required.

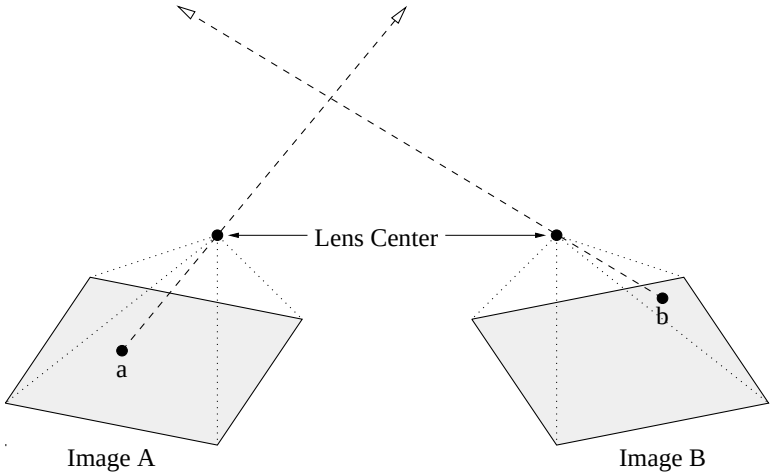

Figure 3-1: Distance calculation for arbitrary camera geometries

3.2 Correlation Search

The algorithm described above assumed that the image correlations were given, but in practice they may be derived. Given unlimited computing power, techniques such as motion estimation may be applied to the pair of images to produce accurate correlations. For a low-power system, a more computationally conservative solution is required.

3.2.1 Arbitrary Camera Geometries

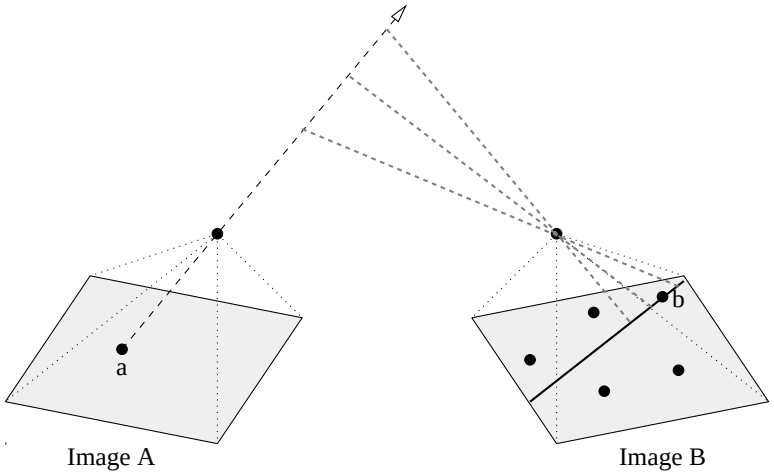

Figure 3-2: Epipolar constraint for arbitrary camera geometries

Consider Figure 3-2. Given feature **a** in image **A**, the algorithm must determine which features in image (B) are possible correlations. The locus of points in space that are imaged at **a** is given by the ray projected from **a** through the lens center of the camera. If this ray is projected into image **B**, the resulting line in the image plane, called the epipolar line, demonstrates which features in **B** may correspond with **a**. Thus the two-dimensional search

for possible correlations is reduced to one-dimension.

Analytically, the epipolar line may be calculated by intersecting the plane determined by **a** and the lens centers of the two cameras with the image plane of **B**. While the epipolar lines may be pre-calculated for a given camera geometry to reduce the run-time computational load of the system, the memory required to store this data would increase power consumption. A camera geometry which produces epipolar lines in the image **B** that are trivially related to the position of the feature in image **A** would be better.

3.2.2 Constrained Camera Geometry

Taking hints from nature in general, and the human vision system in particular, leads to a particularly convenient camera geometry. When the cameras are placed long a common horizontal baseline with aligned optical axes, the epipolar lines associated with any given point in image **A** are horizontal, as demonstrated in Figure 3-3.

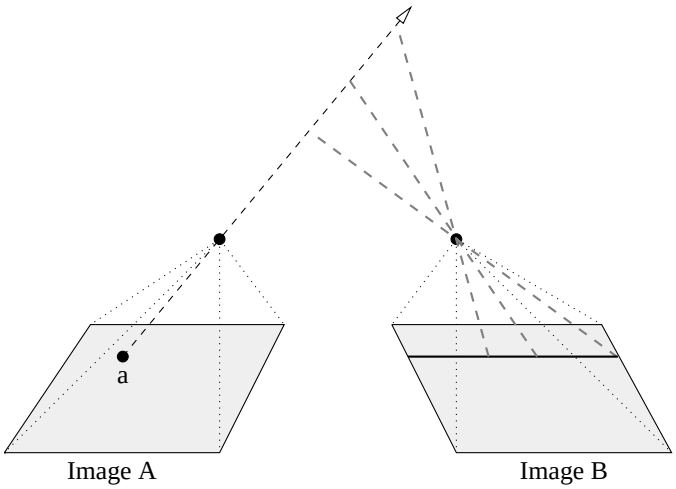

Figure 3-3: Epipolar constraint for a perpendicular geometry

A cursory analysis of this geometry reveals that the vertical location of the epipolar line in the second image is equal to the vertical location of the feature in the first image from which the line was derived. Conversely if a point is chosen on the epipolar line and used to derive a new epipolar line in image **A**, the derived epipolar locus will be a horizontal line passing through **a**. Thus, this geometry eliminates the computational complexity inherent in searching epipolar lines for correspondences in two ways: first, there is no need to compute coordinates of points on the epipolar line, since the entire line may be searched by fixing the vertical coordinate and incrementing the horizontal; second, since the epipolar lines are coincident for all features on a given horizontal line, it is possible to correlate entire lines together rather than correlating points individually.

3.3 Distance From Disparity

Now that a camera geometry which alleviates the complexity of correspondence searches has been determined, the algorithm to compute depths must be addressed. Since the optical axes of the cameras are aligned, the global coordinate system may aligned with

both cameras simultaneously, reducing the requisite coordinate transformation to a one-dimensional translation. This observation coupled with the epipolar constraint allows the distance to be calculated using triangle equations, as shown in Figure 3-4[1]

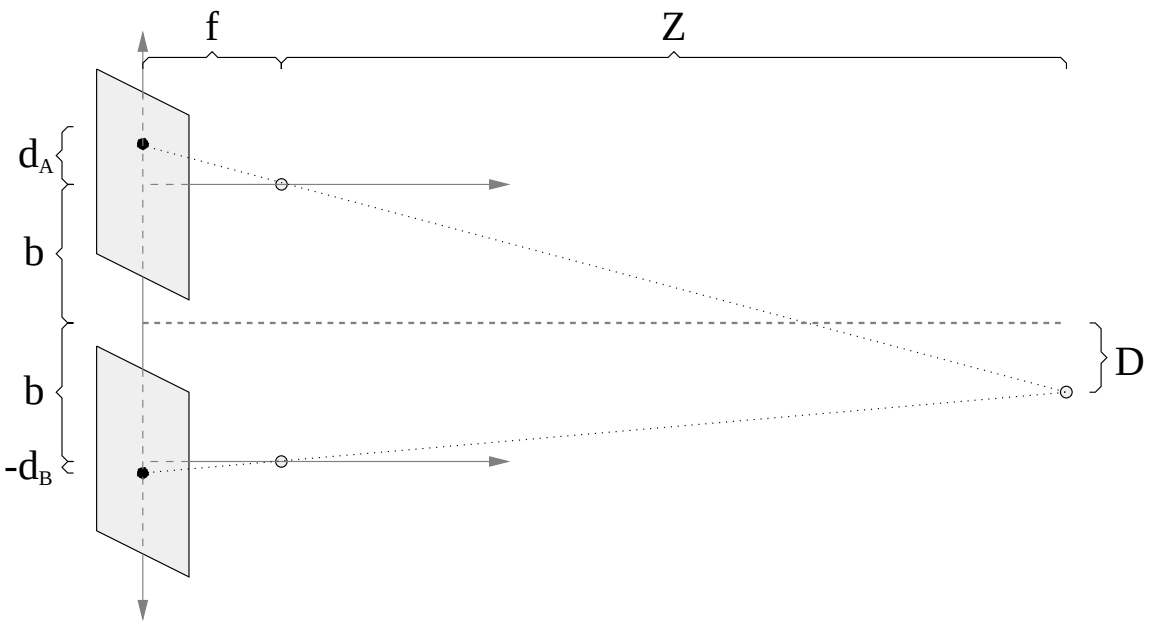

Figure 3-4: Distance from disparity

Analyzing similar triangles produces the following equations:

$$\frac{d_A}{f} = \frac{b + D}{Z} \tag{3.1}$$

$$-\frac{d_B}{f} = \frac{b - D}{Z} \tag{3.2}$$

In these equations, the known quantities are b, the distance each camera is horizontally displaced from the coordinate origin; f, the distance between the lens center and image plane for each camera; and d_A and d_B, the horizontal displacements of the features from the centers of projection for the respective images. Z, the distance from the lens centers to the object in space, and D, the horizontal displacement of the object from, the coordinate axis. Simply adding the above equations removes D from the equations, providing a simple analytic solution for the distance Z in terms of the disparity $d_A - d_B$ and the camera geometry parameters:

$$\frac{d_A - d_B}{f} = \frac{2 \cdot b}{Z} \tag{3.3}$$

$$Z = \frac{2 \cdot f \cdot b}{d_A - d_B} \tag{3.4}$$

[1] If the features in question are displaced vertically from the centerline of the image, an additional secant term must be applied to both f and Z, but since this term falls out in the analysis, it is omitted here.

3.4 Edge Detection

To this point, the features that are extracted from the image data and correlated have not been identified. Machine vision and image processing theory offer suggestions such as motion estimation, block pattern matching, image segmentation, and various forms of edge detection [7, 8], but given the computational requirements of the various algorithms, only certain edge detectors are suitable for implementation in an ADAP array.

For the purposes of computing horizontal axis disparities, a one-dimensional edge detector is preferable, as only vertical edges are useful to the algorithm. The feature correlation algorithm would be confused by predominantly horizontal edges, as show in Figure 3-5. Consider the dark line in space. If the entire dark line in each image is detected and correlated against the other line, then the entire shaded region will be erroneous returned as possible object locations in the depth map. A vertical edge detector would only find the endpoints of this line and reduce the ambiguity.

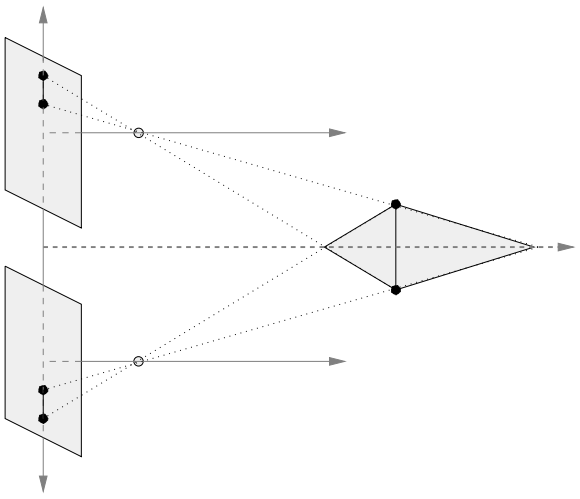

Figure 3-5: Depth ambiguity due to horizontal edges

Gradient based and Laplacian edge detectors [8] involve convolving a small mask with a portion of the image and thresholding the output to determine edge locations. For this application, the pixel data is streamed into the ADAP, so the edge detector only has a small number of adjacent pixel values available to it, so only a horizontal mask may be applied to the image. Additionally, absolute value and threshold operations cannot be implemented in the array, diminishing the utility of such techniques.

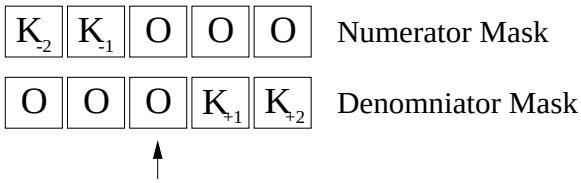

Figure 3-6: Sobel edge detection filter example

The edge detector implemented for this project is an example of a Sobel filter [9]. To

compute the positive edge strength at a pixel location, the filters shown in Figure 3-6 are applied to the surrounding pixels. The four filter constants may be adjusted to adjust for different contrast levels in the input images. The outputs of the two filters are then divided to determine the positive edge strength:

$$E_{x,+} = \frac{K_{-2} \cdot P_{x-2} + K_{-1} \cdot P_{x-1}}{K_{+1} \cdot P_{x+1} + K_{+2} \cdot P_{x+2}} \tag{3.5}$$

By reversing the filters, and inverting the division, a negative edge map may be computed similarly:

$$E_{x,-} = \frac{K_{-2} \cdot P_{x+2} + K_{-1} \cdot P_{x+1}}{K_{+1} \cdot P_{x-1} + K_{+2} \cdot P_{x-2}} \tag{3.6}$$

For both positive and negative edge equations, a pixel sequence with no gradient will produce a small constant value, while a strong edge will produce a large value. A gradient in the opposite direction yields a small fraction. For computing distance maps, both positive and negative edges should be considered to increase the number of points at which depths may be resolved. Adding the two previous equations yields a composite edge detector:

$$E_x = \frac{K_{-2} \cdot P_{x-2} + K_{-1} \cdot P_{x-1}}{K_{+1} \cdot P_{x+1} + K_{+2} \cdot P_{x+2}} + \frac{K_{-2} \cdot P_{x+2} + K_{-1} \cdot P_{x+1}}{K_{+1} \cdot P_{x-1} + K_{+2} \cdot P_{x-2}} \tag{3.7}$$

It should be noted that using the composite edge strength equation reduces the dynamic range of the edge measurement, as the noise-floor derived from constant pixel values doubles due to the addition, while the peak values remain the same.

3.5 Trinocular Stereo Analysis

As alluded to in the previous section, there is an ambiguity in the determination of the spatial location of features using the simple stereo correspondence algorithm. With as few as two features in an image, there may be multiple solutions to the binocular stereo correspondence problem, as exemplified in Figure 3-7.

If only the left and right images, labeled **L** and **R** respectively, are used, there are multiple solutions to the binocular correspondence problem. Projecting the image features back through the lens centers and intersecting the rays yields correspondences at points P_1, P_2, P_3, and P_4. Choosing either P_1 and P_2, or P_3 and P_4 leads to a minimal solution to the correspondence problem, but since there is no way to discern which is correct, all four entries must be added to the distance map. The addition of a third center image can resolve this problem. Projecting the feature points in the center image back reveals that the actual features appear at P_1 and P_2 [10].

In practice, however it is needlessly computationally expensive to project the features from the center image. Citing triangle equations again, it is clear that if a true correspondence exists between feature **r**, and feature **l**, there must exist a feature in image **C** at point **c**, located at half of the disparity between **r** and **l**. If a feature is found at this point, the correspondence is deemed true and an entry is computed for the depth map.

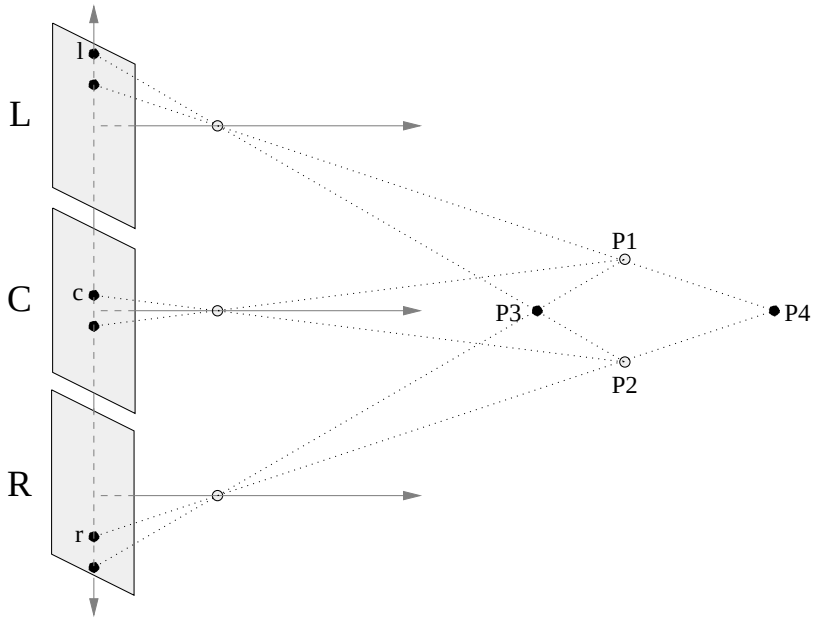

Figure 3-7: Stereo correspondence problem and resolution

3.6 Sub-Pixel Edge Resolution

One problem with the distance-from-disparity method is that the granularity of distances that may be discerned is closely related to the granularity at which the disparities can be measured. The algorithm as given calculates edge positions to single pixel accuracy, so the disparity measurement has a granularity of one pixel. To compute distances more accurately, one may either use a higher resolution imager to decrease the pixel size or implement an algorithm to compute edge positions to sub-pixel accuracy. Since increasing the imager resolution increases the output pixel data rate and required computational power quadratically, this option is not feasible.

There are a number of ways to extract sub-pixel resolved data, many of which can be performed using only the arithmetic operations available on the ADAP [11]. The most straight-forward sub-pixel algorithm was implemented in this system. This algorithm models the output of the edge detector with a parabola, as shown in figure 3-8, choosing the peak of the parabola as the sub-pixel edge position.

Given the equation for a generic parabola $E(x) = a \cdot x^2 + b \cdot x + c$, and three edge values $e(x)$, $e(x+1)$, and $e(x+2)$, it is simple to solve for the fitting parameters **a**, **b**, and **c**. From the modeling equation:

$$E(0) = a \cdot (0)^2 + b \cdot (0) + c = c = e(x) \tag{3.8}$$

$$E(1) = a \cdot (1)^2 + b \cdot (1) + c = a + b + c = e(x+1) \tag{3.9}$$

$$E(2) = a \cdot (2)^2 + b \cdot (2) + c = 4 \cdot a + 2 \cdot b + c = e(x+2) \tag{3.10}$$

Solving for a and b:

$$a = \frac{1}{4}\left(e(x) - 2 \cdot e(x+1) + e(x+2)\right) \tag{3.11}$$

$$b = \frac{1}{2}\left(-3 \cdot e(x) + 4 \cdot e(x+1) - e(x+2)\right) \tag{3.12}$$

Differentiating the parabola equation, to find the extrema:

$$x_{ext} = -\frac{b}{2 \cdot a} = \frac{-.75 \cdot e(x) + e(x+1) - .25 \cdot e(x+2)}{-.50 \cdot e(x) + e(x+1) - .50 \cdot e(x+2)} \tag{3.13}$$

Once x_{ext} is calculated, the edge position is discerned to a fraction of a pixel, and the disparities can be calculated with equal accuracy. x_{ext} is referenced to the current pixel position, so if $x_{ext} \leq 0$ or $x_{ext} > 2$ the value does not lie within the two-pixel range of consideration and is discarded. If the parabola to which the edge points are fit is concave down, the denominator term will be negative. Since the ADAP cannot represent negative values, it is clipped to zero, and the result of the division will clip at its maximum, so the result will automatically be discarded.

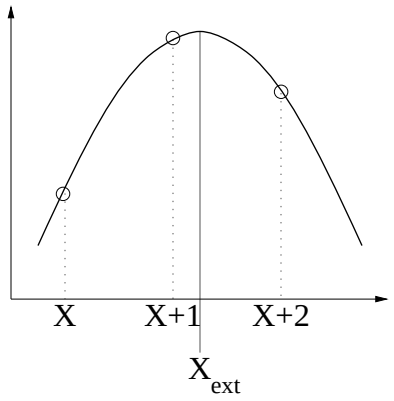

Figure 3-8: Sub-pixel resolution method

Chapter 4

Hardware Implementation

The complete hardware apparatus for the vision processing system requires a total of seven interconnected printed circuit boards, three of which are unique. The three imagers required by the algorithm are mounted on separate boards, each with its own support circuitry. Each imager has its own dedicated ADAP early vision board to perform the edge detection and sub-pixel edge resolution functions. The sub-pixel resolved data is sent to the common controller board, which interfaces to the PC. This chapter describes in detail the design of the three PCBs used in the system. The schematic for the each of the boards is given in the Appendix.

4.1 Imager Board

4.1.1 PB159 Interface

Due to the simple interface to the PB159DM chip, the imager PCB is the smallest and simplest PCB in the system. The PB159 is powered from a single 5V supply, that is generated on the PCB from the unregulated +12V supplied by the PLD board. The on-chip ADC and DAC circuitry requires several biases as listen in Table 4.1. These biases need not be exact, as they are only referenced by a set of column-parallel ADCs in, so they are created using simple resistor dividers with bypass capacitors [5].

Table 4.1: PB159 bias voltages

Name	Voltage	Purpose
VCL	2.50 V	ADC clamp bias
VLnDAC	1.16 V	DAC driver bias
VBiasN	1.16 V	ADC comparator bias
VLn	3.20 V	Column load bias
VLp	1.16 V	ADC source follower bias
VRef2	1.02 V	Noise correction DAC bias

The *Reset* signal for the imager is generated by low-pass filtering the supply. The PB159 also requires a single 14.318MHz clock used as the pixel exposure time and data output reference. The imager is programmed using a synchronous bidirectional serial interface, but the programming interface circuitry is incorporated onto the controller board.

