A Real-Time 256 × 256 Pixel Parallel Image Processing System

by

Zubair Aman Talib

Bachelor of Science in Electrical Engineering
Massachusetts Institute of Technology, June 1997

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

October 1998

© 1998 Massachusetts Institute of Technology. All rights reserved.

Signature of Author

Zubair Talib

Department of Electrical Engineering and Computer Science
19 October 1998

Certified by

Charles G. Sodini, Ph.D.
Professor of Electrical Engineering
Thesis Supervisor

Accepted by

Arthur C. Smith, Ph.D.
Professor of Electrical Engineering
Graduate Officer
A Real-Time 256 × 256 Pixel Parallel Image Processing System

by

Zubair Aman Talib

Submitted to the Department of Electrical Engineering and Computer Science on 19 October 1998 in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis explores how a Pixel-Parallel Image Processor (PPIP) chip serves as the basis for a real-time low-level image processing system as used in a machine vision based intelligent vehicle control application. Utilizing a processor-per-pixel scheme, the PPIP integrates 64 × 64 Processing Elements (PEs) on a single chip. Multiple chips can be arrayed to process larger pixel images.

Previous work had been done to test and demonstrate the PPIP chip. A data-path and controller board are used in conjunction with a 2 × 2 array (four-chip) PPIP test board to process 128 × 128 pixel images.

A revision of the PPIP chip has been tested and characterized. A compact printed-circuit board design utilizes a 4 × 4 array of 16 PPIP chips to process a 256 × 256 pixel image. Logic was designed to govern data transfer to and from the chips and govern communication with the existing data-path and controller hardware. Small in size and requiring no test equipment, the PE Array board is suitable for demonstrating an intelligent vehicle control system. While supporting the existing test and demonstration system, the PE Array board is flexible enough to be incorporated into future systems. Although legacy communication protocols limit the data-path to one input image, future designs, for example, will be able to utilize multiple input images.

Thesis Supervisor: Charles G. Sodini
Title: Professor of Electrical Engineering
Acknowledgments

I'm very grateful to the number of people who have contributed to the successful completion of my master's project.

First, I'd like to extend my gratitude to Professor Charles Sodini for providing me with this research opportunity. Throughout the project he exhibited trust in my judgment and confidence in my ability. I also thank Dr. Ichiro Masaki for serving as my supervisor while Prof. Sodini was on leave of absence.

Without the assistance of Keith Fife and Jeffrey Gealow I most certainly would not have been able to complete the project. Keith and I enjoyed numerous technical discussions over Domino's pizzas. He offered good advice on design and technical matters and provided much needed assistance when it came down to crunch time. His services included, but were not limited to: thesis reading, board soldering, and chip removal. Jeff, whose integrated circuit this project is centered around, responded to numerous email questions — most long after he had graduated. His assistance during the initial debugging stages is especially appreciated.

I thank all of the students and staff in Prof. Sodini's research group. Past students Gary Hall, Nicole Love, and Steven Decker assisted in the infamous stereo vision demonstration for the NSF review. Pablo Acosta-Serafini and Don Hitko often provided support for our workstations and answered my many UNIX questions. Dan McMahl and Jim MacArthur offered assistance and advice with design and printed-circuit board layout. Many thanks are in order to Patricia Varley and Anne Hunter for their generous assistance in dealing with MIT administrative tasks.

Finally, I thank my family. In addition to all her support, and despite her busy schedule, my sister kindly proof-read my thesis. Most of all I thank my Mom and Dad. It was they who encouraged me to apply and attend MIT initially. Many years later they encouraged me to leave our struggling business at home to come back to MIT to finish my master's degree. I'm very grateful for their constant support, encouragement, and prayers. Throughout the years they've been the heroes and role models in my life. I hope that I can continue to learn from their generosity, strength of character, and faith.
# Contents

1 Introduction ........................................ 15
   1.1 Image Processing Computation ..................... 16
      1.1.1 Neighborhood Operations ..................... 16
      1.1.2 Global Operations ............................ 17
      1.1.3 Transformations ............................... 17
   1.2 Definition of Application .......................... 17
   1.3 Pixel Parallel Image Processing .................... 18
   1.4 Project Description ............................... 18
   1.5 Thesis Organization ................................ 19

2 Architecture ........................................... 21
   2.1 Overall System Architecture ....................... 21
      2.1.1 Image Acquisition ............................. 22
      2.1.2 Processing Element Array ..................... 22
      2.1.3 Processing Element Controller ................. 23
      2.1.4 High-Level Processor .......................... 23
      2.1.5 System Controller ............................. 23
   2.2 Previous Work ..................................... 24
      2.2.1 PE Array Test Board ........................... 24
      2.2.2 Format Converter .............................. 25
      2.2.3 High Speed Controller ......................... 25
      2.2.4 Test and Demonstration Setup ................. 25
      2.2.5 Software System ................................ 26
   2.3 New PE Array Board .................................. 28
      2.3.1 Increase Array Size to 256 × 256 .............. 28
      2.3.2 Reduce Board Size ............................. 28
      2.3.3 Portability .................................... 28
      2.3.4 Robustness .................................... 30
      2.3.5 Flexibility .................................... 30

3 Architectural Alternatives .............................. 31
   3.1 Comparison Methodology ............................ 32
      3.1.1 Comparing Low-Level Hardware ................. 32
      3.1.2 Comparing Image Processing Hardware by Algorithm 33
3.1.3 Comparing Image Processing Hardware by Application .................. 33
3.1.4 Comparing Image Processing Hardware by High-Level Tasks .......... 33
3.1.5 Comparing the Entire System ............................................. 34
3.1.6 Qualitative Comparison ..................................................... 34
3.2 Application Specific Hardware .............................................. 35
3.3 Digital Signal Processor ...................................................... 36
3.4 Field Programmable Gate Array .............................................. 37
3.5 Other SIMD Architectures ..................................................... 38
  3.5.1 GOLD System ............................................................. 38
  3.5.2 IMAP-VISION Chip ......................................................... 39
3.6 Summary ................................................................. 39

4 Design ................................................................. 41
  4.1 Processing Element Instruction .............................................. 43
    4.1.1 Operation Code ......................................................... 43
    4.1.2 Address Bits ............................................................ 43
    4.1.3 Function Generator ..................................................... 43
    4.1.4 Interconnect Function ................................................ 45
    4.1.5 Latch Load ............................................................... 45
    4.1.6 Bit-serial Arithmetic Example ....................................... 45
  4.2 Image I/O ............................................................... 49
    4.2.1 Transferring Data ..................................................... 52
    4.2.2 Exchanging Data ....................................................... 53
  4.3 Data Transmission Handshaking .............................................. 60
    4.3.1 Communication Between Boards ....................................... 60
    4.3.2 Data-Path State Machine ............................................. 61
    4.3.3 Idle State ............................................................... 61
    4.3.4 Acknowledge Frame Ready (AckFrame) State ......................... 61
    4.3.5 Exchange State ........................................................ 61
    4.3.6 Acknowledge Line Ready (AckLine) State ........................... 62
    4.3.7 Transfer State ......................................................... 62
    4.3.8 Exchange State ........................................................ 63
    4.3.9 Finish State ............................................................. 64
  4.4 Interchip Communication .................................................. 64
  4.5 Power Supplies ............................................................ 64
  4.6 Clocking ................................................................. 64

5 Test Board ............................................................ 69
  5.1 Purpose ................................................................. 69
    5.1.1 Testing the New PPIP Chips ......................................... 69
    5.1.2 Testing the Logic Integration ...................................... 70
    5.1.3 Testing Power Supplies and Clocks ................................ 70
    5.1.4 Testing for Board Level Problems ................................ 70
    5.1.5 Data-Path Flicker ..................................................... 71
  5.2 Description .............................................................. 71
C Software System .......................... 141
  C.1 Writing Applications ......................... 141
    C.1.1 Sequences .................................. 141
    C.1.2 Pre-Sequences .............................. 141
    C.1.3 Compilation ................................. 142
    C.1.4 Revision Control ............................ 143
  C.2 Testing PPIC Chips ............................ 143
  C.3 Changing the Refresh Interval ............... 144
  C.4 Software Patch for Previous Generation Test Board .... 144
  C.5 Expanding to Three Cameras .................. 144
  C.6 Removing Bad Memory Cells ................... 145
  C.7 Generating the Bit-Code for a New Controller .... 145
  C.8 Demonstrations ............................... 146
List of Figures