4.1.2 Pixel Data Buffering

The PB159 outputs pixel data at a fixed 14.318MHz, but requires blanking periods between lines to allow the ADCs to settle, and between frames to recalibrate the converters and reset the pixels. These blanking signals are output to allow external devices to sync to the imager outputs. The early vision circuitry is not capable of processing data at this speed so the data must be buffered and sent to the processor at a lesser rate.

The data buffer was implemented using a dual ported SRAM. The vision algorithm expects 64 x 64 pixel frames, so a 4k SRAM is required. The $PixClk$, $LValid$, and $FValid$ signals are used to generate the addressing and control signals for the SRAM. The write cycle used for the SRAM is shown in Figure 4-1. The timing of the write cycle is controlled by $\overline{PixClk}$, which is connected to the $R/\overline{W}$ pin of the SRAM. The data output of the PB159 changes on the rising edge of $PixClk$ after a small propagation delay, so data is written on the falling edge of $PixClk$ when the data is valid. Since the RAM should only be written when the line and frame are valid, $\overline{LValid}$ is wired to the SRAM $\overline{CE}$ chip enable pin.

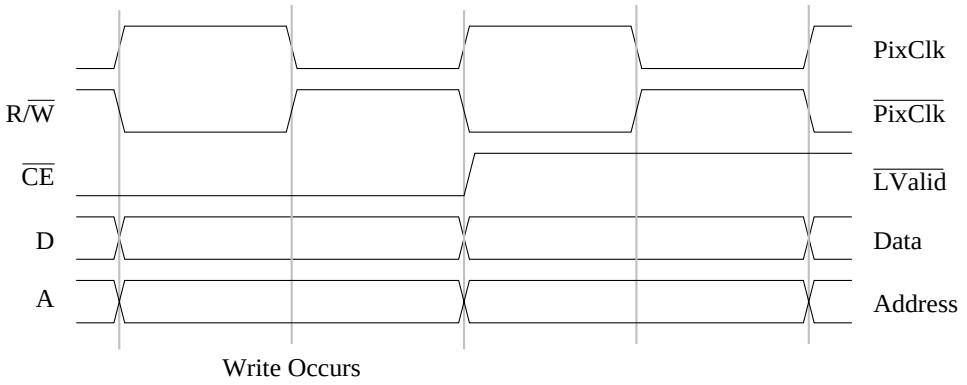

Figure 4-1: SRAM Write timing

The 12-bit address is decomposed into 6-bit row address and a 6 bit column address, each generated with a pair of 74164 counters. The column address is reset by the $LValid$ line valid signal from the PB159 and is clocked by $PixClk$. The row address is reset by the $FValid$ signal in is clocked by $LValid$ signal. The write timing is a bit tenuous since the $R/\overline{W}$ line falls as the address lines are still settling, so the previously written value could be corrupted, but this condition could be easily alleviated by delaying the $\overline{PixClk}$ that controls the write.

4.1.3 Imager PCB I/O

The imager PCB has a 20-pin IDC header for input and a 10-pin IDC header for output. The input connector carries the unregulated +12V to power the board, along with the addressing and control signals to read the pixel data out of the second port of the SRAM and the serial clock and data lines used to configure the imager. The output connector carries the 8-bit pixel data to the processing board. The read cycle uses the address changes to control the read, with the $\overline{CE}$ and $\overline{OE}$ always enabled.

The circuitry that controls the read address and the imager programming signals is implemented on the controller board and is discussed in a later section.

4.2 ADAP Early Vision Board

4.2.1 ADAP Interface

The ADAP is a research part rather than a commercial part, so the required bias and clock circuitry is more complex. The ADAP pinout is split roughly in half, with digital pins on one side and analog pins on the other. It requires 5V analog and digital supplies and has separate analog and digital grounds [1]. The analog and digital supplies are generated separately from the externally supplied unregulated 12V. The analog circuitry in the ADAP arithmetic unit requires several biases as listed in Table 4.2.

Table 4.2: ADAP bias voltages

Name	Voltage	Purpose
V+	0.945 V	ALU op-amp reference
VG	1.435 V	ALU zero voltage
VRef	2.459 V	ALU half-scale
VRef2	3.483 V	ALU full scale
VJ	2.459 V	ADC comparator reset voltage
V-	2.060 V	Interface DAC low reference

These biases are generated using low-noise op-amps in the unity-gain buffer configuration as shown in Figure 4-2. To minimize noise, tantalum and ceramic bypass capacitors are placed at the op-amp supply, non-inverting input, and output. Often a simple filter network is placed in the feedback path to prevent oscillation, but lab tests showed the op-amps to be unity gain stable up to 100Mhz, so the filter was omitted for simplicity. This voltage reference is also used to generate the analog constants required by the ADAP programs and in the external DAC circuitry.

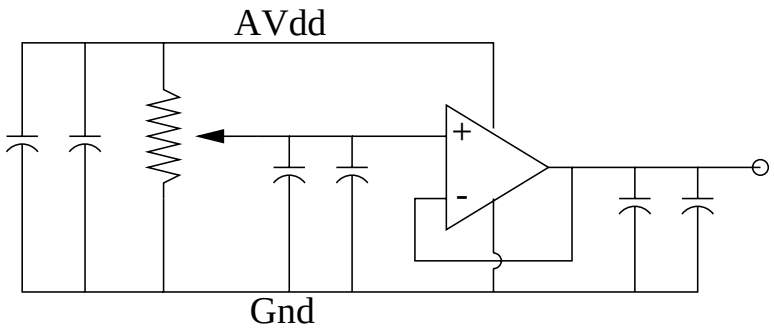

Figure 4-2: ADAP bias generation

The ADAP also requires a complex clocking as shown in Figure 4-3. The ADAP operates on a 10-phase instruction cycle. Φ_1 and Φ_2 are a set of two-phase non-overlapping clocks. VF and VL identify the first and last two phases in an instruction cycle respectively, and

[1] The supplies and grounds could be connected externally, but are separated to minimize digital noise coupling in the analog circuitry

may only change when both Φ_1 and Φ_2 are low. These clocking signals are generated by the controller board, and the circuitry used to generate them is discussed later.

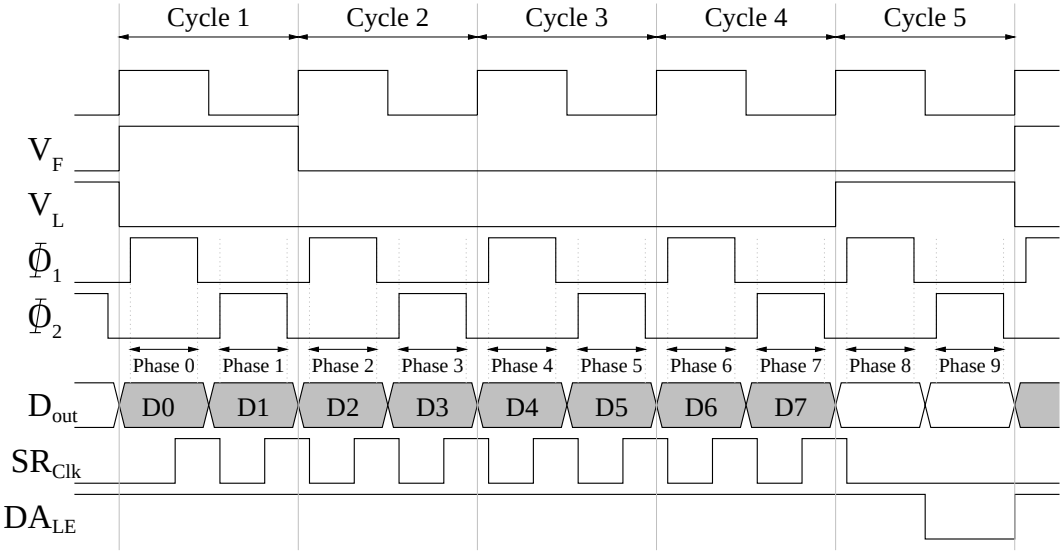

Figure 4-3: ADAP clocking

4.2.2 Data Conversion

The ADAP array expects analog inputs between VG and $VRef2$, but the output from the imagers is in 8-bit parallel format. Similarly, the ADAP output is in an 8-bit serial format. For these digital signals to be used as ADAP inputs, they must first be converted to the ADAP analog format.

To perform this conversion, the data is latched and is passed through an interpolating 8 bit DAC, as shown in Figure 4-4. To operate with 8-bit linearity, the DAC requires a swing near 3V, which is greater than the VG to $VRef2$ swing on the ADAP input, so the DAC is operated between a V_- reference and AV_{dd}, and passes through a stage of variable attenuation to achieve the appropriate levels. The potentiometer should be adjusted such that a 5V input yields an output at $Vref2$, and the $V-$ reference should be set to approximately $V_- = VG \cdot \frac{AVdd}{VRef2} \approx 2.060V$. To reduce noise, the non-inverting input and output of the output buffer are bypassed with small ceramic capacitors.

The DAC has a internal latch that is controlled by the DA_{LE} signal shown in Figure 4-3. When DA_{LE} is low, the transparently latch is enabled. To ensure proper timing and protect the DAC from glitches, the pixel data and ADAP outputs are latched beforehand. The ADAP data output is de-serialized using a serial-in parallel-out shift register that is clocked by SR_{Clk} The pixel data is repetitively and redundantly latched by SR_{Clk}.

4.2.3 ADAP Configurations

The ADAP programs to execute the edge detection and sub-pixel edge resolutions equations are shown in Figures 4-5 and 4-6 respectively. The physical locations of the interconnections in the diagrams indicate which wiring resources are used. The box in the upper right corner of each cell represents the analog storage unit, and the box in the upper left is the arithmetic

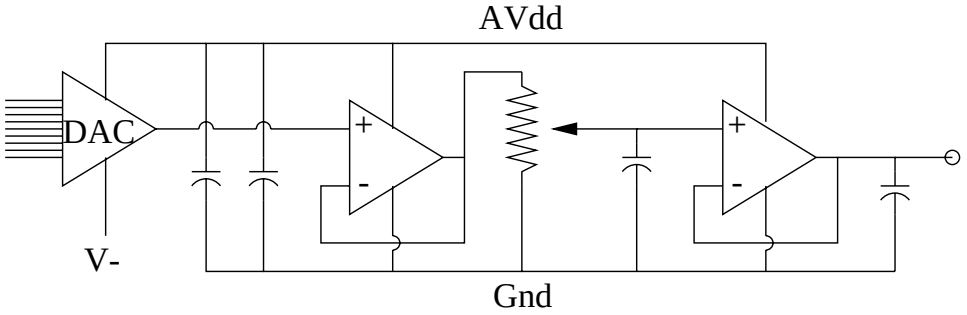

Figure 4-4: DAC interface circuitry

unit, with the operation noted. These equations require several analog constants which are generated in the same manner as the ADAP reference voltages. Table 4.3 lists the required analog constants.

Table 4.3: ADAP analog constants

Constant	Value	Purpose
9	1.497 V	Division 'K' constant
51 (.20)	1.843 V	Edge detection constant
153 (.60)	2.659 V	Edge detection constant
64 (.25)	1.947 V	SPER constant
128 (.50)	VRef	SPER constant
192 (.75)	2.971 V	SPER constant

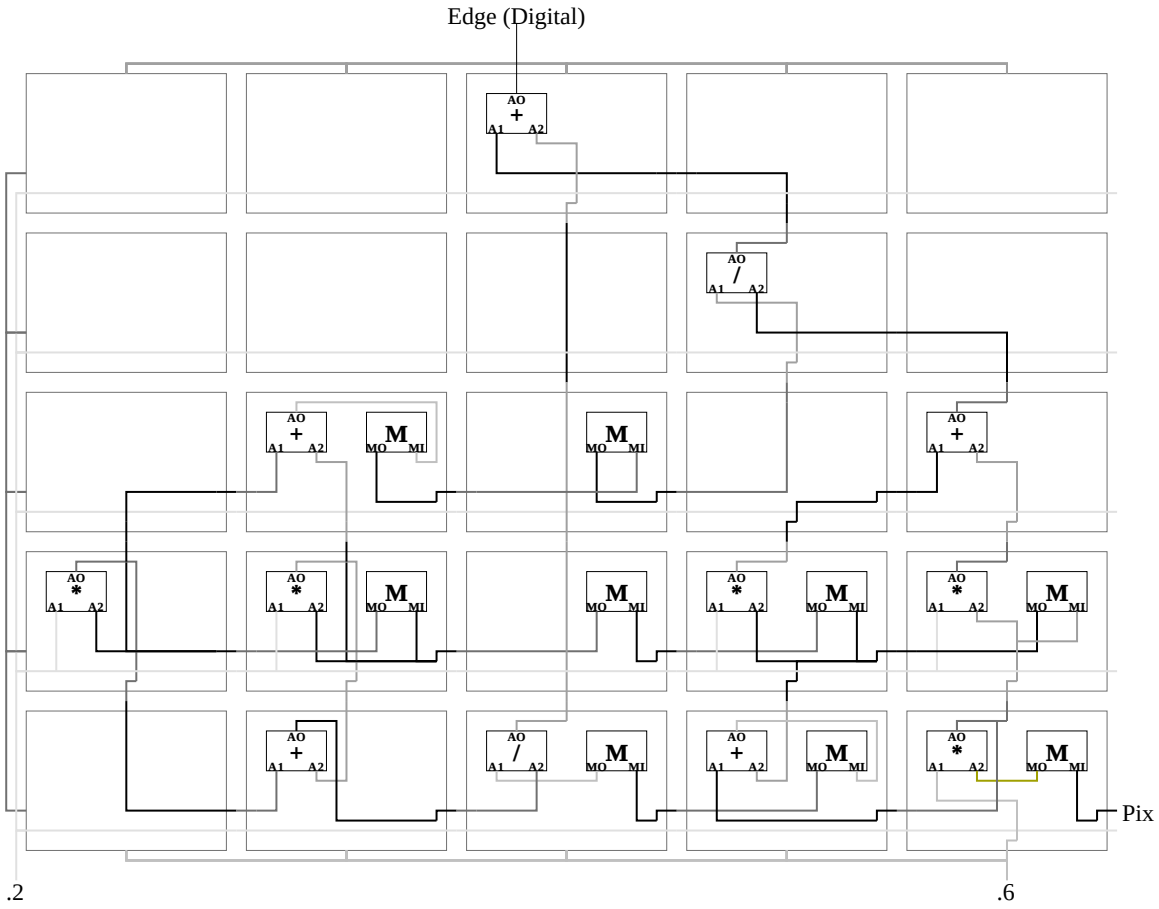

Figure 4-5: ADAP edge detection program

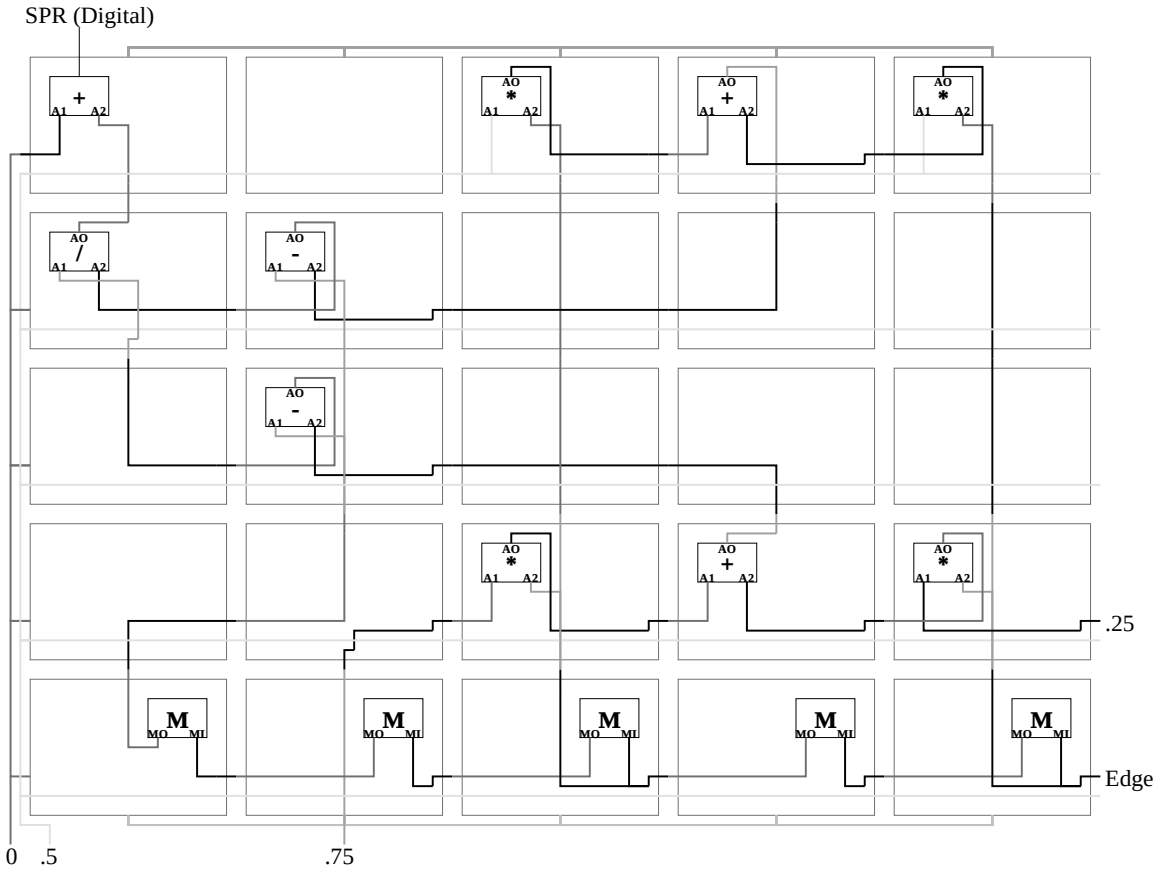

Figure 4-6: ADAP sub-pixel edge resolution program

4.3 PLD / Controller Board

The PLD controller board handles all of the clock generation and interfacing required to run the vision system. For the complete system, the pixel data from the imager board goes directly to the associated ADAP board, and the controller board sends the sub-pixel resolved edge data back to the PC. A test board which sends the raw pixel data directly to the PC was also fabricated. There is a significant amount of commonality between circuits programmed on the two PLDs, but the differences will be noted.

4.3.1 PLD Board I/O Ports

Both versions of the PLD board are dominated by interconnect and I/O ports. Table 4.4 lists all the connectors with their locations and purposes. The three 26-pin IDC headers appear at the left side only on the ADAP version of the board, and the three 10-pin IDC headers appear only on the non-ADAP version. The two switches at the bottom of the board are for power and program / run mode select.

Table 4.4: PLD board I/O ports

Type	Location	Purpose
DB37F	Right Side	Data acquisition card
IDC10	Top Right	Altera ISP programming adapter
IDC26	Top Center	PC parallel port
IDC20	Left (3)	Camera board control
IDC26	Left (3)	ADAP board control and data
IDC10	Left (3)	Pixel data
2-pin	Bottom	Power (+12V unregulated)

4.3.2 Clock Generation

The primary function unit on the controller board is the PLD itself. At the heart of the PLD is the clock generator circuitry for the ADAP. In both versions of the PLD, the ADAP clocking (previously shown in Figure 4-3) is used as the time basis for all functions. In a discrete implementation, the non-overlapping Φ_1 and Φ_2 clocks would be created from a master clock with circuitry similar to Figure 4-7. In this circuit, the length of the non-overlap period is controlled by the number of inverters in the delay chain, and can be increased by adding capacitance to the intermediate nodes.

Inside a PLD, however, this circuit cannot be used. Most PLD compilers will optimize out the delay chain, and node capacitances cannot be added. To resolve this problem, a higher multiple of the desired master clock is input and the delay is created digitally using counters. The circuit shown in Figure 4-8 uses a 32x master clock and creates a non-overlap between of $\frac{1}{16}$ of the clock period for Φ_1 and Φ_2. V_F and V_L are properly aligned between Φ phases.

The master clock counts up a 4-bit binary counter. The outputs of this counter are AND-ed together to generate the clock for a 4-bit BCD counter used to count the ten clock phases. Due to the AND, BCD counter is incremented when the count on the binary counter becomes 1111b. NOR-ing the high three bits of the BCD counter asserts VF on counts

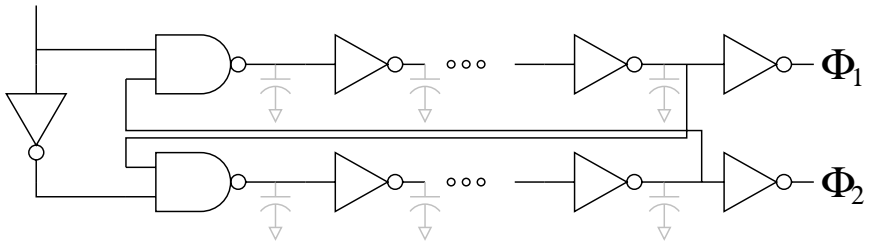

Figure 4-7: Discrete non-overlapping clock generator

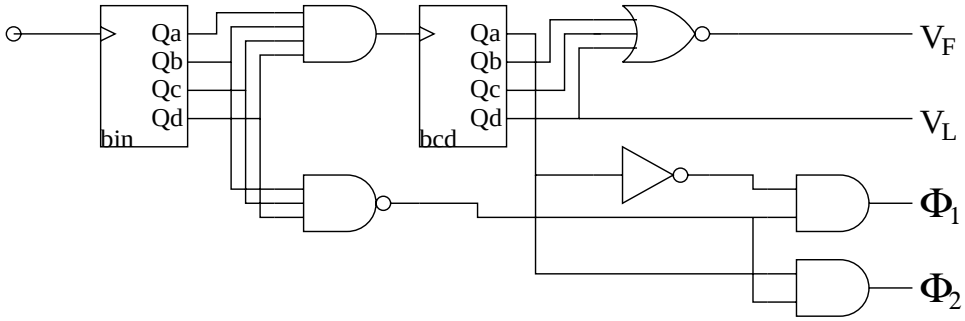

Figure 4-8: PLD clock generator

0000b and 0001b, and the high bit asserts VL directly on counts 1000b and 1001b. Φ_1 and Φ_2 are generated by AND-ing the low bit of the BCD counter with the NAND of the high three bits of the binary counter. This operations ensures that the Φ signals are asserted on counts 0000b through 1101b of the binary clock, yielding a $\frac{1}{16}$ period of non-overlap centered around the V_F and V_L transitions.

The ADAP board also requires the SR_{Clk} and DA_{LE} signals. The SR_{Clk} signal needs to clock the shift register once in each of the first eight clock cycles. To achieve this, the high bit of the binary counter is AND-ed with the a delayed and inverted version of the high bit from the BCD-counter. Due to the single master clock period timing advance of the BCD counter with respect to the binary counter, the BCD output must be delayed to prevent a glitch at very end of clock phase 9. The DA_{LE} signal needs to be asserted for a short period after the bit-shift has been completed. This is achieved by AND-ing the high and low bits of the BCD counter together, to assert the signal throughout clock phase 9.

4.3.3 Data Extraction

For proper data extraction, the controller board must signal the PC to read data when the data is valid. The PLD board connects the negative edge triggered interrupt line of the data acquisition card to V_L, so that the read is triggered by the falling edge of V_L. On the ADAP version of the PLD, the serial data is assembled in a shift register on-chip using the same SR_{Clk} that is sent to the ADAP board. The output of this shift register is then latched in a register using DA_{LE}. In the non-ADAP version, the data is sent to the PLD board earlier in the instruction cycle, so it can be latched on the rising edge of V_L.

To facilitate the transfer of data, the PLD board must also provide address signals to the imager board. For this purpose, a 12-bit counter is incorporated into the PLD. The enable

signal on the counter is cnotrolled as to prevent the counter from rolling over. The counter is clocked by V_F, so that new pixel data is delivered at the start of the instruction cycle and is reset and triggered by the $NewFrame$ input on the PLD. Thus, the PC software asserts $NewFrame$ at the beginning of the read sequence, reads and processes the data from the vision system, then retriggers when it is ready for more data.

4.3.4 ADAP Programming

The 24 digital I/O lines on the data acquisition card are normally used to read the three bytes of data back from the PLD board, but to program the ADAPs, 16 bits are used as outputs. When the mode switch on the PLD board is switched to program mode, the output latches in the PLD are tri-stated to allow the associated pins to be used as inputs. Since the timing is more critical than speed during programming, the master clock in the PLD is routed through an divide-by-16 counter to slow down all the clocks. The shift registers that hold the program information on the ADAP are controlled by Φ_2, so that signal must be used as the time reference for programming. When the PLD board is put into program mode, the PLD connects Φ_2 to the data acquisition card interrupt line.

Six data bits on channels A and B of the data acquisition card are used to transmit the ADAP program data to the PLD board. On each channel, the low five bits contain the serial bit-streams corresponding to the five rows in the ADAP array, and the sixth bit carries the program load signal. The twelve bits of data are latched into a register on the PLD and sent to the ADAP board on the rising edge of Φ_2. The program data is latched into the ADAP on the subsequent rising edge of Φ_2 and is held on the falling edge. The program load signal is asserted throughout the programming sequence and is de-asserted one cycle after the last bits of data are sent to ensure proper data alignment within the ADAP registers.

4.3.5 Imager Configuration

The PhotoBit imager is configured using a two wire synchronous bi-directional serial interface similar to the I^2C standard. The $SClk$ carries a reference clock, while $SData$ carries the data. To program the imager, a bit-stream is sent consisting of a start bit, followed by a byte containing the chip address on the I^2C bus, the register address to read or write, the data itself, and finally a stop bit. A start bit consists of a high to low transition on $SData$ while $SClk$ is high, while the stop bit is a low to high transition on $SData$ with $SClk$ High. All data bits must change only while $SClk$ is low and are sampled on the rising edge.

To meet the timing requirements of the timing protocol, the controller must be able to latch and write bits during both the high and low periods of $SClk$. In program mode, interrupts are triggered and data is written out on the falling edge of Φ_2. Φ_2 is then used to clock a D flip-flop to generate the $SClk$ signal. To normalize the timing, the data signal is latched on the PLD before being sent to the imagers. Referring back to Figure 4-3, the latch signal is generated by OR-ing the high bit of the binary counter with the low bit of the BCD counter. This produces a positive edge roughly halfway between the Φ_2 write signal and the rising edge of $SCLK$ when the bit must be valid.

Chapter 5

Software Integration

To this point, a hardware apparatus that generates sub-pixel resolved edge maps from a scene has been described. To complete the vision system PC software to acquired the edge maps and compute and output the distance maps is required. As programmable components were used in the design of the system, software to configure the imagers and ADAP arrays is also needed. The aforementioned programs interface to the hardware system through the data acquisition card and parallel port. Due to the high data throughput required by the system, custom low-level kernel drivers were implemented to facilitate this interface. This chapter describes much of the software created for the design, simulation, test, and final implementation of the vision system. The source code for many of the programs described here is given in the appendix.

5.1 Algorithm Simulation

As described in Chapter 2, the complete vision system uses software the correlated the sub-pixel resolved edges and compute the distance maps. During the design phase, a framework within which various algorithms could be evaluated was needed. The natural approach to this problem was to simulate the entire vision processing system in software, and migrate portions into hardware as the design progressed. Figure 5-1 shows the correlation between hardware elements and PC software. The ADAP board was designed such that the output from the either the first or second array could be sent back to the PLD board, allowing either edge data or sub-pixel resolved edge data to be sent back to the PC. A separate PLD board was constructed to interface the imagers directly to the PC, bypassing the ADAP early vision processing.

5.1.1 Calibrated Image Generation

During the development of the algorithms the PhotoBit imagers were not yet commercially available, so an alternative source of images was sought. While single scanned images could be used to test the edge detection algorithms, the lack of precise object positioning information limits their usefulness in testing sub-pixel edge resolution. To test the feature correlation and distance map computation routines well, sets of correlated tri-nocular stereo image triples are needed to test robustness against variances in imager positioning, noise, and small differences in lens geometries.

To alleviate the arduous task of creating so many test images in the traditional manner, simple rendering routines were implemented. The rendering routines take polygonal models

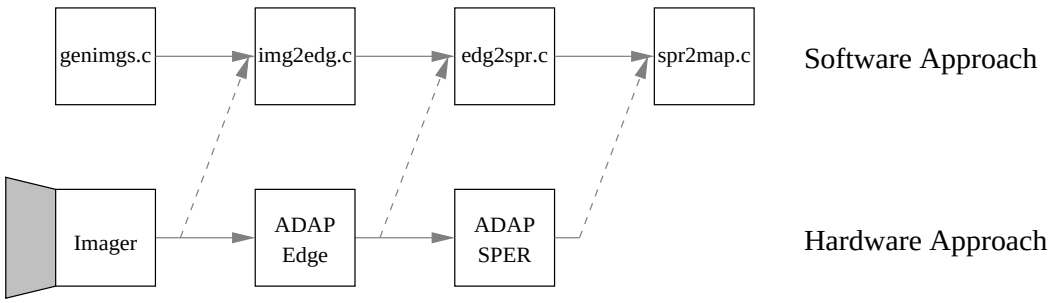

Figure 5-1: Software - hardware equivalence

and places them in arbitrary positions and orientations in space. Using a simple shading model that takes into account ambient illumination and reflected global illumination, the polygons are back-projected using simple perspective equations to generate simulated images. **genimgs.c** takes as its input a control file which denotes which model files should be loaded and positions the models in space. Given the coordinates of the lens center, the center of interest, and various other imager parameters, the scene may be rendered and stored. By moving the camera along a baseline, tri-nocular stereo triples are generated, as shown in Figure 5-2. By modifying the imaging parameters, the algorithms may be tested against many possible non-ideal imaging effects.

Figure 5-2: A rendered image triple

To test for this vision application, rough road scenes were generated. In order to test the sub-pixel algorithm, the original scenes were rendered at several times the required image resolution and dithered to emulate anti-aliasing. In addition to simulating non-ideal camera geometries, separate programs were written to blur the images and add speckle and Gaussian noise, to better simulate real images.

5.1.2 Edge Detection

While it is simple to code a given edge detection algorithm, special care must be taken to ensure that the code reliably simulates the operation of the algorithm when implemented in an ADAP array. While the analog inputs to the array may attain any analog value, the outputs of the ADAP arithmetic units are 8-bit quantized voltages between the arithmetic zero reference and the V_G and the maximum value V_{ref2}. To simulate this, internal voltages are represented as unsigned characters. When addition and subtraction operations are executed in the front-end sample and hold, the output may exceed the input range of the

ADC, but the ADC effectively thresholds over-range values. In multiplication and division operations, the reference voltage on the DAC is changed, but it never exceeds V_{ref2}, so the maximum output is similarly constrained.

The outputs of the positive, negative, and composite edge equations are shown in Figure 5-3. While the positive and negative edge detection operators alone are inherently smaller and execute with smaller latency, they produce and incomplete picture of the scene. The bi-directional edge operator combines the positive and negative edges into a single image. This operator loses some of its dynamic range due to the addition, but it produces more intelligible edge maps.

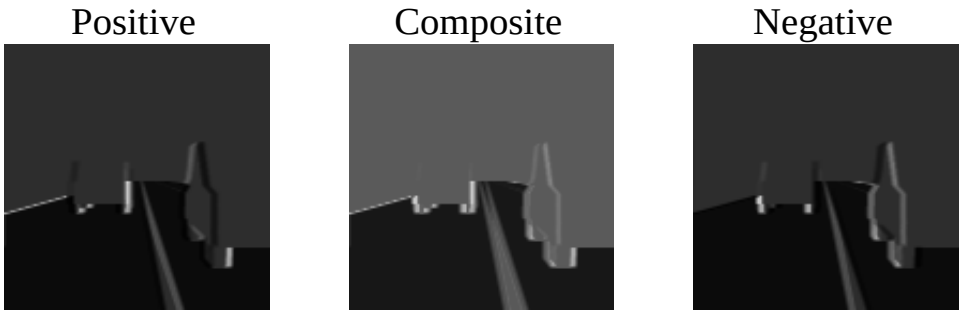

Figure 5-3: Edge detection operator comparison

5.1.3 Sub-Pixel Edge Resolution

The implementation of the sub-pixel algorithm in software faces the same constraints as the edge detectors. The valid outputs from the sub-pixel operation range from 1 to $2 * K$, where K is the ADAP division scaling parameter, as explained in the algorithms chapter. When values outside this range are thresholded in software on the PC, a crisp edge map such as Figure 5-4 is generated. Normally, the values of the valid pixels range from 1 to 18, but the contrast is increased in Figure 5-4 to make the varying edge positions better apparent.

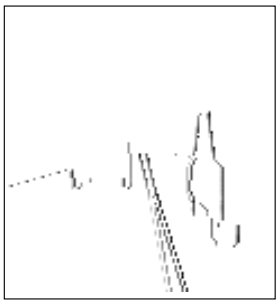

Figure 5-4: Sub-pixel resolved edge map

5.1.4 Distance Map Computation

The feature correlation and distance map computation algorithms are combined in **spr2map.c**. The program reads in three gray-scale images, encoded into the red, green, and blue por-

tions of a portable pixmap (ppm) file. Each feature in each row of the right is tested for correspondence with each feature in the corresponding row of the left image, by checking the midpoint in the center image for a feature with a given threshold. If the correlation is valid the distance is calculated from the disparity, coded as a color level and stored in the distance map. A sample distance map, corresponding to the edge maps in Figure 5-3 is shown in Figure 5-5. In the figure, more distant points are drawn in lighter shades of grey.

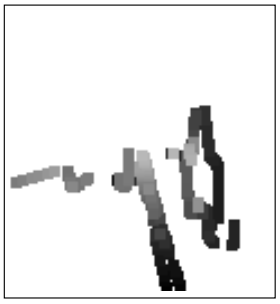

Figure 5-5: Calculated depth map

5.2 Data Acquisition Driver

There are two general paradigms for communication with an external peripheral on a PC: polled I/O and interrupt driven I/O. In polled I/O, the system continuously reads a port and waits for the external hardware to signal that the data is valid. With interrupt driven I/O, the external device triggers a hardware interrupt which forces the operating system to execute special interrupt handling code to read the data. In applications where the data rate is relatively slow, and no processing of the data occurs during acquisition, polled I/O is acceptable. The specifications for this system require 15 $\frac{frames}{sec}$ throughput using 64 x 64 pixel images, yielding an average sampling rate of 61.44 $\frac{ksamples}{sec}$. At this speed, interrupt driven I/O is the only viable option.

While polling can be achieved in user code, interrupt driven routines must be incorporated into the Unix kernel. The kernel driver written for this thesis provides complete access to the three 8-bit I/O ports on the data acquisition card [1]. User-level programs interface to the driver through **IOCTL** operations on the device file. This interface allows user code to configure, read from and write to the I/O ports without root privileges.

The application specific functionality is implemented in the interrupt handler. For generic applications, the internal data structures associated with the driver define input and output buffers that act as the sink and source for data read and written in the interrupt routines. The driver may be configured to read from or write to any of the I/O ports during each interrupt event. More complex **IOCTL** calls allow the user to set the size of both buffers, defined the contents of the output buffer, and read the contents of the input buffer. To meet the interfacing requirements of the external hardware system more complex functionality was implemented to support the configuration of the imagers, the programming of the ADAP arrays, and the frame-level acquisition of data.

[1] A more complete description of the data acquisition hardware is given in the appendix.

5.3 Imager Configuration

The hardware interface used to configure PhotoBit imager was described in the previous chapter. **progcam** is the program that interfaces with the kernel driver and gives it the data to transmitted to the imagers. **progcam** starts by reading the list of bytes to be transmitted from a file into an array and stores it in the driver's output buffer. After checking that the controller board is in program mode, the software configures the interrupt handler to configure the camera and enables interrupts. Once all the driver disables interrupts and signals **progcam** to exit.

The bulk of the programming routine is implemented within the interrupt handler. Φ_2 from the controller board is used to generate interrupts, and S_{CLK} is read through one of the parallel port status bits. SD_{IN} and SD_{EN}, the serial data input and serial data clock enable signals, are output through the parallel port data bits. When configuring the imager, the interrupt handler follows the rough state machine shown in Figure 5-6.

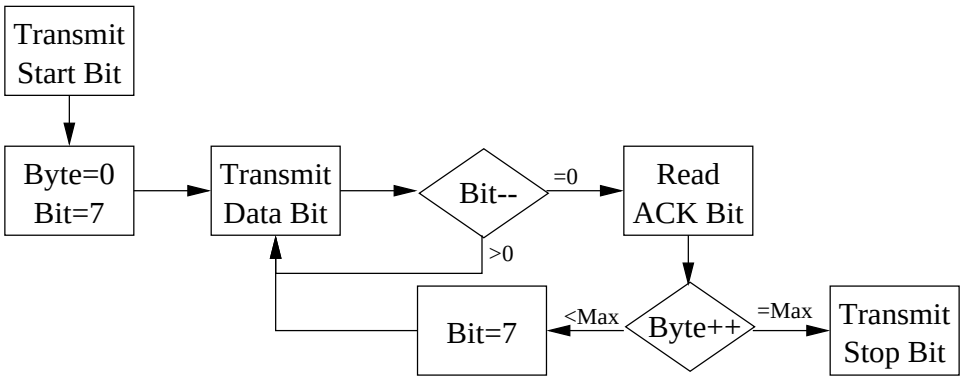

Figure 5-6: Imager configuration flowchart

Before the data is sent the SD_{EN} and SD_{IN} are asserted for 32 interrupts, enabling the serial clock so that the imagers may sync to it and prepare for the start bit. To send the start bit, the interrupt handler waits for an interrupt when S_{Clk} is asserted and sets SD_{IN} to 0. For each byte in the buffer, the bits are transmitted MSB first, with SD_{IN} changing only on interrupts when S_{Clk} is negated. After each byte, the code waits 1 S_{CLK} period for the imager to acknowledge the byte. After the final byte is sent, a stop bit is transmitted by asserting SD_{IN} while S_{Clk} is negated, followed by a 32 interrupt wait period, after which S_{Clk} is disabled.

5.4 ADAP Programming

The hardware interface that supports ADAP programming was also described in the previous chapter. **progadap** is the program that interfaces with the kernel driver and configures it to properly program the ADAP arrays. **progadap** allows the user to specify program files with which to program either or both of the ADAP arrays on the early vision board. The program reads in the program files, fills the data acquisition driver's output buffers, changes two of the data ports from input to output mode, configures the interrupt handler to program the ADAPs, and finally enables interrupts to initiate the programming procedure.

Φ_2 generates the interrupts for ADAP programming. For each interrupt a new byte is written to two of the ports on the data acquisition card, representing the five rows of program data, and the program signal itself. A total of 175 bytes are needed to program each ADAP, so 350 bytes are stored in the output buffer. After the final bytes are sent, the code assert the program signals for an additional cycle to allow the data to be latched into the ADAPs. The interrupt routine follows the state machine shown in Figure 5-7.

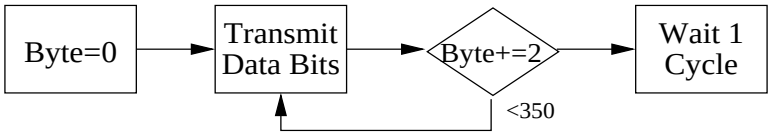

Figure 5-7: ADAP programming flowchart

5.5 Algorithm Execution

Once the imagers are configured and the ADAP arrays are programmed, the system is ready to acquire data and compute depth maps. **xdepth** combines the image processing routines described above with a front end to read the data, and a back end graphical output routine. For testing, two versions of xdepth were implemented. The first version consists of three autonomous processes, one that reads in the sub-pixel resolved edge data from the external hardware, one that performs the correlation and depth map calculations, and one that outputs a color coded depth map to an X-Windows display. The second version incorporates the edge detection and sub-pixel edge resolution code as additional processes and expects raw pixel data from the external hardware.

The interrupt routine for acquiring pixel or edge data is triggered by the falling edge of V_L. To acquire frame data, the three ports of the data acquisition card need to be con figured as inputs. The acquisition procedure is diagrammed in Figure 5-8. Initially, $NewFrame$ is asserted to reset the address counter on the PLD board. Next, a specified number of interrupts are ignored to account for the calculation latency of the system. Once the data coming from the ADAPs corresponds to valid pixel data, the interrupt routine reads the frame data and stores it in the input buffer. When all 4096 valid samples have been acquired, interrupts are disabled, and the user code is signaled that it may read the data from the buffer.

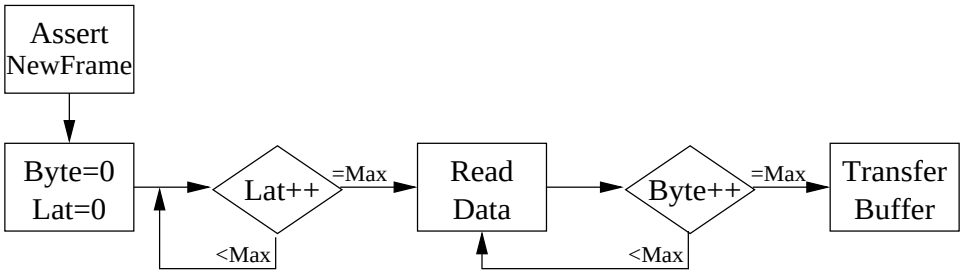

Figure 5-8: Frame acquisition flowchart

Chapter 6

Results and Future Work

The algorithms and hardware apparatus were tested throughout the development of the system. This chapter presents and analyzes the results from the software simulations and hardware tests.

6.1 Algorithm Simulation

The vision algorithms implemented in this thesis were thoroughly simulated using the custom **img2edg**, **edg2spr**, and **spr2map** software and images from the **genimgs** rendering software and scanned photographs.

Initial tests concerned the robustness of the algorithms to variations in camera geometries. In **spr2map**, the *thres* parameter, which determines how far a feature in the center image may be from the midpoint of the disparities in the right and left images and still be deemed a valid correspondence, is paramount in determining the sensitivity to geometric variances. If the *thres* parameter is chosen too low, the small camera movements render the majority of the true correspondences invalid. If *thres* is chosen too high, problems with spatial aliasing between distant objects arise. A moderate *thres* value of 0.5 pixels, worked well in simulation. For objects at reasonable distance, accurate distance maps were discernible with rotations up to 2^o about any axis, and with translations of up to 5% of the baseline distance. The algorithms seemed most sensitive to rotations along the baseline axis, as this produced the greatest change in the images.