1.1 Vision based intelligent vehicle ........................................... 15
2.1 Machine-vision based intelligent vehicle system architecture .......... 21
2.2 Compressive vs. linear transfer characteristic .............................. 23
2.3 Pixel-parallel image processing system ..................................... 24
2.4 Test and demonstration system .............................................. 26
2.5 C++ implementation of Sobel vertical edge detection .................... 29
3.1 Image processing hardware: Flexibility vs. Performance ................. 35
4.1 Processing element .......................................................... 42
4.2 Instruction format ........................................................... 42
4.3 Boolean function generator ................................................. 44
4.4 Memory allocation for two 8-bit integers and their sum .................. 46
4.5 Breakdown of $256 \times 256$ image into 16 PPIP chips ..................... 49
4.6 Serial access memories ..................................................... 50
4.7 Data transmission format ................................................... 51
4.8 Pixel data numbering convention ......................................... 52
4.9 Serial access memory cell .................................................. 53
4.10 Transferring image data through serial access memories ................ 54
4.11 Processing element column ............................................... 55
4.12 Pre-Exchange ............................................................... 55
4.13 First three Typical Exchange Sequences .................................. 57
4.14 Last three Typical Exchange Sequences ................................... 58
4.15 Post-Exchange ............................................................. 59
4.16 Communication between Data-Path, Controller, and PE Array .......... 60
4.17 Data-Path handshaking state machine .................................... 62
4.18 Transfer State: Data-Path handshaking timing ............................ 63
4.19 Interchip communication ................................................... 65
4.20 PPIP clock timing .......................................................... 65
4.21 50% duty-cycle clock circuit .............................................. 67
4.22 Timing diagram for 50% duty-cycle circuit ................................ 68
4.23 Greater than 50% duty-cycle clock circuit ................................ 68
4.24 Timing diagram for greater than 50% duty-cycle circuit ................. 68
5.1 Test Board ................................................................. 72
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>16 PPIP Board</td>
<td>81</td>
</tr>
<tr>
<td>6.2</td>
<td>Connector Board</td>
<td>82</td>
</tr>
<tr>
<td>A.1</td>
<td>Controller Interface</td>
<td>95</td>
</tr>
<tr>
<td>A.2</td>
<td>Data Path Receiver Interface</td>
<td>96</td>
</tr>
<tr>
<td>A.3</td>
<td>Data Path Transmitter Interface</td>
<td>97</td>
</tr>
<tr>
<td>A.4</td>
<td>HDPP Devices</td>
<td>98</td>
</tr>
<tr>
<td>A.5</td>
<td>CPLD and Data Path Handshaking</td>
<td>99</td>
</tr>
<tr>
<td>A.6</td>
<td>Clock Drivers</td>
<td>100</td>
</tr>
<tr>
<td>A.7</td>
<td>Power Supplies</td>
<td>101</td>
</tr>
<tr>
<td>B.1</td>
<td>Controller Interface</td>
<td>130</td>
</tr>
<tr>
<td>B.2</td>
<td>Data Path Receiver Interface</td>
<td>131</td>
</tr>
<tr>
<td>B.3</td>
<td>Data Path Transmitter Interface</td>
<td>132</td>
</tr>
<tr>
<td>B.4</td>
<td>Connectors</td>
<td>133</td>
</tr>
<tr>
<td>B.5</td>
<td>Connectors</td>
<td>134</td>
</tr>
<tr>
<td>B.6</td>
<td>Clock Drivers</td>
<td>135</td>
</tr>
<tr>
<td>B.7</td>
<td>CPLD and Data Path Handshaking</td>
<td>136</td>
</tr>
<tr>
<td>B.8</td>
<td>HDPP Devices Northwest Quadrant</td>
<td>137</td>
</tr>
<tr>
<td>B.9</td>
<td>HDPP Devices Northeast Quadrant</td>
<td>138</td>
</tr>
<tr>
<td>B.10</td>
<td>HDPP Devices Southwest Quadrant</td>
<td>139</td>
</tr>
<tr>
<td>B.11</td>
<td>HDPP Devices Southeast Quadrant</td>
<td>140</td>
</tr>
</tbody>
</table>
List of Tables

1.1 Image pre-processing tasks .......................... 18
3.1 Comparison of image processing devices, $3 \times 3$ median filter .... 39
4.1 PPIP pin names and functions ....................... 41
4.2 Operation codes ..................................... 43
4.3 Function generator truth table ........................ 44
4.4 Instructions for sit-serial addition ................... 46
4.5 Processing element activity for bit-serial addition .......... 47
4.6 Function generator bits for 2-bit addition ............... 48
4.7 Function generator bits for 3-bit addition ............... 48
4.8 PPIP clock timing .................................... 66
5.1 PPIP Memory Test ..................................... 77
5.2 PPIP Functionality Test ............................... 78
Chapter 1

Introduction

Since we as humans use our eyes as our main sensory input for navigation and guidance, it is natural to consider machine vision as a means of navigation for an autonomous robot or an intelligent vehicle. Until recently, however, the hardware and software available to implement such a system were not feasible for commercial use. Recent progress in semiconductor technology and advances in algorithms engender the opportunity to use machine vision systems in real world applications.

Given these advancements in technology, machine vision could potentially serve as the primary means of navigation for an intelligent vehicle. Figure 1.1 depicts how a vehicle can be controlled by machine vision. The ability to translate the observed images into vehicle control signals lies in the image processing. The basic method is as follows: First, a camera mounted on the vehicle captures images. Next, image processing hardware translates the image into a real-world parameter of interest (such as distance to the nearest obstacle). Finally, the system controller sends instructions to the vehicle to adjust its speed and heading according to the task attempting to be accomplished.

The image processing task can be further broken down into a low-level pre-processing stage, a high-level image understanding stage, and a vehicle controller. The pre-processor uses raw image data to translate or transform the image into a format that the high-level processor can use. In turn, the high-level processor analyzes the pre-processed image to determine features or parameters of interest. The vehicle controller then issues control instructions to the vehicle to adjust heading and speed accordingly. Tracking and avoiding

![Diagram of Vision based intelligent vehicle.](image)

**Figure 1.1:** Vision based intelligent vehicle.
obstacles is used as an example to further describe these sub-tasks.

The image processing system essentially works as follows: The pre-processor uses an optical flow algorithm, comparing two sequential image frames, to determine the motion vector at each pixel. Finding the magnitude and direction of these vectors at each pixel has not reduced the amount of image data, but rather translated it into more useful information.

The high-level processor examines the motion vectors and reduces the amount of data to a few useful parameters. By removing noisy, extraneous data and examining groups of similar motion vectors, the high-level processor can determine the direction and rate of travel for various objects in the image. Based on this information, real-world parameters such as the time-to-collision with the nearest obstacle can be determined. These parameters are then passed to a vehicle controller which, based on the application requirements, sends control instructions to the vehicle.

When applying image processing to vehicle control, it is necessary to utilize multiple image processing applications like the obstacle avoidance example presented above. Performing multiple image processing application creates a challenge for general purpose hardware. Microprocessors and Digital Signal Processor (DSP) chips are simply not capable of efficiently performing the massive amount of computation in real-time.

The goal of this project is to design a build a machine vision system that is well suited for the image processing tasks of an intelligent vehicle application. To support the proposed machine vision system, first a description of general image processing computation is presented, followed by the specific image processing requirements for machine vision. Finally a pixel-parallel architecture is presented as an integral part of the image processing solution.

1.1 Image Processing Computation

Image processing computation can be divided into three broad classes: neighborhood operations, global operations, and transformations [1].

1.1.1 Neighborhood Operations

Local or neighborhood operations are two-dimensional filters where a small neighborhood of pixels are combined or manipulated to determine a new value at the pixel being processed. Typical neighborhood sizes are $3 \times 3$ or $5 \times 5$ square pixels. Neighborhood operations take an input image and produce an output image that is a smoothed, enhanced, or has some features highlighted. While these types of operations are computationally very intensive, the computation performed for each pixel is typically not that complex. Since the output images are calculated through the repeated application of identical operations to each pixel, neighborhood operations are particularly well suited to parallel processing.