The algorithms were tested with blurred and noisy images to further evaluate the robustness. Blur was simulated by convolving the input images with a small square mask. The edge detection and sub-pixel resolutions algorithms had no difficulty dealing with images blurred with a 5x5 mask. Adding Gaussian and speckle noise to the images did not significantly corrupt the results, due to the robustness of the tri-nocular stereo algorithm. Gaussian noise did not significantly affect the edge detection, and thus could not corrupt the output. While speckle noise is capable of producing false edges, the false edges produce false depth points only if they can be correlated with features in the other images.

Using both real and rendered images, the accuracy of the algorithms was evaluated. For most automotive vision applications, relative accuracy is the most important parameter, but given proper calibration, absolute accuracy was also achieved. With both scanned and rendered images, distances were calculated with errors less than 1-2% of the true distance. More importantly, the computed distances were linear and monotone with true distance.

6.2 Hardware Evaluation

While the timing on the hardware components could be checked using bench equipment, the overall functionality of the system could not be evaluated until the system, the driver software, and the user-level programs were complete. The functionality of the PLD board was necessary for the testing of the other boards. Once the PLD was properly programmed the output timing and programming signals were verified using a logic analyzer.

6.3 Imager PCB

The imager PCB worked properly on first power-up. Using a logic analyzer, the outputs of the $FValid$, $LValid$, and $PixClk$ were recorded to ensure proper timing. While it was possible to read the output using the version of the PLD board designed to accommodate software early vision, the data was meaningless when the camera was not reconfigured, as the addressing scheme continuously overwrite the data in the SRAM.

progcam was initially evaluated by observing the parallel port signals, but the configuration was unsuccessful although the outputs seemed correct. In the initial design of the PLD board, the S_{Clk} lines for the three imagers were driven by a single pin on the PLD, but the loading proved to be excessive so the signal was buffered before being sent. Similarly, the S_{Data} lines could not be driven by the parallel port pin directly, so they were latched on the PLD board. Once these changers were made, the imagers could be properly programmed.

After configuring the imagers to output centered 64 x 64 frames, and verifying the configuration with the logic analyzer, the software early vision hardware could be fully tested. The hardware was able to output images at approximately $24\frac{frames}{sec}$, exceeding the $20\frac{frames}{sec}$ target. The frame rate was limited only by latency of the data acquisition interface, not by the hardware or software. Tests indicated that the interrupt latency of the PC and DAQ card would not guarantee correct sampling only for signals generating interrupts faster than approximately 100 kHz. This limited the maximum attainable frame rate to $24.414\frac{frames}{sec}$.

6.3.1 ADAP Early Vision Board

Unfortunately, similar success were not achieved with the ADAP boards. The timing signals for the ADAP board were verified using a logic analyzer. The biases for the arrays were all generated on the early vision board, so the relevant voltages were easily observed and adjusted. Using simple programs that configured the switch matrices within the ADAP cells to connect various chip inputs, the proper functionality of the **progadap** software was established.

When the arithmetic units not the ADAP were tested, problems arose. Connecting one ALU input V_G, and the other to a constant input, performing an addition operation, and reading the digital output bits effectively turns the ADAP cell into a simple ADC. In this configuration, the ADC reached slightly above half-scale when a full scale input was applied. When the input signals were routed through more than a single switch, further degradation of the ADC swing was observed. Further tests showed a DC current between the two inputs to the arithmetic unit. Since the ADAP is implemented using switched-capacitor circuitry, only a minute DC input current is expected. The observed current between the nodes was roughly proportional to the voltage difference between the nodes, and was orders of

magnitude higher than the expected switched-capacitor charge, implying a parasitic short between the ALU inputs. The excessive input currents explained the degradation in the swing of the ADC, and voltage was being dropped across the switches in the switch matrix.

The failures were fairly consistent across the set of ADAP chips tested. Given the age of the chips [1], degradation due to poor passivation or static discharge in some of the chips was not unexpected. Due the problems with the ADAP, only the software early vision implementation of the system could be evaluated.

6.3.2 System Evaluation

Since the hardware early vision system was inoperable, the software version of the test system was used in the final evaluation. With the early vision algorithms implemented as part of **xdepth**, the performance of the system was equal the performance of the software acting on scanned photographs as described above. On the Pentium 233 laptop PC, The early vision algorithms consumed very little processor time, and the three processing routines were able to operate in the background without decreasing the data sampling rate. The $20\frac{frames}{sec}$ goal was easily exceeded by the completed system.

6.4 Future Work

The natural extension of this work is consolidate the design into a much smaller chipset. The imager and the processor should be implemented on a single chip with all the biasing done internally. This would allow the output ADC to be removed from the imager, and the input DAC to be elminated from the ADAP. By increasing the size of the array, the two algorithms could be implemented together, eliminating the intermediate DAC. Additionally, the clock generators could be implemented on each array to save the distribution of multiple signals between boards. Since CMOS technology and scaling have changed in the years since the ADAP was designed, the effeciency of using analog rather than very low power digital circuitry would need to be reevaluated. The edge correlation and distance map computation could be implemented on an ASIC, but a separate display subsystem would be required if the PC is eliminated from the system.

[1] The ADAP chips were fabricated in 1995 and originally tested in 1996, over three years before they were used in this work.

Appendix A

Apparatus Setup Instructions

The hardware used in the test system is comprised of several volatile components that must be reprogrammed after each power cycle to ensure proper operation. All of the components are in-system programmable, so the system need not be disassembled. The Altera PLD is an EEPROM-based part, so it does not lose its program, but since the other components need the PLD to be programmed correctly before they can be programmed, it is safest to reprogram it before testing. The ADAP processors and cameras both store their programs in registers, so it is essential that they be reprogrammed at power-up.

A.1 System Setup

Before the system is used, all of the boards must be properly connected. The setup is different depending on whether the ADAP boards are used for the early vision processing, so the two cases are treated separately.

A.1.1 Software Early Vision

- Connect the output of the DC transformer to the two pin port on the PLD board.
- Plug one of the two connector ribbon cables into each of the camera PCBs.
- Plug the other end of the camera cables into the PLD controller board. The cable from the left camera should be plugged into the headers closest to the edge, followed by the center and right cameras.
- Plug the large AMP cable into the DB37 connector on the PLD board and into the PCMCIA data acquisition card in the laptop.
- Plug the DB25-IDC26 cable into the IDC26 header at the top of the PLD board.
- Plug the IDC10 connector from the ByteBlaster programming adapter into the IDC10 header at the top left of the PLD board.

A.1.2 ADAP Early Vision

- Set the jumpers on the three ADAP boards. For the algorithms as defined, the header directly to the left of each ADAP should be fully jumpered, the header to the right of

the lower ADAP should be jumpered to 2, and the header to the right of the upper ADAP should be jumpered to 0.

- Screw the three ADAP boards and the PLD together.
- Connect the output of the DC transformer to either of the two pin ports on the bottom ADAP board, and daisy chain the rest of the boards together using the two pin power plugs.
- Using the short IDC26-IDC26 cables, connect each ADAP board to the PLD board. The top board should connect to the header closest to the edge of the PLD board.
- Plug one of the two connector ribbon cables into each of the camera PCBs.
- Plug the other ends of the camera cables into the 10 pin header on the ADAP and the 20 pin header on the PLD board. The left camera should connect to the top ADAP board and the connector nearest the edge of the PLD board.
- Plug the large AMP cable into the DB37 connector on the PLD board and into the PCMCIA data acquisition card in the laptop.
- Plug the DB25-IDC26 cable into the IDC26 header at the top of the PLD board.
- Plug the IDC10 connector from the ByteBlaster programming adapter into the IDC10 header at the top left of the PLD board.

A.2 PLD Programming

The Altera EPM7128S part may be programmed through a standard 10-pin JTAG interface, which is incorporated into the design of the PLD controller board. The JTAG interface is interfaced to either the serial or parallel port on a PC with adapters supplied by Altera [12, 13]. The device actual programming routines are available as part of the Altera Max+plus II software.

The two versions of the controller board for the different test setup configurations require different PLD programs:

- Ensure that the PCMCIA ethernet card is inserted and properly connected to an ethernet port.
- Boot the laptop into Windows 95 by rebooting and entering **win95** at the LILO prompt.
- Log into the Windows NT domain MTL as **mspaeth**.
- Launch the Altera MAX+plus II software by double clicking the icon on the desktop.
- Plug the Altera ByteBlaster programming adapter into the printer port on the laptop.
- Power on the PLD board.

- From the menu, choose File → Open to load the appropriate project file. All of the libraries associated with this thesis are located on the NT system **eurpoa**, and may be accessed via **\\eurpoa\mspaeth\altera\adap\lib** or through the SMB mounted **Y:** drive. For the setup using PC software for the early vision routines, the project file is **adapctl3.gdf**. For the setup using the ADAPs, the project file is **adapctl2.gdf**.
- From the menu choose File → Project → Set Project to Current File.
- From the menu choose File → Project → Save and Compile to make sure everything is up to date.
- From the menu choose Max+plus II → Programmer.
- Click on OK to program the device.

The device should program and verify properly. If errors occur, make sure the cables are firmly connected and that the PLD board is properly powered. An alternative programming interface using the JAM in-system programming protocol was also tested and ported to Linux, but it proved to be cumbersome and unreliable under any operating system.

A.3 Camera Programming

The camera PCBs were designed to buffer 4096 pixel values addressed in a 64 pixel by 64 pixel array. For proper system operation, each camera must be programmed to output a 64 x 64 window centered within the image frame. The details of the programmable registers on the PB159 are given in [5], and the details of the programming code are discussed in a later appendix.

As the programming and test routines are run from Linux, the following procedure must be followed to prepare for any of the later routines:

- Ensure that that PCMCIA data acquisition card is inserted.
- Boot the computer into Linux by rebooting and entering **work** at the LILO prompt.
- Log into XWindows as **mspaeth**.
- Open an xterm window by clicking on the xterm icon in the button bar.
- Plug the DB25-IDC26 cable into the parallel port on the laptop.
- Type **/sbin/lsmod** and ensure that the pcmd24_cs driver is loaded.
- Type **source ~/.login** to make sure the path is correct.

To program the camera:

- Turn on the PLD board. The PLD should have been programmed earlier.
- Put the PLD board into program mode using the switch at the lower left. In program mode the red **Prog** LED should be lit.
- Type **cd ~/classwork/research/pb159**.

- Type **progcam pb159.prog**.
- Switch the PLD board back into run mode.

The camera boards are powered by the PLD board, so if the PLD board is turned off the camera chips will need to be reprogrammed.

A.4 ADAP Programming

The ADAP arrays come up in a random state at power on, so they must be programmed before they will perform any useful operations. An ADAP program begins as text file defining the array size, and listing the switches that are turned on in each cell. The **.prg** program files may be converted to **xfig** figures to graphically view the switch interconnections using the program **adap2fig**. The **.prg** files are converted to **.bin** binary files for use by the programming algorithm by **adap2prg**. These two programs can operate on the file formats developed for this project, or the file formats used by Dr. David Martin for his thesis [6].

To program the ADAP arrays:

- Check that the PLD controller board is turned on.
- Turn on the three ADAP boards.
- Switch the PLD board into program mode.
- Type **cd ~/classwork/research/adap**.
- Type **progadap -a edg_11.bin -b spr_11.bin**.
- Switch the PLD board back into run mode.

A.5 Running XDepth

Once the cameras have been configured and the ADAPs have been programmed (if necessary), the system is ready to run **xdepth**. If the system is being tested with the ADAP boards to perform the early vision, **xdepth_adap** should be executed, otherwise **xdepth_soft** should be run. In either case, the program will spawn an x-window to display the results of the algorithm. In the display, the distances are indicated by colors with red being near, fading to green and blue for far objects. The black background indicates that no object was detected for that point.

The xdepth software is not a fully implemented X application, so if the window will not auto-refresh if it is covered by another. If this occurs, simply terminate and restart the program.

Appendix B

ADAP Simulation Routines

The following C code simulates the operation of the ADAP processors executing the programs necessary to perform the edge detection and sub-pixel edge resolution routines. Initially, these programs were used to test the robustness of the trinocular stereo routines against inaccuracies in camera placement and alignment, additive and salt-and-pepper noise in the images, and loss of image focus. The input images for these tests were generated digitally using road scenes modeled with polygons, but the source for this software is beyond the scope of this thesis.

The image manipulation routines were designed to operate on a stream of images in PGM (portable grey map) format. To reduce the total number of files processed, the three grey images comprising a trinocular stereo image triple were encoded in the red, green, and blue components of a PNM (portable any map) file, and each color component was processed separately.

In order to properly simulate the unique characteristics of ADAP arithmetic special care must be taken. While the output of each ALU is in analog format, it must be treated as a digital value since it is generated by a DAC. Since addition and subtraction operations are done in the analog domain before the A-D conversion, neither values below 0_{dig} nor above 255_{dig} can be output, since they would merely saturate the ADC. Multiplications consist of scaling one input by fraction of 256 represented by the other input, so the product is always less than either input. For division, the result is scaled by a discrete integer to regain some of the dynamic range lost by the ADAPs inability to represent fractional quantities.

B.1 Edge Detection : img2edg.c

This program simulates the following equations, for positive edge maps, negative edge maps, and composite edge maps respectively:

$$E_x = 9 \cdot \frac{.60 \cdot p_{x+2} + .60 \cdot p_{x+1}}{.12 \cdot p_{x-1} + .12 \cdot p_{x-2}} \tag{B.1}$$

$$E_x = 9 \cdot \frac{.60 \cdot p_{x-2} + .60 \cdot p_{x-1}}{.12 \cdot p_{x+1} + .12 \cdot p_{x+2}} \tag{B.2}$$

$$E_x = 9 \cdot \left[\frac{.60 \cdot p_{x+2} + .60 \cdot p_{x+1}}{.12 \cdot p_{x-1} + .12 \cdot p_{x-2}} + \frac{.60 \cdot p_{x-2} + .60 \cdot p_{x-1}}{.12 \cdot p_{x+1} + .12 \cdot p_{x+2}} \right] \tag{B.3}$$

```
// img2edg.c
// Detects positive, negative, or composite edge maps
// Mark Spaeth, October 1997

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#define RIGHT 1
#define LEFT  2

static char *usage="\
Usage : %s [-n<# frames>] [-i<inputbase>] [-o<outputbase>] [-r / -l / -b]\n\n\
Finds edge maps in a pnm style bitmap files\n\
Default behavior is a filter from stdin to stdout\n\n";

void main(int argc, char *argv[]) {
  char *arg,*inpbase,*outbase;
  char *inpname,*outname;
  char c, edge=3;
  FILE *inpfile, *outfile;
  int inpbaselen,outbaselen;
  int nframes,xsize,ysize;
  int i,frame,x,y,pval;
  int mode,err,pscale;
  long pcnt;
  int pvals[5][3];
  double pnumerl,pdenoml,pnumerr,pdenomr,pout;

  nframes=1000000;
  frame=0;
  inpfile=stdin;
  outfile=stdout;
  inpbase=NULL;
  outbase=NULL;

  for (i=1;i<argc;i++) {
    arg=argv[i];
    if (*arg!='-') {
      fprintf(stderr,"Invalid argument: %s\n\n",arg);
      fprintf(stderr,usage,argv[0]);
      exit(1);
    }
    switch (*(++arg)) {
    case 'b':
      edge=LEFT|RIGHT;             // Composite edge maps
      break;
    case 'h':
      fprintf(stderr,usage,argv[0]);
      exit(1);
      break;
    case 'i':
      if (strlen(++arg)>0) {
        inpbase=(arg);
      } else {
        fprintf(stderr,"Error: null input filename\n\n");
        exit(1);
      }
      break;
    case 'l':
      edge=LEFT;                   // Positive edge maps
      break;
    case 'n':
      nframes=atoi(++arg);
      break;
    case 'o':
      if (strlen(++arg)>0) {
```

```
      outbase=(arg);
    } else {
      fprintf(stderr,"Error: null output filename\n\n");
      exit(1);
    }
    break;
  case 'r':
    edge=RIGHT;                  // Negative edge maps
    break;
  default:
    fprintf(stderr,"Invalid argument: -%s\n\n",arg);
    fprintf(stderr,usage,argv[0]);
    exit(1);
    break;
  }
}
if (inpbase!=NULL) {
  inpbaselen=strlen(inpbase);
  inpname=(char *)malloc((strlen(inpbase)+11)*sizeof(char));
  strcpy(inpname,inpbase);
  strcat(inpname,"000000.img");
} else {
  inpfile=stdin;
}
if (outbase!=NULL) {
  outbaselen=strlen(outbase);
  outname=(char *)malloc((strlen(outbase)+11)*sizeof(char));
  strcpy(outname,outbase);
  strcat(outname,"000000.edg");
} else {
  outfile=stdout;
}
frame=0;
while(1) {
  if (inpbase!=NULL) {
    for(i=6;i>0;i--)
      inpname[inpbaselen+6-i]='0'+(char)((frame % (int)pow(10,i))/(int)pow(10,i-1));
    if ((inpfile=fopen(inpname,"r"))==NULL) {
      fprintf(stderr,"Input file not found: %s\n",inpname);
      break;
    } else {
      fprintf(stderr,"Converting file: %s\n",inpname);
    }
  }
  if(fscanf(inpfile,"P%i\n",&mode)!=1) {
    if ((inpbase!=NULL)||(mode<1)||(mode>6))
      fprintf(stderr,"Invalid ppm header\n");
    break;
  }
  while (((c=fgetc(inpfile))=='#')&&(c!=EOF))
    while (((c=fgetc(inpfile))!='\n')&&(c!=EOF));
  if (c==EOF) {
    fprintf(stderr,"Unexpected EOF\n");
    break;
  } else {
     ungetc(c,inpfile);
  }
  if (fscanf(inpfile,"%i %i\n",&xsize,&ysize)!=2) {
    fprintf(stderr,"Error reading image size\n");
    break;
  }
  if (xsize<5) {
    fprintf(stderr,"Image width too small\n");
    break;
  }
  if (mode%3!=1) {
    if (fscanf(inpfile,"%i\n",&pscale)!=1) {
      fprintf(stderr,"Error reading pixel scale\n");
```

```
      break;
    }
  } else {
    pscale=1;
  }
  if (outbase!=NULL) {
    for(i=6;i>0;i--)
      outname[outbaselen+6-i]='0'+(char)((frame % (int)pow(10,i))/(int)pow(10,i-1));
    outfile=fopen(outname,"w");
  }
  fprintf(outfile,"P%i\n",5+(mode%3==0));
  fprintf(outfile,"%i %i\n",xsize-4,ysize);
  fprintf(outfile,"255\n");
  pcnt=0;
  err=0;
  for (y=0;y<ysize&&!err;y++) {
    for (x=0;x<xsize&&!err;x++) {
      for (i=0;i<1+2*((mode%3==0))&&!err;i++) {
        if (mode<4) {
               if ((err=(fscanf(inpfile,"%d ",&pval)!=1))) continue;
        } else {
          if ((err=((pval=fgetc(inpfile))==EOF))) continue;
        }
        pvals[pcnt%5][i]=(unsigned char)((pval*255)/pscale);
        if (x>=4) {
          switch (edge) {
          case LEFT:
            pnumerl=.6*(pvals[(pcnt+5)%5][i]+pvals[(pcnt+4)%5][i]);
            pnumerl=(pnumerl>255)?255:pnumerl;
            pdenoml=(2304/(.12*(pvals[(pcnt+2)%5][i]+pvals[(pcnt+1)%5][i])));
            pdenoml=(pdenoml>255)?255:pdenoml;
            pout=(pnumerl*pdenoml)/256;
            break;
          case RIGHT:
            pnumerr=.6*(pvals[(pcnt+1)%5][i]+pvals[(pcnt+2)%5][i]);
            pnumerr=(pnumerr>255)?255:pnumerr;
            pdenomr=(2304/(.12*(pvals[(pcnt+4)%5][i]+pvals[(pcnt+5)%5][i])));
            pdenomr=(pdenomr>255)?255:pdenomr;
            pout=(pnumerr*pdenomr)/256;
            break;
          default:
            pnumerl=.6*(pvals[(pcnt+5)%5][i]+pvals[(pcnt+4)%5][i]);
            pnumerl=(pnumerl>255)?255:pnumerl;
            pdenoml=(2304/(.12*(pvals[(pcnt+2)%5][i]+pvals[(pcnt+1)%5][i])));
            pdenoml=(pdenoml>255)?255:pdenoml;
            pnumerr=.6*(pvals[(pcnt+1)%5][i]+pvals[(pcnt+2)%5][i]);
            pnumerr=(pnumerr>255)?255:pnumerr;
            pdenomr=(2304/(.12*(pvals[(pcnt+4)%5][i]+pvals[(pcnt+5)%5][i])));
            pdenomr=(pdenomr>255)?255:pdenomr;
            pout=(pnumerl*pdenoml+pnumerr*pdenomr)/256;
            break;
          }
          pout=(pout>255)?255:pout;
          fprintf(outfile,"%c",(unsigned char)(pout));
        }
      }
      pcnt++;
    }
  }
  if (err) {
    fprintf(stderr,"Unexpected EOF reading data\n");
    break;
  }
  if (inpbase!=NULL) fclose(inpfile);
  if (outbase!=NULL) fclose(outfile);
  if ((++frame)>=nframes) break;
}
fprintf(stderr,"%i frames processed.\n\n",frame);
```

```
  if (inpbase!=NULL) free(inpname);
  if (outbase!=NULL) free(outname);
}
```