As the names suggest, the output value at each pixel of a linear neighborhood operation is simply a linear function of the input pixel and its neighbor; and for a non-linear neighborhood operation, the output value at each pixel is a non-linear function of the input pixel and its neighbors. An example of a linear neighborhood function is a two dimensional convolution. An example of a nonlinear neighborhood operation is a median filter—where the value at each pixel is found by computing the median value of itself and its neighboring pixels.
1.1.2 Global Operations

Global operations involve all the pixels in the input image. An example of a global operation is histogram equalization. Histogram equalization computes the histogram statistics of the input image. The intensity histogram can be found by counting the number of pixels at each possible discrete intensity value (e.g. 256 for 8-bit intensity resolution). Because global operations do not typically produce an output value for each pixel, they tend to be computationally much less intensive than neighborhood operations.

An example of how a local and global operation can be used to perform a useful task is a simplified obstacle avoidance application. Using multiple cameras, a stereo vision algorithm determines the distance to each pixel. Neighborhood operations such as edge detection and correlation determine the depth map. By scanning through the entire image, the nearest object is found by finding the pixels with the smallest values. Since all the pixels are compared in this step, this is an example of a global operation.

1.1.3 Transformations

Transformations share properties common to both local and global operators. Like the neighborhood operations, transformations take an input image and produce a transformed output image (i.e. replace each pixel in the image with a new value). However, unlike the neighborhood operations, the transformations do not necessarily solely use the local neighborhood of pixels to determine the new value. The Hough transform, for example, uses the entire column of pixels. In the Discrete Fourier Transform (DFT), each output pixel value is dependent on all the other pixels in the image. Image rotation is an example of a transformation where the distance between the source pixel and the destination pixel are different for every pixel.

1.2 Definition of Application

In order to determine which machine vision hardware performs the most efficiently, it is first necessary to identify the image processing application and narrow down the computational requirements for that application.

Image processing is used for enhancement, restoration, compression, and for machine understanding. As diagrammed in Figure 1.1, the algorithms that are generally used for image understanding first pre-process the image and then use a high-level algorithm to interpret or extract out the features or parameters of interest. Some specific intelligent vehicle applications include adaptive cruise control, obstacle avoidance, and following traffic lane markings. The high-level aspects of these algorithms require real-world parameters such as the distance to the nearest vehicle, the rate and direction that the nearest obstacle is traveling, and the location of lane markings within an image. Table 1.1 lists some image pre-processing applications and their associated tasks.
Table 1.1
Image pre-processing tasks.

<table>
<thead>
<tr>
<th>Image Pre-Processing Task</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median Filtering</td>
<td>Noise Reduction</td>
</tr>
<tr>
<td>Smoothing and Segmentation</td>
<td>Noise Reduction</td>
</tr>
<tr>
<td>Optical Flow</td>
<td>Motion Estimation</td>
</tr>
<tr>
<td>Edge Detection</td>
<td>Lane Following</td>
</tr>
<tr>
<td></td>
<td>Stereo Vision</td>
</tr>
</tbody>
</table>

1.3 Pixel Parallel Image Processing

Having identified the image processing application, the hardware can now be chosen to exploit the computational requirements of those applications. For the applications described in Section 1.2, the pre-processing computation consists primarily of local neighborhood operations. The pre-processing stage of the algorithm is the most computationally intensive part of image processing [2].

The computational requirements of neighborhood operations are quite massive due to the large amount of pixel data. The actual computation performed for each pixel is, however, not that complex. In addition, the type of processing performed does not utilize the pipelined architectures, large data paths, and complex hierarchical memory structures of general purpose microprocessors and digital signal processors. A mismatch, therefore, exists between the computational requirements and the processing power. This mismatch motivates a simpler processor architecture that exploits the massive amount of data parallelism; the method of assigning one processor to each data element and performing all operations in parallel [3].

A processor-per-pixel scheme employing a Single Instruction stream, Multiple Data stream (SIMD) design is well suited for local neighborhood computation. New research by Gealow presents an integrated circuit design utilizing logic pitch-matched to Dynamic Random Access Memory (DRAM memory) cells to accomplish this task [4]. The Pixel-Parallel Image Processor (PPIP) chip integrates an array of $64 \times 64$ processing elements, each with 128-bits of memory, a 1-bit wide, 256-function boolean function generator to perform the computation, and the ability to communicate with north, south, east, and west neighbors. The PPIP chips can be used in parallel so that sixteen chips can together process a $256 \times 256$ pixel image.