B.2 Sub-pixel Edge Resolution : edg2spr.c

This code simulates the following equation:

$$\alpha = \frac{-.75 \cdot e_{x-1} + e_x - .25 \cdot e_{x+1}}{-.50 \cdot e_{x-1} + e_x - .50 \cdot e_{x+1}} \tag{B.4}$$

```
// edg2spr.c
// Resolves edges to sub-pixel resolution
// Mark Spaeth, October 1997

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

static char *usage="\
Usage : %s [-n<# frames>] [-i<inputbase>] [-o<outputbase>]\n\n\
Finds edge edges to subpixel resolution  in a pnm style edge files\n\
Default behavior is a filter from stdin to stdout\n\n";

void main(int argc, char *argv[]) {
  char *arg,*inpbase,*outbase;
  char *inpname,*outname;
  char c;
  FILE *inpfile, *outfile;
  int inpbaselen,outbaselen;
  int nframes,xsize,ysize;
  int i,frame,x,y,pval;
  int mode,err,pscale;
  long pcnt;
  int pvals[3][3];
  double pnumer,pdenom,pout;

  nframes=1000000;
  frame=0;
  inpfile=stdin;
  outfile=stdout;
  inpbase=NULL;
  outbase=NULL;

  for (i=1;i<argc;i++) {
    arg=argv[i];
    if (*arg!='-') {
      fprintf(stderr,"Invalid argument: %s\n\n",arg);
      fprintf(stderr,usage,argv[0]);
      exit(1);
    }
    switch (*(++arg)) {
    case 'h':
      fprintf(stderr,usage,argv[0]);
      exit(1);
      break;
    case 'i':
      if (strlen(++arg)>0) {
        inpbase=(arg);
      } else {
        fprintf(stderr,"Error: null input filename\n\n");
        exit(1);
      }
```

```
    break;
  case 'n':
    nframes=atoi(++arg);
    break;
  case 'o':
    if (strlen(++arg)>0) {
      outbase=(arg);
    } else {
      fprintf(stderr,"Error: null output filename\n\n");
      exit(1);
    }
    break;
  default:
    fprintf(stderr,"Invalid argument: -%s\n\n",arg);
    fprintf(stderr,usage,argv[0]);
    exit(1);
    break;
  }
}
if (inpbase!=NULL) {
  inpbaselen=strlen(inpbase);
  inpname=(char *)malloc((strlen(inpbase)+11)*sizeof(char));
  strcpy(inpname,inpbase);
  strcat(inpname,"000000.edg");
} else {
  inpfile=stdin;
}
if (outbase!=NULL) {
  outbaselen=strlen(outbase);
  outname=(char *)malloc((strlen(outbase)+11)*sizeof(char));
  strcpy(outname,outbase);
  strcat(outname,"000000.spr");
} else {
  outfile=stdout;
}
frame=0;
while(1) {
  if (inpbase!=NULL) {
    for(i=6;i>0;i--)
      inpname[inpbaselen+6-i]='0'+(char)((frame % (int)pow(10,i))/(int)pow(10,i-1));
    if ((inpfile=fopen(inpname,"r"))==NULL) {
      fprintf(stderr,"Input file not found: %s\n",inpname);
      break;
    } else {
      fprintf(stderr,"Converting file: %s\n",inpname);
    }
  }
  if(fscanf(inpfile,"P%i\n",&mode)!=1) {
    if ((inpbase!=NULL)||(mode<1)||(mode>6))
      fprintf(stderr,"Invalid ppm header\n");
    break;
  }
  while (((c=fgetc(inpfile))=='#')&&(c!=EOF))
    while (((c=fgetc(inpfile))!='\n')&&(c!=EOF));
  if (c==EOF) {
    fprintf(stderr,"Unexpected EOF\n");
    break;
  } else {
     ungetc(c,inpfile);
  }
  if (fscanf(inpfile,"%i %i\n",&xsize,&ysize)!=2) {
    fprintf(stderr,"Error reading image size\n");
    break;
  }
  if (xsize<3) {
    fprintf(stderr,"Image width too small\n");
    break;
  }
```

```
    if (mode%3!=1) {
      if (fscanf(inpfile,"%i\n",&pscale)!=1) {
        fprintf(stderr,"Error reading pixel scale\n");
        break;
      }
    } else {
      pscale=1;
    }
    if (outbase!=NULL) {
      for(i=6;i>0;i--)
        outname[outbaselen+6-i]='0'+(char)((frame % (int)pow(10,i))/(int)pow(10,i-1));
      outfile=fopen(outname,"w");
    }
    fprintf(outfile,"P%i\n",5+(mode%3==0));
    fprintf(outfile,"%i %i\n",xsize-2,ysize);
    fprintf(outfile,"255\n");
    pcnt=0;
    err=0;
    for (y=0;y<ysize&&!err;y++) {
      for (x=0;x<xsize&&!err;x++) {
        for (i=0;i<1+2*((mode%3==0))&&!err;i++) {
          if (mode<4) {
                  if ((err=(fscanf(inpfile,"%d ",&pval)!=1))) continue;
          } else {
            if ((err=((pval=fgetc(inpfile))==EOF))) continue;
          }
          pvals[pcnt%3][i]=(unsigned char)((pval*255)/pscale);
          if (x>=2) {
            pnumer=pvals[(pcnt-1)%3][i]-(.75*pvals[(pcnt+1)%3][i]+.25*pvals[(pcnt)%3][i]);
            pdenom=pvals[(pcnt-1)%3][i]-(.5*pvals[(pcnt+1)%3][i]+.5*pvals[(pcnt)%3][i]);
            pout=(pdenom>4)&&(pnumer>=0) ? (127*pnumer/pdenom) : 255;
            pout=(pout>255)?255:pout;
            fprintf(outfile,"%c",(unsigned char)rint(pout));
          }
        }
        pcnt++;
      }
    }
    if (err) {
      fprintf(stderr,"Unexpected EOF reading data\n");
      break;
    }
    if (inpbase!=NULL) fclose(inpfile);
    if (outbase!=NULL) fclose(outfile);
    if ((++frame)>=nframes) break;
  }
  fprintf(stderr,"%i frames processed.\n\n",frame);
  if (inpbase!=NULL) free(inpname);
  if (outbase!=NULL) free(outname);
}
```

B.3 Depth Map Computation : spr2map.c

This code performs correlates sub-pixel resolved edge map files output from *edg2spr.c* and outputs a color coded depth map. Various camera geometry parameters may be input on the command line to control the depth range of the output depth map.

```
// spr2map.c
// Converts .spr files into depthmaps
// Mark Spaeth, October 1998

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
#include <string.h>

typedef struct nodetag node;
typedef struct nodetag {
  double xval;
  node *next;
} nodetag;

void freenodelist(node *);

static char *usage="\
Usage : %s [-n<# frames>] [-i<inputbase>] [-o<outputbase>]\n\
           [-b<baselinelength>] [-m<maxdist>] [-r<resolution]\n\
           [-c<convolution>] [-t<threshold>] [-f<focal length>]\n\n\
Finds distance maps to subpixel resolution in a pnm style spr files\n\
Default behavior is a filter from stdin to stdout\n\n";

void main(int argc, char *argv[]) {
  char *arg,*inpbase,*outbase;
  char *inpname,*outname;
  FILE *inpfile, *outfile;
  double coffset, maxdist, res, thres, focal;
  int conv;
  int *distmap, distmaplen;
  node **edglocs[3];
  node *ntmp[3];
  int inpbaselen,outbaselen;
  int nframes,xsize,ysize;
  int i,j,frame,x,y,pval;
  int mode,err,pscale;
  long pcnt;
  char c,match;
  double sloc;
  int loc;

  nframes=1000000;
  inpbase=NULL;
  outbase=NULL;
  distmap=NULL;
  for (i=0;i<3;i++)
    edglocs[i]=NULL;
  coffset=2;
  maxdist=50;
  res=.1;
  conv=11;
  thres=2.0;
  focal=351.677;
  frame=0;
  inpfile=stdin;
  outfile=stdout;

  for (i=1;i<argc;i++) {
    arg=argv[i];
    if (*arg!='-') {
      fprintf(stderr,"Invalid argument: %s\n\n",arg);
      fprintf(stderr,usage,argv[0]);
      exit(1);
    }
    switch (*(++arg)) {
    case 'b':
      if ((sscanf((arg+1),"%lf",&coffset))!=1) {
        fprintf(stderr,"Invalid argument: %s\n\n",arg);
        fprintf(stderr,usage,argv[0]);
        exit(1);
      }
      if (coffset<=0) {
        fprintf(stderr,"Camera baseline offset must be positive\n\n");
        exit(1);
```

```
  }
  break;
case 'c':
  if ((sscanf((arg+1),"%d",&conv))!=1) {
    fprintf(stderr,"Invalid argument: %s\n\n",arg);
    fprintf(stderr,usage,argv[0]);
    exit(1);
  }
  if (conv<=0) {
    fprintf(stderr,"Distance convolution must be positive\n\n");
    exit(1);
  }
  break;
case 'f':
  if ((sscanf((arg+1),"%lf",&focal))!=1) {
    fprintf(stderr,"Invalid argument: %s\n\n",arg);
    fprintf(stderr,usage,argv[0]);
    exit(1);
  }
  if (thres<=0) {
    fprintf(stderr,"Focal length must be positive\n\n");
    exit(1);
  }
  break;
case 'h':
  fprintf(stderr,usage,argv[0]);
  exit(1);
  break;
case 'i':
  if (strlen(++arg)>0) {
    inpbase=(arg);
  } else {
    fprintf(stderr,"Error: null input filename\n\n");
    exit(1);
  }
  break;
case 'm':
  if ((sscanf((arg+1),"%lf",&maxdist))!=1) {
    fprintf(stderr,"Invalid argument: %s\n\n",arg);
    fprintf(stderr,usage,argv[0]);
    exit(1);
  }
  if (maxdist<=0) {
    fprintf(stderr,"Maximum resolved distance must be positive\n\n");
    exit(1);
  }
  break;
case 'n':
  nframes=atoi(++arg);
  break;
case 'o':
  if (strlen(++arg)>0) {
    outbase=(arg);
  } else {
    fprintf(stderr,"Error: null output filename\n\n");
    exit(1);
  }
  break;
case 'r':
  if ((sscanf((arg+1),"%lf",&res))!=1) {
    fprintf(stderr,"Invalid argument: %s\n\n",arg);
    fprintf(stderr,usage,argv[0]);
    exit(1);
  }
  if (res<=0) {
    fprintf(stderr,"Distance resolution must be positive\n\n");
    exit(1);
  }
```

```
    break;
  case 't':
    if ((sscanf((arg+1),"%lf",&thres))!=1) {
      fprintf(stderr,"Invalid argument: %s\n\n",arg);
      fprintf(stderr,usage,argv[0]);
      exit(1);
    }
    if (thres<=0) {
      fprintf(stderr,"Match threshold must be positive\n\n");
      exit(1);
    }
    break;
  default:
    fprintf(stderr,"Invalid argument: -%s\n\n",arg);
    fprintf(stderr,usage,argv[0]);
    exit(1);
    break;
  }
}

distmaplen=ceil(maxdist/res);
distmap=(int *)malloc(distmaplen*sizeof(int));

if (inpbase!=NULL) {
  inpbaselen=strlen(inpbase);
  inpname=(char *)malloc((strlen(inpbase)+11)*sizeof(char));
  strcpy(inpname,inpbase);
  strcat(inpname,"000000.spr");
} else {
  inpfile=stdin;
}
if (outbase!=NULL) {
  outbaselen=strlen(outbase);
  outname=(char *)malloc((strlen(outbase)+11)*sizeof(char));
  strcpy(outname,outbase);
  strcat(outname,"000000.map");
} else {
  outfile=stdout;
}

frame=0;
while(1) {
  if (inpbase!=NULL) {
    for(i=6;i>0;i--)
      inpname[inpbaselen+6-i]='0'+(char)((frame % (int)pow(10,i))/(int)pow(10,i-1));
    if ((inpfile=fopen(inpname,"r"))==NULL) {
      fprintf(stderr,"Input file not found: %s\n",inpname);
      break;
    } else {
      fprintf(stderr,"Converting file: %s\n",inpname);
    }
  }
  if(fscanf(inpfile,"P%i\n",&mode)!=1) {
    if ((inpbase!=NULL)||((mode!=3)&&(mode!=6)))
      fprintf(stderr,"Invalid ppm header\n");
      err++;
    break;
  }
  while (((c=fgetc(inpfile))=='#')&&(c!=EOF))
    while (((c=fgetc(inpfile))!='\n')&&(c!=EOF));
  if (c==EOF) {
    fprintf(stderr,"Unexpected EOF\n");
      err++;
    break;
  } else {
     ungetc(c,inpfile);
  }
  if (fscanf(inpfile,"%i %i\n",&xsize,&ysize)!=2) {
```

```
  fprintf(stderr,"Error reading image size\n");
    err++;
  break;
}
if (xsize<3) {
  fprintf(stderr,"Image width too small\n");
    err++;
  break;
}
if (fscanf(inpfile,"%i\n",&pscale)!=1) {
  fprintf(stderr,"Error reading pixel scale\n");
  err++;
  break;
}

for (i=0;i<3;i++) {
  edglocs[i]=(node **)malloc(ysize*sizeof(node *));
  for (j=0;j<ysize;j++) {
    edglocs[i][j]=(node *)malloc(sizeof(node));
    edglocs[i][j]->next=NULL;
    edglocs[i][j]->xval=-2;
  }
}

pcnt=0;
err=0;
for (y=0;y<ysize&&!err;y++) {
  for(i=0;i<3;i++)
    ntmp[i]=edglocs[i][y];
  for (x=0;x<xsize&&!err;x++) {
    for (i=0;i<3&&!err;i++) {
      if (mode==3) {
        if ((err=(fscanf(inpfile,"%d ",&pval)!=1))) continue;
      } else {
        if ((err=((pval=fgetc(inpfile))==EOF))) continue;
      }
      if (pval!=255) {
        sloc=x-2.+((double)pval/127.);
        if (abs(sloc-ntmp[i]->xval)>1) {
          ntmp[i]->next=(node *)malloc(sizeof(node));
          ntmp[i]=ntmp[i]->next;
          ntmp[i]->next=NULL;
          ntmp[i]->xval=sloc;
        } else {
          ntmp[i]->xval=.5*(sloc+ntmp[i]->xval);
        }
      }
    }
    pcnt++;
  }
}

if (err) {
  fprintf(stderr,"Unexpected EOF reading data\n");
  break;
} else {
  for (i=0;i<distmaplen;i++)
    distmap[i]=0;

  for (y=0;y<ysize;y++) {
    ntmp[0]=edglocs[0][y];

    while (ntmp[0]->next!=NULL) {
      ntmp[2]=edglocs[2][y];

      while (ntmp[2]->next!=NULL) {
        if ((ntmp[2]->next->xval)<(ntmp[0]->next->xval)) {
          sloc=.5*(ntmp[2]->next->xval+ntmp[0]->next->xval);
```

```
              ntmp[1]=edglocs[1][y];
              match=0;
              while (ntmp[1]->next!=NULL) {
                match+=(abs(ntmp[1]->next->xval-sloc)<=thres);
                ntmp[1]=ntmp[1]->next;
              }
              if (match) {
                loc=floor(focal*coffset/(res*(ntmp[0]->next->xval-ntmp[2]->next->xval)));
                loc=loc-floor(conv/2);
                for (i=0;i<conv;i++)
                  if ((loc+i)<distmaplen) distmap[loc+i]++;
              }
            }
            ntmp[2]=ntmp[2]->next;
          }
          ntmp[0]=ntmp[0]->next;
        }
      }

      if (outbase!=NULL) {
        for(i=6;i>0;i--)
          outname[outbaselen+6-i]='0'+(char)((frame % (int)pow(10,i))/(int)pow(10,i-1));
        outfile=fopen(outname,"w");
      }
      fprintf(outfile,"%lf\n",res);
      for (i=0;i<distmaplen;i++)
        fprintf(outfile,"%d\n",distmap[i]);
    }
    if (inpbase!=NULL) fclose(inpfile);
    if (outbase!=NULL) fclose(outfile);
    for (i=0;i<3;i++)
      if (edglocs[i]!=NULL) {
        for (y=0;y<ysize;y++)
          freenodelist(edglocs[i][y]);
        free(edglocs[i]);
        edglocs[i]=NULL;
      }
    if ((++frame)>=nframes) break;
  }
  fprintf(stderr,"%i frames processed.\n\n",frame);
  if (inpbase!=NULL) free(inpname);
  if (outbase!=NULL) free(outname);
  if (distmap!=NULL) free(distmap);
}

void freenodelist(node *l) {
  node *t;

  while (l->next!=NULL) {
    t=l->next;
    l->next=t->next;
    free(t);
  }
  free(l);
}
```

Appendix C

PCMCIA Data Acquisition Driver Details

C.1 DAQ Card Hardware

The data acquision card used in this thesis is the ADAC PCM55DIO, which interfaces with the laptop though a type-II PCMCIA socket. The interface hardware within the card consits of an 82C55 digital I/O controller and an Intel 82C54 programmable interval timer. The 82C55 provides three 8-bit digital I/O ports, which are available at the external connector no the card. The 82C54 includes three 16 bit counters, which may be configured to operate in several different modes using interal and external clocks, external clock gating, and cascading. One interrupt is assigned to the card by the PC, and interrupts may be generated by the 82C55, 82C54, or triggered externally.

The memory map of the PCM55DIO is shown in Table C.1. When configured as inputs, writing to the i82C55 ports does nothing. When configured as outputs, reading the ports returns the last byte written. The counters were not needed for the vision system, so support was them was not included in the driver.

Table C.1: PCM55DIO memory map

Address	Read	Write
Base + 0x00	Port A input	Port A output
Base + 0x01	Port B input	Port B output
Base + 0x02	Port C input	Port C output
Base + 0x03	-	Set 82C55 control register
Base + 0x04	Read counter A	Set counter A
Base + 0x05	Read counter B	Set counter B
Base + 0x06	Read counter C	Set counter C
Base + 0x07	-	Set 82C54 control register
Base + 0x07	Read back	Set interrupt and clock sources

The driver uses the three ports of the 8255 for reading back data from the controller board. While programming the ADAPs, ports A and B are written to send data to the contoller board. Whien configuring the imagers, the parallel port data bits are written to send the bitstream, and the parallel port status bits are read to discern the proper timing.

C.2 DAQ Driver Structure

The driver needs to be interrupt driven to achieve the desired data throughput. While it is possible to implement a polling I/O interface in user-level code, interrupt driven routines require kernel privileges. The internal kernel data structure is enumerated in Table C.2. Many of the the high-level interrupt driven functions of the driver need to access parallel port hardware in addition to the data acquisition hardware, so several of the driver parameters must be configured at run time, rather than autodetected when the driver is loaded.

Table C.2: DAQ driver data structure

```
typedef struct pcmd24_info_t {
  dev_node_t node;
  ioaddr_t port;                  // DAQ card base address
  ioaddr_t shipport;              // Parallel port base address
  unsigned char latency;          // Calculation latency
  unsigned char latcnt;
  unsigned char waitcnt;
  unsigned char flag;             // Interrupt state machine state
  unsigned char *inbuf;           // Input data buffer
  int inbuflen;                   // Input buffer length
  int inbufpos;                   // Input buffer pointer
  unsigned char *outbuf;          // Output data buffer
  int outbuflen;                  // Output buffer length
  int outbufpos;                  // Output buffer pointer
  unsigned char intop;            // Interrupt operation
  unsigned char i8255ctl;         // 82C55 control register
  unsigned char intctl;           // DAQ interrupt control register
  unsigned char irqstat;          // DAQ interrupt number
  unsigned char block;
} pcmd24_info_t;
```

The **port** and **irqstat** parameters are autodetected by the driver when it is loaded. **shipport** is the address of the parallel port used for programming the ADAP chips and configuring the imagers. **latency** defines the calculation latency of the ADAP arrays, and how many data samples should be ignored after asserting $NewFrame$. **inbuflen** and **outbuflen** are the lengths of the input and output data buffers respectively, and **inbuf** and **outbut** are the data buffers themselves. **intop** is a set of flags that describes what operations should be performed during each interrupt cycle.

C.3 DAQ driver IOCTL interface

All of the functions of the data acquisition driver are controlled through IOCTLs on the device special file. The complete list of IOCTL definitions is found in the **pcmd24.h** file along with the flag definitions for the bitmapped fields. Some of the more important IOCTLs are described in Table C.3. All IOCTL constants are prepended with "PCMD24_IOCTL_".