This thesis present a pixel-parallel computing structure as a particularly efficient architecture for handling the pre-processing requirements of a machine vision based intelligent vehicle.

1.4 Project Description

The goal of this research project is to utilize 16 PPIP chips together to perform real-time image processing on a $256 \times 256$ pixel image. The project entails: (1) designing and building
1.5. THESIS ORGANIZATION

a printed-circuit board that contains 16 PPIP chips, logic and circuitry necessary to drive the chips, and (2) an interface to a data-path and controller board for data input/output and receiving control instructions, respectively. The processing element array board should (1) be small in size, (2) require no test equipment, and (3) be flexible enough to be used with the current test system while not constraining the design of future systems.

Some of the challenges of this project included:

- Testing and characterizing a new run of the PPIP chips.
- Understanding how to interface between the processing element array board to the data-path and controller boards. This involved implementing a board-level state machine that ties together the controller instructions, the protocol for transferring data, and the signals used to transfer data to and from the PPIP chip.
- Developing a flexible on-board clocking circuit, precluding the need for test equipment.
- Dealing with board level issues such as crosstalk, fanout, and signal reflections.
- Isolating the legacy design decisions from the basic operation of the processing element array board so as not to constrain future systems that will use the board.

1.5 Thesis Organization

This chapter describes the motivation for research in pixel-parallel image processing for intelligent vehicle control applications. It defines the applications for which the proposed machine vision system is intended. This chapter also introduces the pixel parallel architecture as a key component in that system.

Chapter 2 describes the system architecture for the overall application as well as the architecture of the pixel-parallel image processing system designed in this project. Chapter 3 discusses alternative image processing architectures. Chapter 4 presents the various design issues associated with using the PPIP image processing chips. Chapters 5 and 6 describe the implementation and debugging of the processing element array boards. Chapter 7 summarizes the accomplishments and offers suggestions for future work.

Appendices A and B document the processing element array boards. Appendix C describes the software system and discusses image processing application development.
Chapter 2

Architecture

This chapter describes the overall system architecture of a machine-vision based intelligent vehicle control system. The chapter presents previous research on a pixel-parallel image processing demonstration system. In addition, the chapter presents and discusses the requirements for the processing element array board.

2.1 Overall System Architecture

The general architecture shown in the system in Figure 1.1 has been refined to fit the requirements for intelligent vehicle image processing applications. The refined architecture includes three wide dynamic range imagers and a format converter, a $256 \times 256$ pixel parallel Processing Element (PE) Array, a local processing element controller, and a system controller. The refined system is shown in Figure 2.1. This section describes how the PE Array fits into the entire Machine Vision System.

The Machine Vision System consists of an image data path and a control path for the transmission of pixel data and instructions, respectively. In the image path, the following occurs: (1) three equally spaced, horizontally aligned imagers capture image data; (2) a format converter rearranges the data for optimal throughput; (3) the PE Array performs the image pre-processing (e.g. median filtering, edge detection, optical flow); (4) the high-

![Diagram of machine-vision based intelligent vehicle system architecture.](image)

**Figure 2.1:** Machine-vision based intelligent vehicle system architecture.
level processor takes the processed images and sends parameter information (i.e. distance to the nearest obstacle, location of lane marking within the image, etc.) to the system controller; (5) the system controller sends control instructions to the actual vehicle (slow down, speed up, turn left, turn right, etc.)

In the control path, the following occurs: (1) the system controller initiates the image processing algorithm (e.g. determine distance to obstacle, find lane marking); (2) the PE controller delivers low-level instructions directly to the PE Array; (3) the high-level processor is instructed to take the processed image data and return the appropriate real-world data (e.g. distance to the closest vehicle). The system controller requests real-world parameters from the high-level processor which, in turn, requests the pre-processed image from the PE Array.