Table C.3: DAQ driver IOCTLs

IOCTL	Argument	Return	Description
DISABLIRQ	-	-	Disables interrupts
ENABLEIRQ	-	-	Enables interrupts
RESETRUN	-	-	Reset buffers and enables interrupts
IRQSTAT	-	char	Returns the interrupt number and status
PORTBASE	-	int	Returns the card base address
IRQSRC	char	-	Sets the interrupt source
SETINTOP	char	-	Sets the interrupt operation
GETINTOP	-	char	Returns the interrupt operation
INBUFLEN	int	-	Sets the input buffer size
INBUFCNT	-	int	Returns the input buffer pointer
CLRINBUF	-	-	Resets the input buffer
GETINBUF	char *	-	Reads the input buffer
OUTBUFLEN	int	-	Sets the output buffer size
OUTBUFCNT	-	int	Returns the output buffer pointer
CLROUTBUF	-	-	Resets the output buffer
SETOUTBUF	char *	-	Writes the output buffer
LATENT	char	-	Sets the calculation latency
LATCNT	-	char	Reads the current latency count
RESETLAT	-	-	Resets the latency count
SET8255	char	-	Sets 8255 control register
GET8255	-	char	Reads 8255 control register
SETIN	char	-	Sets selected 8255 ports to inputs (bitmapped)
SETIN*	-	-	Sets selected 8255port to input
SETOUT	char	-	Sets selected 8255 ports to outputs (bitmapped)
SETOUT*	-	-	Sets selected 8255 port to output
READPORT	char	char	Reads the selected I/O port address
READ	char	3*char	Reads the selected 8255 ports (bitmapped)
READ*	-	char	Reads the selected 8255 port
WRITE	4*char	-	Writes to the selected 8255 ports (bitmapped)
WRITE*	char	-	WRites to the selected 8255 port
SHIPPORT	-	char	
ADAPPORT	-	char	Set the parallel port base address
PARPORT	-	char	
SHIPFLAG	char	-	
ADAPFLAG	char	-	Reads the driver state machine flags
FLAG	char	-	

C.4 Interrupt Handler

For this application, the important and timing critical driver functions are implemented in the interrupt handler. That section of the driver code is copied here for reference.

```
static void pcmd24_interrupt(int irq, void *unused, struct pt_regs *regs) {
  char reg,i,bit;
  int len;

  if (pcmd24->intop == INT_PROGCAM) {
    if (pcmd24->flag & SHIP_WAIT) {
      outb(SDEN|SDIN,pcmd24->shipport|SHIP_DAT);
      pcmd24->waitcnt++;
      if (pcmd24->waitcnt == SHIP_WAITLEN) {
        pcmd24->waitcnt = 0;
        pcmd24->flag &= ~SHIP_WAIT;
        if (pcmd24->outbufpos>=pcmd24->outbuflen) {
          pcmd24->flag |= SHIP_DONE;
          outb(0,pcmd24->shipport|SHIP_DAT);
          pcmd24->irqstat&=0x7f;
          disable_irq(pcmd24->irqstat);
        }
      }
      return;
    }
    if (pcmd24->flag & SHIP_START) {
      if (inb(pcmd24->shipport|SHIP_CTL)&SCLK) {
        outb(SDEN,pcmd24->shipport|SHIP_DAT);
        pcmd24->flag &= ~SHIP_START;
        pcmd24->flag |= SHIP_SEND;
      } else
        outb(SDEN|SDIN,pcmd24->shipport|SHIP_DAT);
      return;
    }
    if (pcmd24->flag & SHIP_ACK) {
      if (!(inb(pcmd24->shipport|SHIP_CTL)&SCLK)) {
        outb(SDEN,pcmd24->shipport|SHIP_DAT);
        pcmd24->flag &= ~SHIP_ACK;
      }
      return;
    }
    if (pcmd24->flag & SHIP_STOP) {
      if (inb(pcmd24->shipport|SHIP_CTL)&SCLK) {
        if (pcmd24->waitcnt) {
          outb(SDEN|SDIN,pcmd24->shipport|SHIP_DAT);
          pcmd24->flag &= ~SHIP_STOP;
          pcmd24->flag |= SHIP_WAIT;
          pcmd24->waitcnt=0;
        } else
          pcmd24->waitcnt++;
      } else
        outb(SDEN,pcmd24->shipport|SHIP_CTL);
      return;
    }
    if (pcmd24->flag & SHIP_SEND) {
      if (!(inb(pcmd24->shipport|SHIP_CTL)&SCLK)) {
        bit=((pcmd24->outbuf[pcmd24->outbufpos]>>(7-pcmd24->waitcnt))&0x01)<<7;
        outb(SDEN|bit,pcmd24->shipport|SHIP_DAT);
        pcmd24->waitcnt++;
        if (pcmd24->waitcnt>7) {
          pcmd24->flag |= SHIP_ACK;
          pcmd24->waitcnt=0;
          pcmd24->outbufpos++;
          if (pcmd24->outbufpos >= pcmd24->outbuflen) {
            pcmd24->flag &= ~SHIP_SEND;
            pcmd24->flag |= SHIP_STOP;
          }
```

```
      }
    }
    return;
  }
}

if (pcmd24->intop == INT_PROGADAP) {
  if (pcmd24->flag & ADAP_START) {
    outb(pcmd24->outbuf[0],pcmd24->port|I8255PA);
    outb(pcmd24->outbuf[1],pcmd24->port|I8255PB);
    pcmd24->flag |= ADAP_WAIT;
    return;
  }
  if (pcmd24->flag & ADAP_WAIT) {
    pcmd24->waitcnt++;
    if (pcmd24->waitcnt == ADAP_WAITLEN) {
      pcmd24->waitcnt = 0;
      pcmd24->flag &= ~ADAP_WAIT;
      if (pcmd24->outbufpos>=pcmd24->outbuflen) {
        pcmd24->flag = ADAP_DONE;
        outb(0x00,pcmd24->port|I8255PA);
        outb(0x00,pcmd24->port|I8255PB);
        pcmd24->irqstat&=0x7f;
        disable_irq(pcmd24->irqstat);
      }
    }
    return;
  }
  if (pcmd24->flag & ADAP_SEND) {
    outb(pcmd24->outbuf[pcmd24->outbufpos++],pcmd24->port|I8255PA);
    outb(pcmd24->outbuf[pcmd24->outbufpos++],pcmd24->port|I8255PB);
    if (pcmd24->outbufpos>=pcmd24->outbuflen) {
      pcmd24->flag = ADAP_STOP;
    }
    return;
  }
  if (pcmd24->flag & ADAP_STOP) {
    outb(0x00,pcmd24->port|I8255PA);
    outb(0x00,pcmd24->port|I8255PB);
    pcmd24->flag = ADAP_WAIT;
    return;
  }
}

if (pcmd24->intop == INT_GETFRAME) {
  if (pcmd24->flag & ADAP_START) {
    outb(NEWF,pcmd24->shipport|SHIP_DAT);
    pcmd24->flag &= ~ADAP_START;
    return;
  }
  if (pcmd24->latcnt) {
    if (pcmd24->latcnt == pcmd24->latency)
      outb(0,pcmd24->shipport|SHIP_DAT);
    pcmd24->latcnt--;
    return;
  }
  len=pcmd24->inbuflen/3;
  pcmd24->inbuf[pcmd24->inbufpos]=inb(pcmd24->port|I8255PA);
  pcmd24->inbuf[pcmd24->inbufpos+len]=inb(pcmd24->port|I8255PB);
  pcmd24->inbuf[pcmd24->inbufpos+len*2]=inb(pcmd24->port|I8255PC);
  pcmd24->inbufpos++;
  if (pcmd24->inbufpos>len) {
    pcmd24->flag |= ADAP_DONE;
    pcmd24->irqstat&=0x7f;
    disable_irq(pcmd24->irqstat);
  }
  return;
}
```

```
  if (pcmd24->latcnt) {
    pcmd24->latcnt--;
    return;
  }

  if ((pcmd24->intop&INT_READMASK)&&(pcmd24->inbuf!=NULL)) {
    if (pcmd24->intop&INT_READA) {
      reg=inb(pcmd24->port|I8255PA);
      if (pcmd24->inbufpos<pcmd24->inbuflen)
        pcmd24->inbuf[pcmd24->inbufpos++]=reg;
    }
    if (pcmd24->intop&INT_READB) {
      reg=inb(pcmd24->port|I8255PB);
      if (pcmd24->inbufpos<pcmd24->inbuflen)
        pcmd24->inbuf[pcmd24->inbufpos++]=reg;
    }
    if (pcmd24->intop&INT_READC) {
      reg=inb(pcmd24->port|I8255PC);
      if (pcmd24->inbufpos<pcmd24->inbuflen)
        pcmd24->inbuf[pcmd24->inbufpos++]=reg;
    }
  }

  if ((pcmd24->intop&INT_WRITEMASK)&&(pcmd24->outbuf!=NULL))
    for (i=0;i<3;i++)
      if (pcmd24->intop&(1<<(4+i)))
        if (pcmd24->outbufpos<pcmd24->outbuflen)
          outb(pcmd24->outbuf[pcmd24->outbufpos++],pcmd24->port|i);
}
```

Appendix D

ADAP And Camera Programming Software

The source code listed here uses the facilities of the data acquisition card drivers to program the PhotoBit Imagers and ADAP processors.

D.1 progcam.c

The Photobit imager has programmable registers internally that control the behavior of the imager. At power up, the imager outputs 8 bit image data at a rate of 30 512 x 384 frames per second. The default resolution is higher than necessary to calculate depth maps and provides more data than the ADAP chips can process in the given amount of time. For proper execution, the PB159 is programmed to output a 64 x 64 pixel window at the center of the image. Other features such as exposure time and shutter control are not modified.

The imager is programmed through a routine called SHIP. The imager has a serial clock input and a bidirectional data port. The data is latched on the rising edge of the clock, and should be held stable while the clock is high. The imager is programmed by sending sending a serial bit stream over the data line. Programming is initiated by sending a start signal, followed by an chip address byte, a register address byte, and the data to be written to that address. After each byte is sent, the imager acknowledges by pulling the data pin low for a clock cycle [5].

To ensure precise timing, an interrupt-driven routine is used. Data is sent to the controller board on the falling rising edge of the ADAP **PHI2** signal, which is also used as the timing reference for the serial clock. The following code reads a program data file and performs the necessary programming algorithm.

```
// progcam.c
// Programs the PhotoBit PB159DM imagers thruogh the digital controller board
// Mark Spaeth, February 1999

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <asm/io.h>
#include "../pcmd24_cs/pcmd24.h"
#include <fcntl.h>

static char *usage="\
```

```
Usage: %s <pb159 program> [lpt port]\n\n\
Configures a pb159 image through the ADAP controller board\n\n";

int readline(FILE *, char *, int *);

void main(int argc, char *argv[]) {
  int pcmdev;
  short port=0x378;
  char *prog, *inpline, done;
  FILE *file;
  int line,ret,len;
  int i,j,k,c;

  done=0;
  line=0;
  len=0;
  prog=(char *)malloc(sizeof(char));
  inpline=(char *)malloc(256*sizeof(char));
  printf("%x\n",inpline);

  if (argc==1) {
    fprintf(stderr,usage,argv[0]);
    exit(1);
  }

  if ((file=fopen(argv[1],"r"))==NULL) {
    fprintf(stderr,"File not found: %s\n\n",argv[1]);
    fprintf(stderr,usage,argv[0]);
    exit(1);
  }

  if (argc>2) {
    if ((sscanf(argv[1],"%x",&port)!=1)||port<0x100) {
      fprintf(stderr,"Invalid LPT port address\n\n");
      fprintf(stderr,usage,argv[0]);
      exit(1);
    }
  }
  fprintf(stderr,"Using LPT port at 0x%3x\n",port);

  if ((pcmdev=open("/dev/dio0",O_RDONLY))==-1) {
    fprintf(stderr,"Unable to open DIO device\n");
    exit(1);
  }
  fprintf(stderr,"Using DIO port at 0x%3x\n",ioctl(pcmdev,PCMD24_IOCTL_PORTBASE,NULL));

  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_DISABLIRQ,NULL))<0)
    fprintf(stderr,"DisableIRQ failed\n");
  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_READPORT,port+1))<0)
    fprintf(stderr,"ReadPort failed\n");
  if (ret&PROG) {
    fprintf(stderr,"Error: Controller board is not in program mode\n");
    exit(1);
  }
  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_IRQSRC,INT_EXT))<0)
    fprintf(stderr,"IRQSrc failed\n");
  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_LATENT,0))<0)
    fprintf(stderr,"Latent failed\n");
  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_SHIPPORT,port))<0)
    fprintf(stderr,"SHIPPort failed\n");
  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_SETINTOP,INT_PROGCAM))<0)
    fprintf(stderr,"SetIntOp failed\n");

  while (!done) {
    if (readline(file,inpline,&line)==-1)
      exit(1);
    if (!strncmp(inpline,"START",5))
      continue;
```

```
    if (!strncmp(inpline,"STOP",4))
      continue;
    if (!strncmp(inpline,"WAIT",4))
      continue;
    if (!strncmp(inpline,"END",3)) {
      done=1;
      continue;
    }

    if (sscanf(inpline,"0x%x",&c)==1) {
      prog=(char *)realloc(prog,(len+1)*sizeof(char));
      prog[len++]=c;
      continue;
    }

    fprintf(stderr,"Error: Unrecognized command at line %d: %s\n",line,inpline);
    exit(1);
  }

  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_OUTBUFLEN,len))<0)
    fprintf(stderr,"OutBufLen failed\n");
  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_SETOUTBUF,prog))<0)
    fprintf(stderr,"SetOutBuf failed\n");

  printf("Programming the PB-159DM...\n");
  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_RESETRUN,NULL))<0)
    fprintf(stderr,"ResetRun failed\n");

  while (!(ioctl(pcmdev,PCMD24_IOCTL_SHIPFLAG,NULL)&SHIP_DONE))
    usleep(1000);

  printf("Programming complete...\n");
  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_DISABLIRQ,NULL))<0)
    fprintf(stderr,"DisableIRQ failed\n");

  free(prog);
  free(inpline);
  fclose(file);
  close(pcmdev);
}


int readline(FILE *inpfile, char *line, int *lineno) {
  char *tmp, *inpline;

  inpline=(char *)malloc(256*sizeof(char));
  do {
    if (fgets(inpline,256,inpfile)==NULL) {
      fprintf(stderr,"Unexpected EOF in data file\n");
      return -1;
    }
    (*lineno)++;
    tmp=inpline;
    while((*tmp==' ')||(*tmp=='\t')) tmp++;
    if ((*tmp=='#')||(*tmp=='\n')||(*tmp=='\0')) {
      strcpy(line,"");
    } else {
      strcpy(line,tmp);
    }
  } while (!strlen(line));
  free(inpline);
  return 0;
}
```

D.2 progadap.c

As previously described, each ADAP array cell contains a 35-bit shift register which controls the operation of the ALU and the interconnections in the switch matrix. The output of each cell's shift register is connected to the input of the register in the cell immediately east, so a programming signal consists of five parallel 175-bit sequences sent into the five data in ports on the ADAP. To program the ADAP, the **LOAD** signal is raised and the data is sent in, as clocked by the **PHI2** signal. Special care must be taken to lower the **LOAD** signal at the correct time or the data will be misaligned in the shift registers, and the program will be wrong [6].

Like the camera programming code, an interrupt driven routine is used here to ensure precise timing. This program supports programming the the two ADAP chips on each board independently or together. The following code reads in program files for either or both ADAP chips and programs them.

```
// progadap.c
// Programs the ADAP arrays using the digital controller board
// Mark Spaeth, March 1999

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../pcmd24_cs/pcmd24.h"
#include <errno.h>
#include <fcntl.h>

#define NBITS 35

int readline(FILE *, char *, int *);

static char *usage="
Usage : %s [-lpt <port>] [-a <prog>] [-b <prog>]
Programs the 2 ADAPs on the early vision board\n\n";

void main(int argc, char *argv[]) {
  int pcmdev;
  int port=0x378, lpt=1;
  char *inpnamea=NULL, *inpnameb=NULL;
  FILE *inpfilea, *inpfileb;
  char *arg, *tmp, *prog;
  int dataa=0,datab=0;
  int line,h,w,ret;
  int i,j,k,l;

  if (argc==1) {
    fprintf(stderr,usage,argv[0]);
    exit(1);
  }

  for (i=1;i<argc;i++) {
    arg=argv[i];
    if (*arg!='-') {
      fprintf(stderr,"Error: Invalid argument -- %s\n\n",arg);
      fprintf(stderr,usage,argv[0]);
      exit(1);
    }
    arg++;
    if (!strcmp(arg,"lpt")) {
      if (++i<argc) {
        arg=argv[i];
        if (sscanf(arg,"%d",&lpt)==1) {
          switch (lpt) {
          case 1:
```

```
          port=0x378;
          break;
        case 2:
          port=0x278;
          break;
        case 3:
          port=0x3BC;
          break;
        default:
          fprintf(stderr,"Error: Invalid LPT port identifier\n");
          fprintf(stderr,usage,argv[0]);
          exit(1);
          break;
        }
      } else {
        fprintf(stderr,"Error: Unable to read LPT port identifier\n");
        fprintf(stderr,usage,argv[0]);
        exit(1);
      }
    } else {
      fprintf(stderr,"Error: Missing LPT port identifier\n");
      fprintf(stderr,usage,argv[0]);
      exit(1);
    }
  } else if (!strcmp(arg,"a")) {
    if (++i<argc) {
      inpnamea=argv[i];
    } else {
      fprintf(stderr,"Error: Missing file name\n");
      fprintf(stderr,usage,argv[0]);
      exit(1);
    }
  } else if (!strcmp(arg,"b")) {
    if (++i<argc) {
      inpnameb=argv[i];
    } else {
      fprintf(stderr,"Error: Missing file name\n");
      fprintf(stderr,usage,argv[0]);
      exit(1);
    }
  } else {
    fprintf(stderr,"Error: Unrecognized argument : -%s\n",arg);
    fprintf(stderr,usage,argv[0]);
    exit(1);
  }
}

if ((inpnamea==NULL)&&(inpnameb==NULL)) {
  fprintf(stderr,"Error: No input program files specified\n");
  fprintf(stderr,usage,argv[0]);
  exit(1);
}

fprintf(stderr,"Using LPT port at 0x%3x\n",port);

if ((pcmdev=open("/dev/dio0",O_RDONLY))==-1) {
  fprintf(stderr,"Unable to open DIO device\n");
  exit(1);
}
fprintf(stderr,"Using DIO port at 0x%3x\n",ioctl(pcmdev,PCMD24_IOCTL_PORTBASE,NULL));

ioctl(pcmdev,PCMD24_IOCTL_DISABLIRQ,NULL);
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_READPORT,port+1))<0)
  fprintf(stderr,"ReadPort failed\n");
if (ret&PROG) {
  fprintf(stderr,"Error: Controller board is not in program mode\n");
  exit(1);
}
```

```
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_IRQSRC,INT_EXT))<0)
  fprintf(stderr,"IRQSrc failed\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_LATENT,0))<0)
  fprintf(stderr,"Latent failed\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_SHIPPORT,port))<0)
  fprintf(stderr,"SHIPPort failed\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_SETOUTA,NULL))<0)
  fprintf(stderr,"SetOutA failed\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_WRITEA,0))<0)
  fprintf(stderr,"WriteA failed\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_SETOUTB,NULL))<0)
  fprintf(stderr,"SetOutB failed\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_WRITEB,0))<0)
  fprintf(stderr,"WriteB failed\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_SETINC,NULL))<0)
  fprintf(stderr,"SetInC failed\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_SETINTOP,INT_PROGADAP))<0)
  fprintf(stderr,"SetIntOp failed\n");

tmp=(char *)malloc(128*sizeof(char));
if (inpnamea!=NULL) {
  if ((inpfilea=fopen(inpnamea,"r"))!=NULL) {
    line=0;
    if (readline(inpfilea,tmp,&line)==EOF) exit(1);
    if (sscanf(tmp,"array %d %d\n",&w,&h)!=2) {
      fprintf(stderr,"Error: Missing array declaration in file %s\n",inpnamea);
      exit(1);
    }
    if ((w!=5)||(h!=5)) {
      fprintf(stderr,"Error: Array size not 5x5 in file %s\n",inpnamea);
      exit(1);
    }
  } else {
    fprintf(stderr,"Unable to open file %s\n",inpnamea);
    exit(1);
  }
}
if (inpnameb!=NULL) {
  if ((inpfileb=fopen(inpnameb,"r"))!=NULL) {
    line=0;
    if (readline(inpfileb,tmp,&line)==EOF) exit(1);
    if (sscanf(tmp,"array %d %d\n",&w,&h)!=2) {
      fprintf(stderr,"Error: Missing array declaration in file %s\n",inpnameb);
      exit(1);
    }
    if ((w!=5)||(h!=5)) {
      fprintf(stderr,"Error: Array size not 5x5 in file %s\n",inpnameb);
      exit(1);
    }
  } else {
    fprintf(stderr,"Unable to open file %s\n",inpnameb);
    exit(1);
  }
}
free(tmp);

prog=(char *)malloc((2*5*NBITS)*sizeof(char));
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_WRITEA,inpnamea==NULL ? 0x00 : 0x20))<0)
  fprintf(stderr,"WriteA failed\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_WRITEB,inpnameb==NULL ? 0x00 : 0x20))<0)
  fprintf(stderr,"WriteB failed\n");

k=0;
dataa=0;
datab=0;
for(i=0;i<5;i++) {
  for(j=0;j<NBITS;j++) {
```

```
      if (inpnamea!=NULL) {
        if (fscanf(inpfilea,"%d",&dataa)!=1) {
          fprintf(stderr,"Error: Can't read bit data from file %s %d %d\n",inpnamea,i,j);
          exit(1);
        }
        dataa&=0x1f;
        dataa|=0x20;
      }
      if (inpnameb!=NULL) {
        if (fscanf(inpfileb,"%d",&datab)!=1) {
          fprintf(stderr,"Error: Can't read bit data from file %s %d %d\n",inpnameb,i,j);
          exit(1);
        }
        datab&=0x1f;
        datab|=0x20;
      }
      prog[k++]=dataa;
      prog[k++]=datab;
    }
  }
  if (inpnamea!=NULL) fclose(inpfilea);
  if (inpnameb!=NULL) fclose(inpfileb);

  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_OUTBUFLEN,2*5*NBITS))<0)
    fprintf(stderr,"OutBufLen failed\n");
  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_SETOUTBUF,prog))<0)
    fprintf(stderr,"SetOutBuf failed\n");

  printf("Programming the ADAP chips...\n");
  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_RESETRUN,NULL))<0)
    fprintf(stderr,"ResetRun failed\n");

  while (!(ioctl(pcmdev,PCMD24_IOCTL_ADAPFLAG,NULL)&ADAP_DONE))
    usleep(1000);

  printf("Programming complete...\n");
  ioctl(pcmdev,PCMD24_IOCTL_DISABLIRQ,NULL);

  free(prog);
  close(pcmdev);
}

int readline(FILE *inpfile, char *line, int *lineno) {
  char *tmp, *inpline;

  inpline=(char *)malloc(256*sizeof(char));
  do {
    if (fgets(inpline,256,inpfile)==NULL) {
      fprintf(stderr,"Unexpected EOF in data file\n");
      return EOF;
    }
    (*lineno)++;
    tmp=inpline;
    while((*tmp==' ')||(*tmp=='\t')) tmp++;
    if ((*tmp=='#')||(*tmp=='\n')||(*tmp=='\0')) {
      strcpy(line,"");
    } else {
      strcpy(line,tmp);
    }
  } while (!strlen(line));
  free(inpline);
  return 0;
}
```

Appendix E

XDepth Software

```
#include "xdepth.h"

int main (int argc, char * argv) {

  /* IPC data */
  char *shmptr;
  int shmsize;

  /* Device data */
  int pcmdev;
  short lptport=0x378;

  /* Configuration parms */
  char latency;

  /* Globals */
  char *tmp, *arg;
  char v=0;
  int ret, argn;

  /* Signals */
  struct sigaction sigact;

  sigact.sa_handler=inthandler;
  sigemptyset(&sigact.sa_mask);
  sigact.sa_flags=0;
  sigaction(SIGINT, &sigact, 0);

  v=0;
  shmid=0;
  quit=NULL;
  imgh=64;
  imgw=64;
  xscale=4;
  nbins=128;
  sprscale=9;
  pidread=getpid();
  pidedge=0;
  pidspr=0;
  pidcorr=0;
  pidplot=0;
  latency=22;

  argn=1;
  while(argn<argc) {
    arg=argv[argn];
    if ((*arg)!='-')
      die("Invalid argument format");
    switch (*(++arg)) {
```

```
    case 'v':
      v=1;
      break;
    case 'l':
      if (++argn<argc)
        latency=atoi(argv[argn]);
      else
        die("Invalid argument format");
      break;
    case 'x':
      if (++argn<argc)
        xscale=atoi(argv[argn]);
      else
        die("Invalid argument format");
      break;
    default:
      die("Unrecognized argument");
   }
   argn++;
}

if ((pcmdev=open("/dev/dio0",O_RDONLY))==-1)
  die("Unable to open DIO device");
ioctl(pcmdev,PCMD24_IOCTL_DISABLIRQ,NULL);

shmsize=(7*imgh*imgw*sizeof(char))+4; // (2*imgw*sizeof(char));
if (shmsize>0x20000)
  die("Shared memory region is too large");

if ((shmid=shmget(IPC_PRIVATE, shmsize, SHM_R|SHM_W))<0)
  die("shmget failed");

if ((shmptr=shmat(shmid,0,0))==(char *)-1)
  die("shmat failed (read)");

if (v) printf("Shm range: %x %x\n",shmptr,shmptr+shmsize-1);

if (v) printf("Defining arrays\n");
img[0]=tmp=(char *)shmptr;
img[1]=tmp=(char *)(tmp+(imgh*imgw*sizeof(char)));
img[2]=tmp=(char *)(tmp+(imgh*imgw*sizeof(char)));
spr[0]=tmp=(char *)(tmp+(imgh*imgw*sizeof(char)));
spr[1]=tmp=(char *)(tmp+(imgh*imgw*sizeof(char)));
spr[2]=tmp=(char *)(tmp+(imgh*imgw*sizeof(char)));
dep   =tmp=(char *)(tmp+(imgh*imgw*sizeof(char)));
tailr =tmp=(char *)(tmp+(imgh*imgw*sizeof(char)));
taile =tmp=(char *)(tmp+sizeof(char));
tails =tmp=(char *)(tmp+sizeof(char));
quit  =tmp=(char *)(tmp+sizeof(char));

if (v) printf("Setting parms\n");
(*tailr)=0;
(*taile)=0;
(*tails)=0;
(*quit)=0;

if (v) fprintf(stdout,"Forking output\n");

if ((pidplot=fork())<0)
  die("fork error (output)");

if (pidplot==0) {
  dox();
  if (shmctl(shmid,IPC_RMID,0)<0)
    die("shmctl error (output)");
  return 0;
}
```

```
if (v) fprintf(stdout,"Forking edge\n");

if ((pidedge=fork())<0)
  die("fork error (edge)");


if (pidedge==0) {
  spredge();
  if (shmctl(shmid,IPC_RMID,0)<0)
    die("shmctl error (edge)");
  return 0;
}

if (v) fprintf(stdout,"Forking corr\n");

if ((pidcorr=fork())<0)
  die("fork error (corr)");


if (pidcorr==0) {
  correlate();
  if (shmctl(shmid,IPC_RMID,0)<0)
    die("shmctl error (corr)");
  return 0;
}

if (v) fprintf(stdout,"Using LPT port at 0x%x\n",lptport);

if (v) fprintf(stdout,"Using DIO port at 0x%x\n",ioctl(pcmdev,PCMD24_IOCTL_PORTBASE,NULL));

if (v) fprintf(stdout,"DIO using interrupt %d\n",ioctl(pcmdev,PCMD24_IOCTL_IRQSTAT));

if ((ret=ioctl(pcmdev,PCMD24_IOCTL_READPORT,lptport|1))<0)
  die("ReadPort failed");

if (~ret&PROG) {
  shmctl(shmid,IPC_RMID,0);
  die("Controller board is not in run mode");
} else
  if (v) fprintf(stdout,"Controller board is in run mode\n");

if (v) fprintf(stdout,"IRQSrc\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_IRQSRC,INT_EXT))<0)
  die("IRQSrc failed");

if (v) fprintf(stdout,"Latent\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_LATENT,latency))<0)
  die("Latent failed");

if (v) fprintf(stdout,"InBufLen\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_INBUFLEN,3*imgh*imgw))<0)
  die("InBufLen failed");

if (v) fprintf(stdout,"SetIn\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_SETIN,PORTA|PORTB|PORTC))<0)
  die("SetIn failed");

if (v) fprintf(stdout,"ParPort\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_PARPORT,lptport))<0)
  die("ParPort failed");

if (v) fprintf(stdout,"SetIntOp\n");
if ((ret=ioctl(pcmdev,PCMD24_IOCTL_SETINTOP,INT_GETFRAME))<0) {
  fprintf(stderr,"%d\n",ret);
  die("SetIntOp failed");
}

while (!(*quit)) {
```

```
    if (v) fprintf(stdout,"ResetRun\n");
    if ((ret=ioctl(pcmdev,PCMD24_IOCTL_RESETRUN,NULL))<0)
      die("ResetRun failed");

    if (v) fprintf(stdout,"WaitDone\n");
    while (!(ioctl(pcmdev,PCMD24_IOCTL_FLAG,NULL)&ADAP_DONE)&&!(*quit))
      usleep(1000);

    if (v) fprintf(stdout,"WaitFrame\n");
    while ((*tailr)&&!(*quit))
      usleep(1000);

    if (v) fprintf(stdout,"GetInBuf\n");
    if ((ret=ioctl(pcmdev,PCMD24_IOCTL_GETINBUF,shmptr))<0)
      die("GetInBuf failed");

    (*tailr)+=imgh;
  }

  if ((ret=ioctl(pcmdev,PCMD24_IOCTL_DISABLIRQ,NULL))<0)
    if (v) fprintf(stderr,"Error: DisablIRQ failed");

  close(pcmdev);
  if (shmctl(shmid,IPC_RMID,0)<0)
    die("shmctl error (main)");
  return 0;
}

/*****************************************************************/

void die(char *msg) {
  fprintf(stderr,"Error: %s\n",msg);
  if (quit!=NULL) (*quit)=1;
  exit(1);
}

/*****************************************************************/

void inthandler(int sig) {
  shmctl(shmid,IPC_RMID,0);
  die("Caught SIGINT\n");
}
```