2.1.1 Image Acquisition

A problem exists with conventional imagers when applied in outdoor intelligent vehicle applications. Conventional imagers are unable to provide detailed image data for both the very bright and very dark intensity regions in an image. In order for the image processing system to detect brightly lit and dimly lit objects in the same scene, the imager must have a wide dynamic range. Research by Decker presents an example of a CMOS imager with the required dynamic range [5]. The imager captures a 256 × 256 pixel array with 8 bits of pixel resolution at frame rates much faster than 30 frames per second. The innovative aspect of the imager is its non-linear light-intensity characteristic as shown in Figure 2.2. For low intensities, linear increases in charge collected correspond to linear increases in intensity. For high intensities, the charge collected corresponds to a compressed range of intensities. By compressing the high intensities, the imager is capable of capturing both dimly lit and brightly lit objects simultaneously.

Still another challenge is faced when attempting to determine three-dimensional depth from a two-dimensional images. First the distance to an object must be calculated by holding the two cameras at a fixed distance and evaluating the perceived spatial shift of an object between the two cameras. A stereo vision algorithm then calculated the distance by correlating the vertical edge maps of the two cameras. Finally, to reduce correlation errors, a third camera is placed equidistant between the two cameras.

Three cameras sending images 30 times per second equals a data transmission of nearly 50 Megabits of data per second. To ensure the data transfers quickly and efficiently into the PE Array, a format converter rearranges the image data in real-time.

2.1.2 Processing Element Array

The low-level pre-processing is performed by a Processing Element (PE) Array. The Array consists of 16 Pixel-Parallel Image Processor (PPIP) chips. Each PPIP chip, capable of processing 64 × 64 pixels images, contains a (PE) for each pixel. Each PE contains 128 bits of memory and a three-input one-bit 256 boolean function generator to perform the actual processing. PEs are able to communicate with north, south, east, and west neighbors even when connected over chip boundaries. The sixteen PPIP chips, in parallel, yield a system capable of processing 256 × 256 pixel images [4].
2.1. OVERALL SYSTEM ARCHITECTURE

![Graph showing compressive vs. linear transfer characteristic.](image)

**Figure 2.2:** Compressive vs. linear transfer characteristic.

While the PPIP chip operates at a clock speed less than 25MHz, it is capable of performing nearly 3 billion 8-bit additions per second per PPIP chip. Since control instructions are decoded once for the entire array, power consumption is quite low. Typical power dissipation is 300mW per chip.

### 2.1.3 Processing Element Controller

The PE controller is responsible for broadcasting the low-level instructions to the entire PE Array. Sequences of low-level instructions are stored in Static Random Access Memory (SRAM) on the PE controller board at compile-time. The system controller sends instructions to the PE controller to change sequence parameters or to execute a different application.

### 2.1.4 High-Level Processor

The high-level processor takes the pre-processed image and extracts out meaningful real-world parameters. For example, when running an application to determine the distance to the nearest vehicle, the PE array performs the stereo vision algorithm to determine the depth at each pixel. The high-level processor searches through this depth map to determine the location of the nearest objects. The distance to these objects can be calculated based on the known separation between the 3 cameras.

### 2.1.5 System Controller

The system controller receives real-world data from the high-level processor, such as the time-to-collision with the nearest object, or distance to the nearest vehicle. With this
information the system controller determines the speed and direction that the vehicle needs to travel. Based on the differential between actual parameters and the desired parameters, control instructions are sent to the vehicle to actuate the throttle, brake, and steering mechanisms.

The implementation details of the high-level processor and the system controller have not yet been realized.

2.2 Previous Work

Hardware and software have been developed to test the pixel-parallel image processing chips. A pixel-parallel image processing system is shown in Figure 2.3. In the image data path: the imager captures the image; the ADC converts the analog signal to digital; the front format converter rearranges the data for optimal throughput; the processing element (PE) array performs the image processing; the back format converter rearranges the data back to bit stream, which could then be converted to standard NTSC video. In the control path: the host computer sends a sequence of instructions at compile time to the controller board which, in turn, sends the instructions to the PE Array in real-time [6].