Appendix F

Schematics

F.1 ADAP Early Vision Board

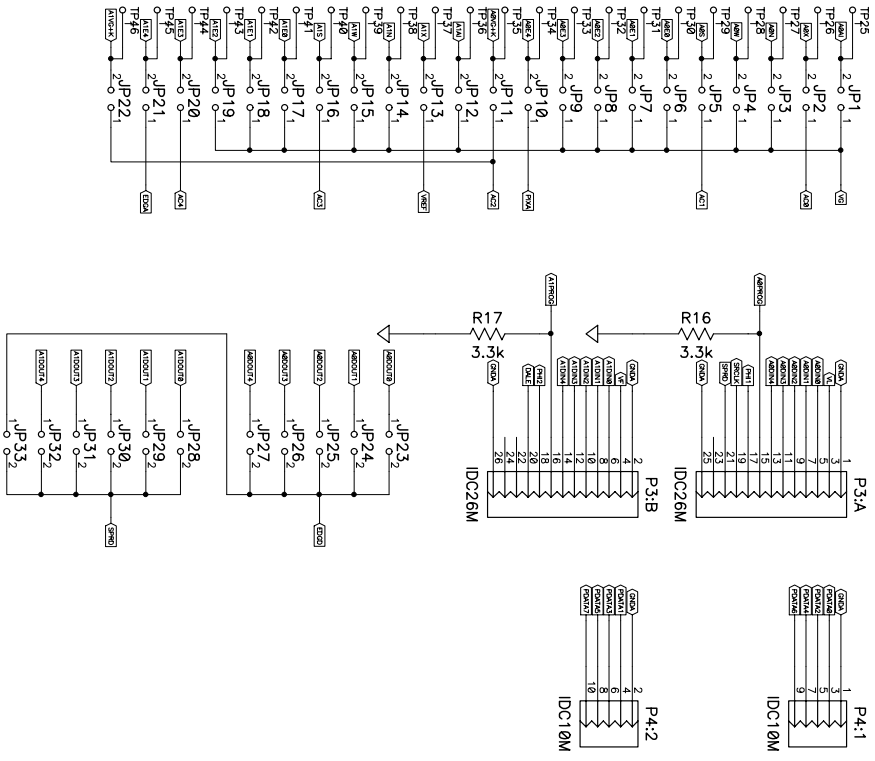

Figure F-1: ADAP board I/O ports

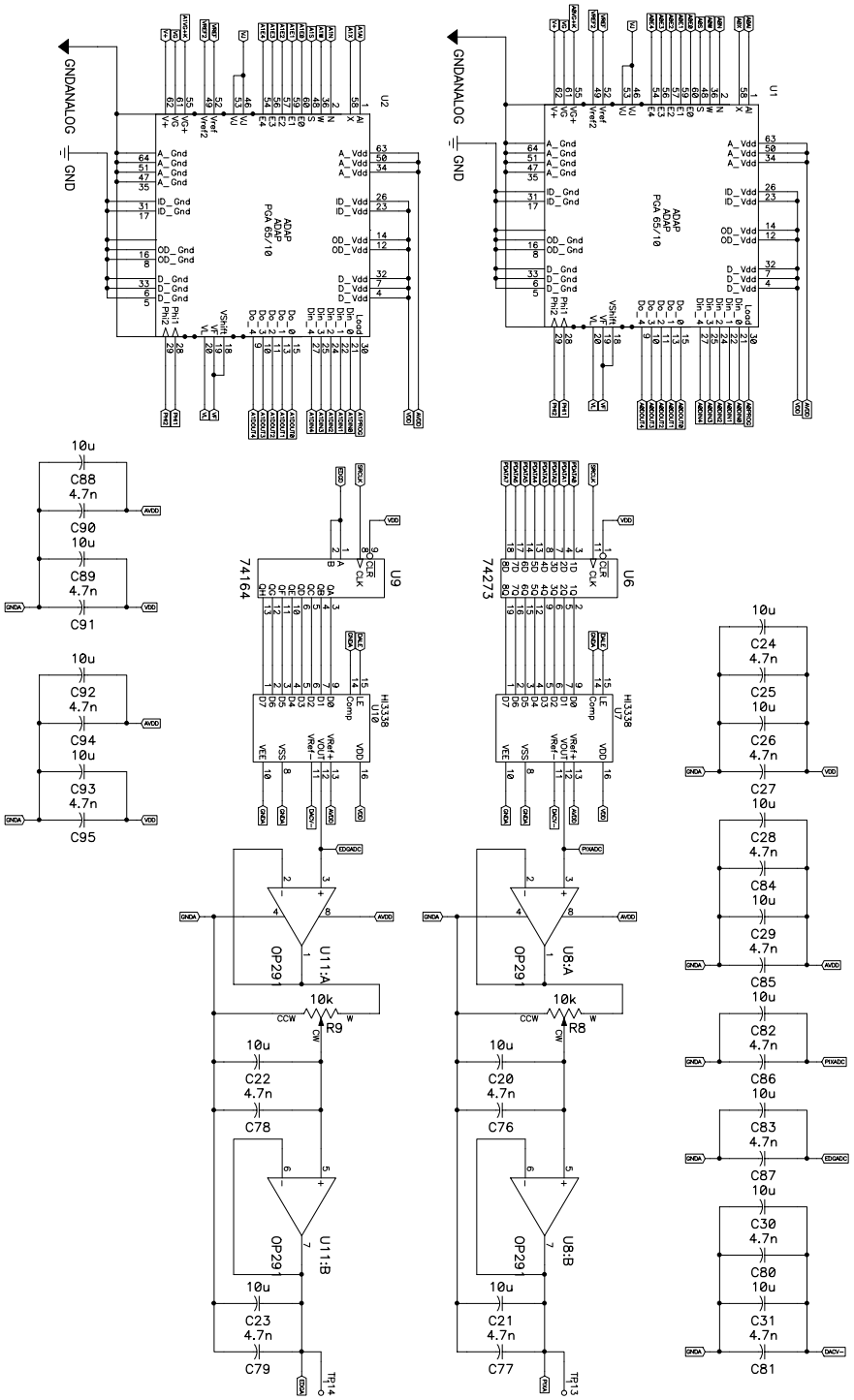

Figure F-2: ADAP board ADAP and interface circuitry

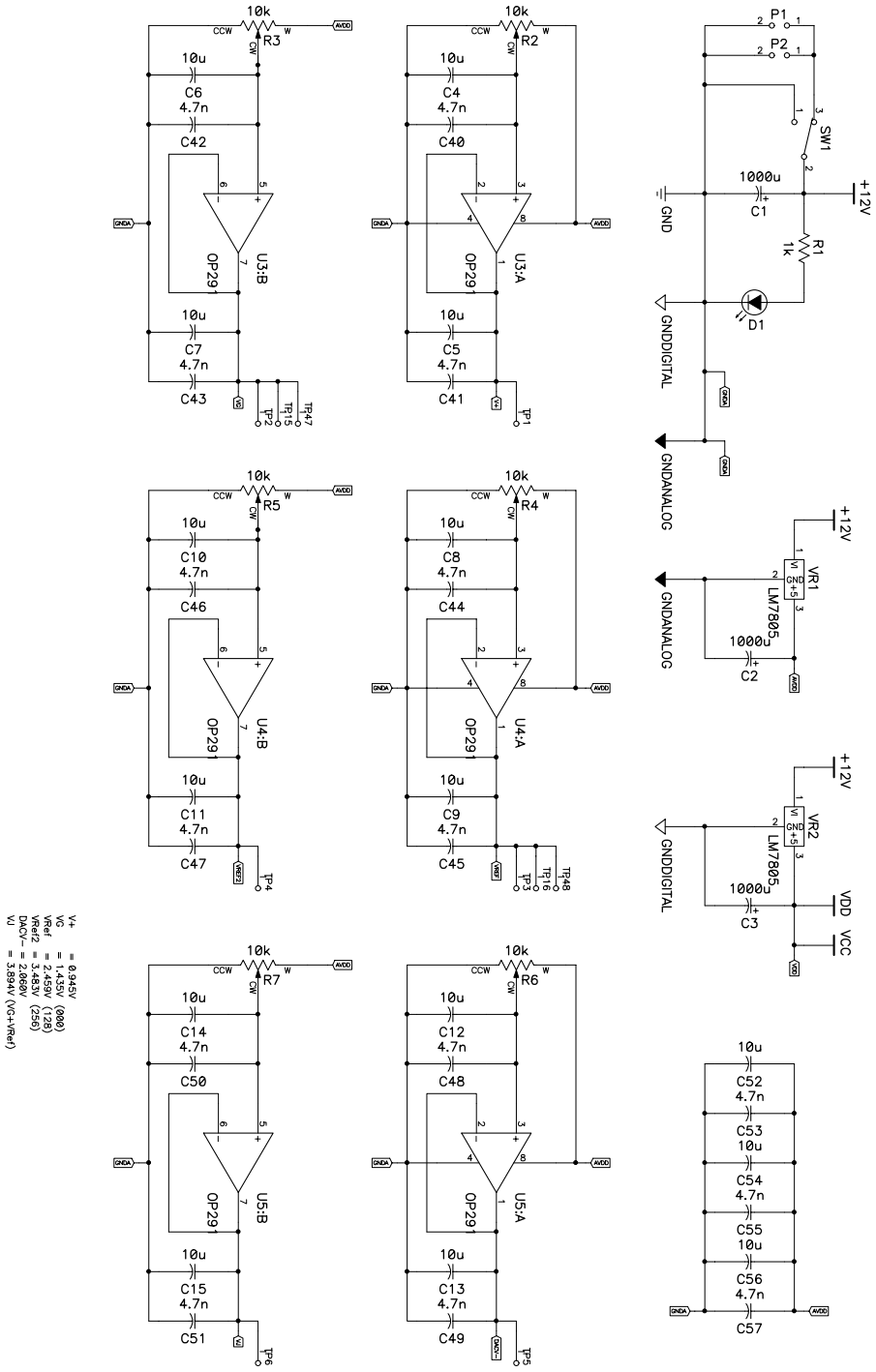

Figure F-3: ADAP board power and bias circuitry

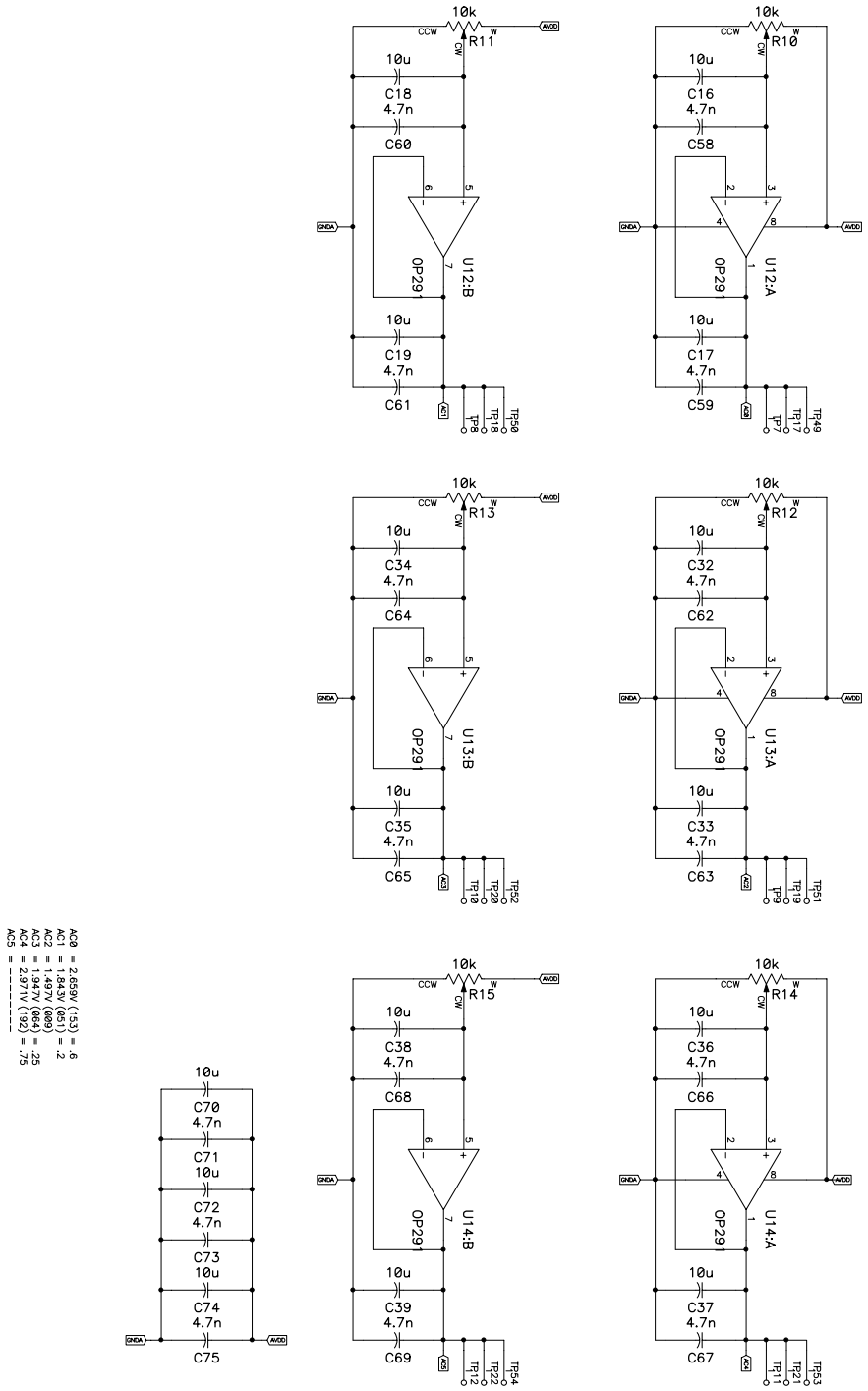

Figure F-4: ADAP board analog constants

F.2 Imager Board

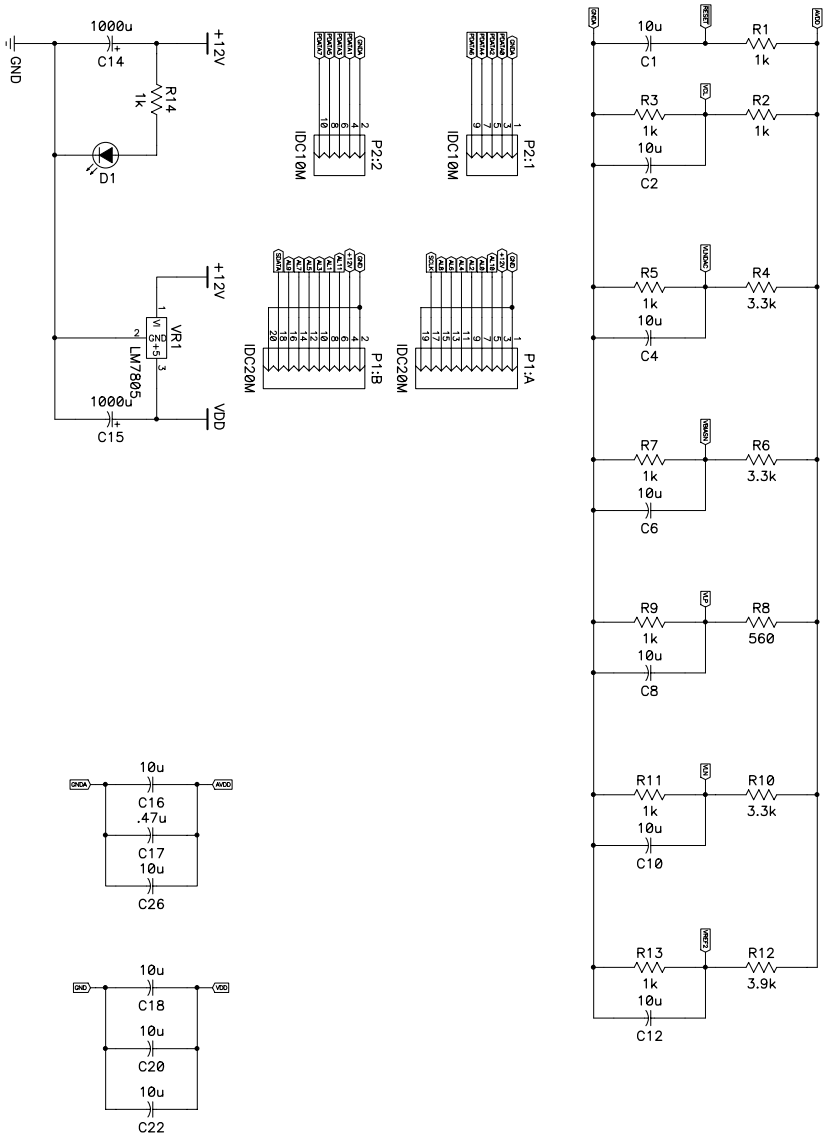

Figure F-5: Imgaer board bias, power, and I/O ports

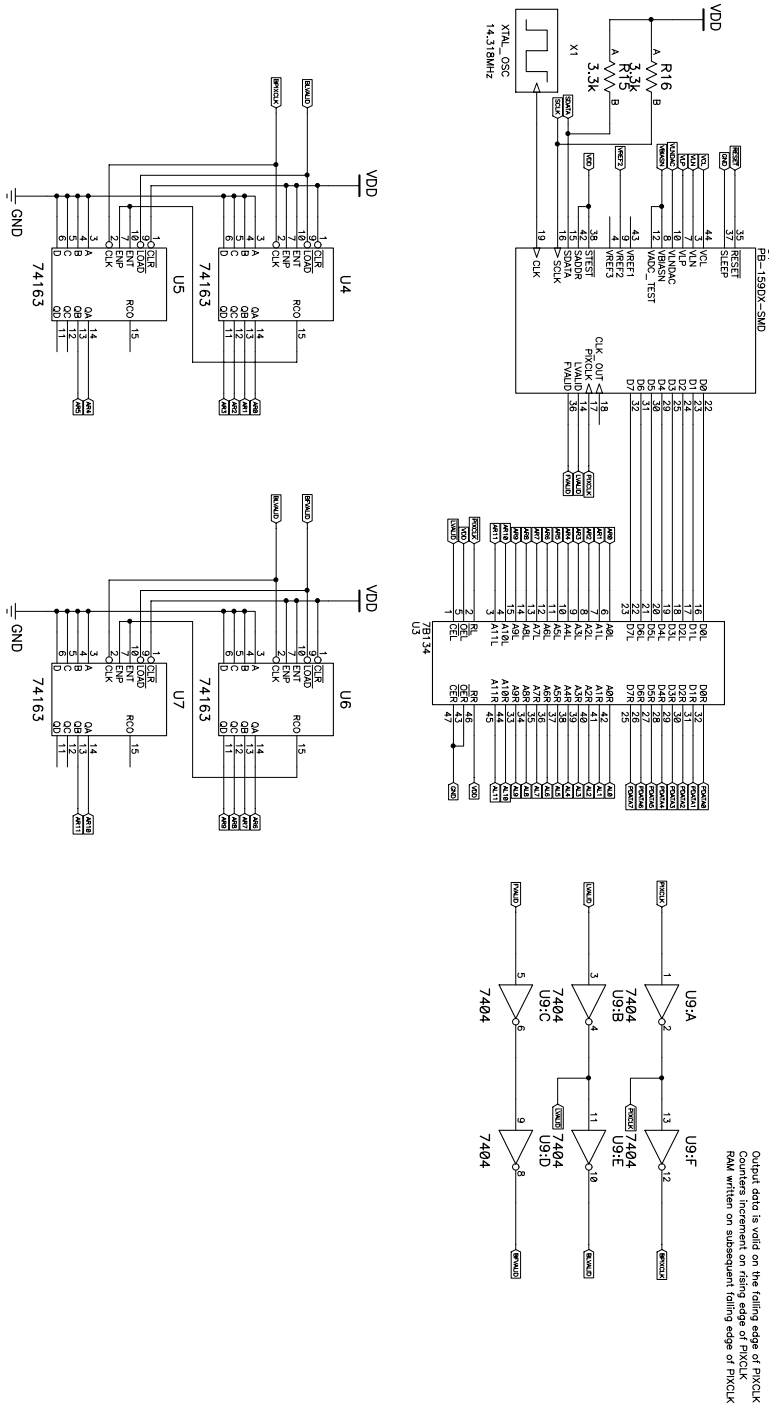

Figure F-6: Imager board imager, address, and SRAM circuitry

F.3 PLD Board

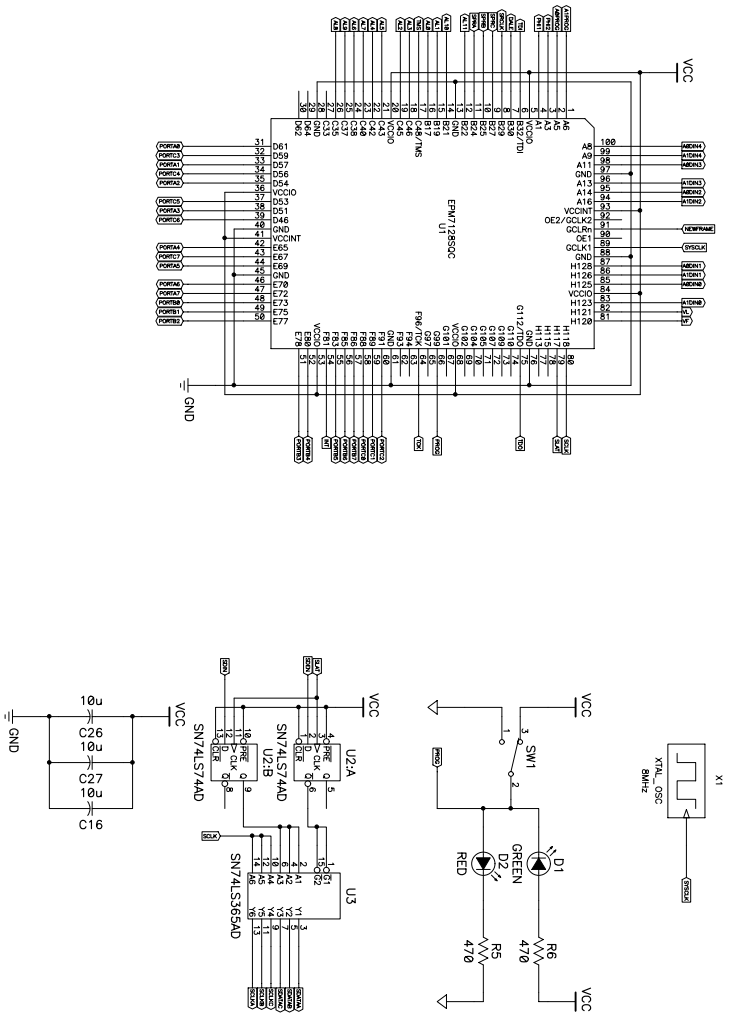

Figure F-7: PLD board PLD interface

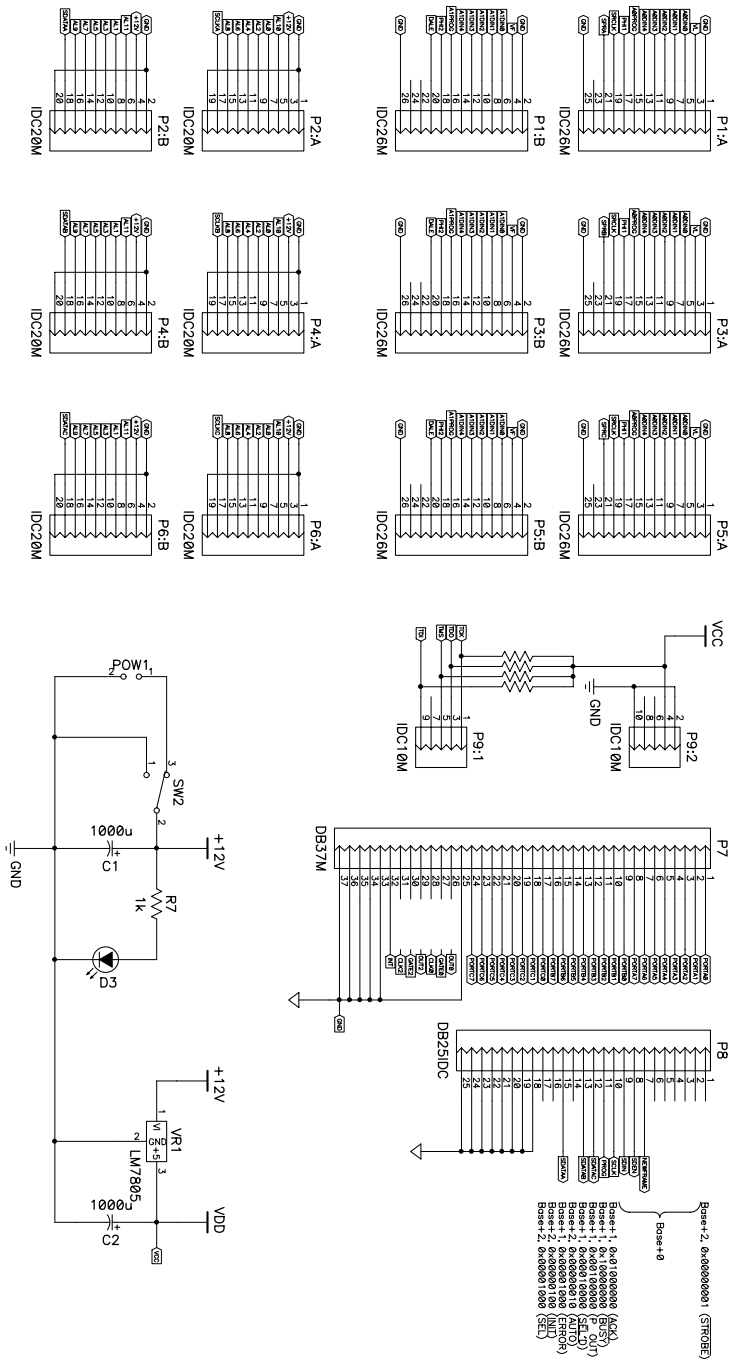

Figure F-8: PLD board power and I/O ports

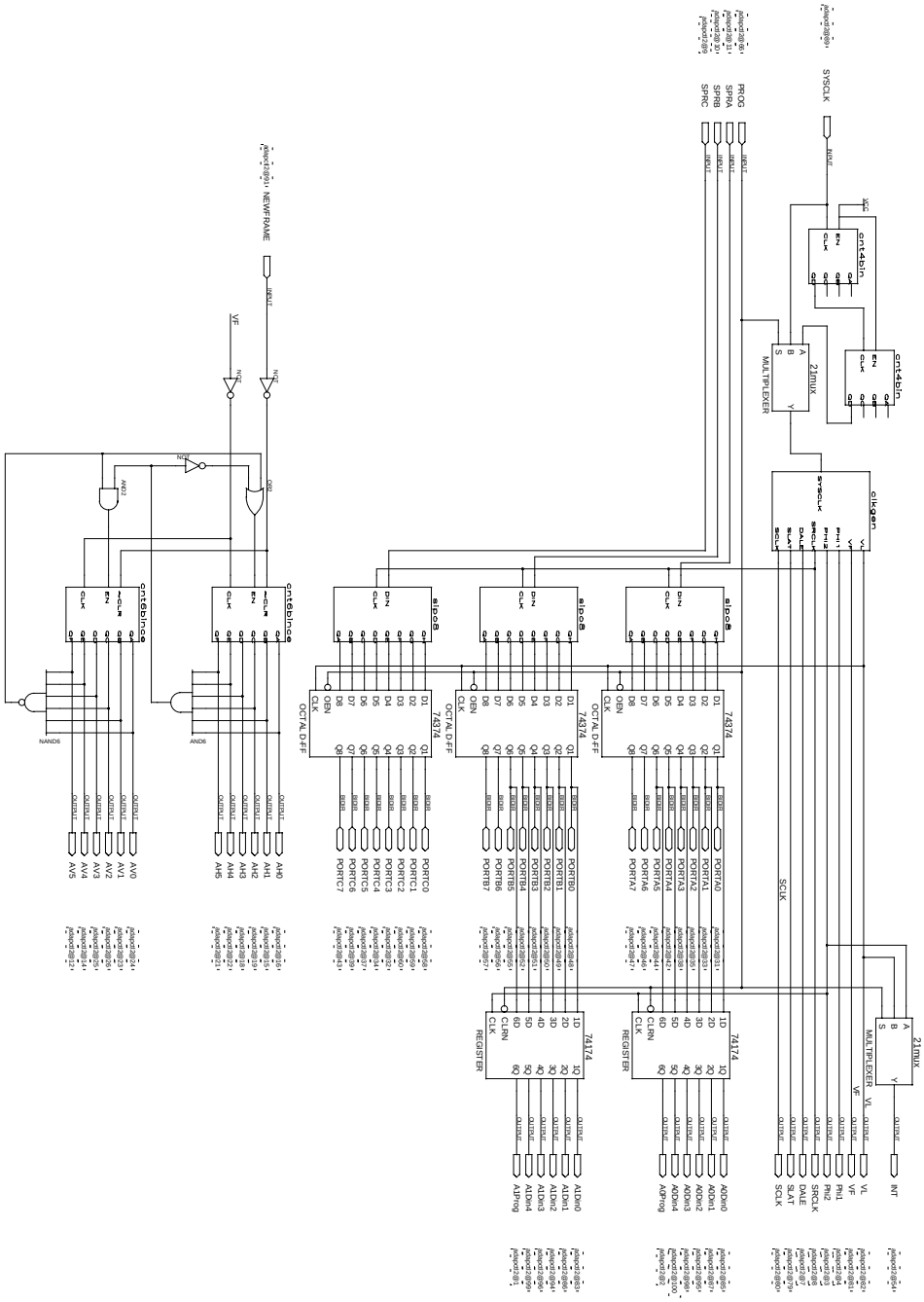

Figure F-9: PLD internal circuitry

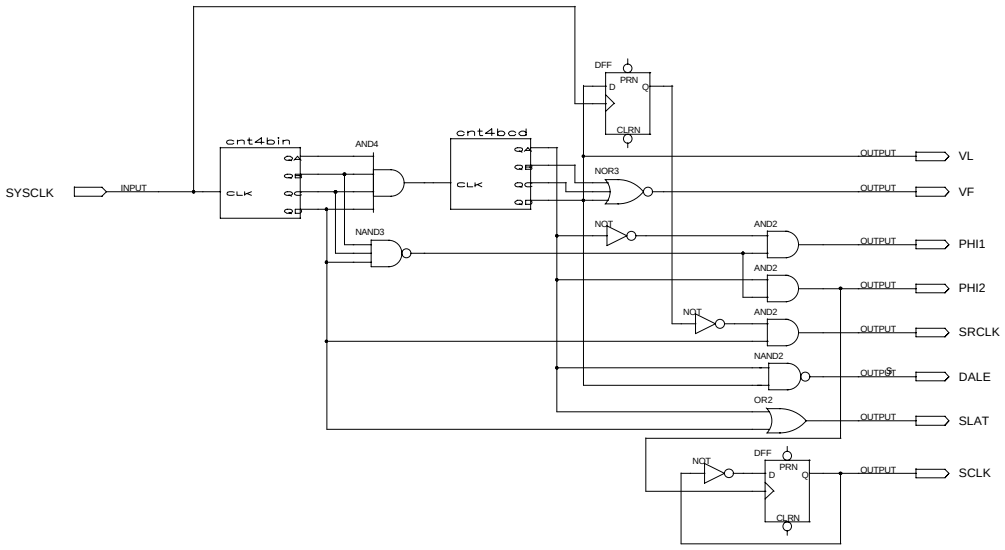

Figure F-10: PLD clock generator circuitry

Bibliography

[1] C. L. Keast. *An Integrated Image Acquisition, Smoothing, and Segmentation Focal Plane Processing.* Phd thesis, Massachusetts Institute of Technology, Deartment of Electrical Engineering and Computer Science, February 1992.

[2] D. L. Standley. An object position and orientation ic with embedded imager. *IEEE Journal of Solid-State Circuits*, pages 1853–1859, February 1992.

[3] Steven J. Decker. *A Wide Dynamic Range CMOS Imager With Parallel On-Chip Analog-to-Digital Conversion.* Phd thesis, Massachusetts Institute of Technology, Deartment of Electrical Engineering and Computer Science, September 1997.

[4] S. A. Paul and H.-S. Lee. A 9-b charge-to-digital converter for integrated image sensors. *IEEE Journal of Solid-State Circuits*, pages 1931–1938, December 1996.

[5] PhotoBit Corporation. Pb159dx product specifications. http://www.photobit.com, August 1998.

[6] David A. Martin. *ADAP: A Mixed-Signal Array Processor With Early Vision Applications.* Phd thesis, Massachusetts Institute of Technology, Deartment of Electrical Engineering and Computer Science, August 1996.

[7] Berthold K. P. Horn. *Robot Vision.* MIT Press, Cambridge, Massachusetts, 1986.

[8] Jae S. Lim. *Two-Dimensional Signal and Image Processing.* Prentice Hall, Upper Saddle River, New Jersey, 1990.

[9] J. C. Russ. *The Image Processing Handbook.* CRC Press, Boca Raton, Florida, 1995.

[10] Ichiro Masaki et al. New architecture paradigms for analog vlsi chips. *Vision Chips – Implementing Vision Algorithms With Analog VLSI Circuits*, pages 353–375, 1995.

[11] D. Naidu and R. Fisher. A comparison of algorithms for sub-pixel peak detection. Technical Report Tech Report 553, University of Edinburgh Department of Artificial Intelligence, 1991.

[12] Altera Corporation. Max 7000 programmable logic device family data sheet. http://www.altera.com, February 1998.

[13] Altera Corporation. Byteblaster parallel port download cable. http://www.altera.com, August 1996.

[14] American Data Acquisition Corporation. Adac pcm55dio manual, 1997.

[15] Intel Corporation. 82c54 chmos progammable interval timer data sheet. http://www.intel.com, October 1994.

[16] Intel Corporation. 82c55a chmos progammable peripheral interface data sheet. http://www.intel.com, October 1995.

[17] Allesandro Rubini. *Linux Device Drivers.* O'Reilly and Associates, Inc, Sebastopol, CA, 1998.

[18] Neil Matthew and Richard Stones. *Beginning Linux Programming.* WROX Press, Chicago, IL, 1996.