2.2.1 PE Array Test Board

The PE Array test board, consisting of four PPIP chips, is capable of processing a $128 \times 128$ pixel image. With external connectors for power and signal generators, the test board enables the PPIP chips to be tested and characterized with various power levels and clocking schemes.
2.2.2 Format Converter

In order to quickly transfer an image and efficiently utilize the array, data must be rearranged in real-time, and passed to the PE array in parallel. The format converter design incorporates SRAM and Field Programmable Gate Arrays (FPGAs) to perform the format conversion. The format converter hardware is capable of storing $256 \times 256$ pixel images into a memory buffer and then transferring the data to the array during the vertical blanking period of the imager. The format converter, referred to as the Data-Path board, takes NTSC video as input and converts it to digital before reformatting and sending it to the PE Array. The processed digital data is reformatted to bit-stream and converted back to analog (NTSC) [7].

2.2.3 High Speed Controller

In order to obtain maximum performance from the PE Array, it is necessary to use a high speed controller board to provide the instructions to the PE Array. Sending array instructions directly from the host computer would require that the instructions be generated at run time, as well as that the bus interface be able to support a high delivery rate of low-level instructions. Since the same low-level operations are repeated for each image in most image processing tasks, run-time generation of instructions is not necessary. In order to increase the array utilization, the host computer generates sequences of array instruction at compile time and stores them on the controller board. The host computer then calls a sequence of these array instructions, and the controller, in turn, sends the low-level instructions to the PE Array [3].

2.2.4 Test and Demonstration Setup

The actual test setup used to demonstrate the image processing system is diagrammed in Figure 2.4. The setup consists of a Sun SPARCstation IPX as the host computer, a CCD imager, a standard NTSC display, and a VMEbus chassis containing the format converter, controller, and a PE Array test board.

First, an application is written and compiled on the host computer. The compiled sequence of array instructions is then sent to the controller via the VMEbus. In test mode, the host computer send input images to the format converter board, which reformats and delivers the digital data to the PE Array. The controller sends the processing instructions to the PE Array, and the format converter takes the processed images from the PE Array and sends them back to the host computer for comparison and analysis.

The system can also be run in a real-time demonstration mode. In this mode, the format converter takes analog NTSC data from the CCD imager, converts the signal to digital, reformats the digital data, and delivers the input images to the PE Array. The controller sends the processing instructions to the PE Array. The format converter takes the processed images from the PE Array, converts them back to NTSC analog and sends them to the display in real-time.
2.2.5 Software System

In order to make the hardware described practical and useful, a C++ software framework was developed. The system provides a means for application programmers to write and develop high-level programs that can be tested and debugged in a software simulation environment and then compiled to run in the hardware implementation. The actual implementation of simple parallel arithmetic using the primitive instruction set of the one bit processing elements can be quite tedious and cumbersome. In order to preclude the programmer from needing to understand the actual hardware implementation of the system, a higher level C++ based instruction set was developed. The application programmer develops an application by constructing sequences of instructions. Code generators exist for many useful operations in the array such as adding, multiplying, comparing, etc. Each of the instructions executed is performed on every pixel in parallel. Parallel variables can be allocated into the 128 bits of memory. There is also support for conditional code generators, providing the ability to perform operations only on certain subsets of the array. For example, in order to threshold all pixels whose values lie between 0 and 20 down to 0, the command inside identifies the pixels and then a conditional write assigns their value to 0.

In addition to having access to a library of code generators and conditional operations, the programmer is capable of implementing their own bit-serial implemented code generator for additional flexibility. These user defined code generators, known as pre-sequences, enable the programmer to implement conditional branches, jumps, and repeat loops.

Software had been written to test the functionality and performance of the PPIP chips as well as to demonstrate the pixel-parallel image processing system in real-time. Real-time applications include: median filtering, smoothing and segmentation, optical flow, and a simple edge detection.

An example of an application written in the format of this programmable framework
demonstrates the power of the pixel-parallel image processing system. Below is a description of the implementation of the Sobel edge detection algorithm for vertical edges. The algorithm consists of convolving the input image with the following $3 \times 3$ kernel:

$$
\begin{bmatrix}
1 & 0 & -1 \\
2 & 0 & -2 \\
1 & 0 & -1 \\
\end{bmatrix}
$$

This is a diagrammatic interpretation of the instructions used to implement the edge detection application.

\begin{align}
M & \leftarrow \text{image} \\
\text{temp} & \leftarrow M^{\text{North}} \\
\text{edge} & \leftarrow M^{\text{South}} \\
\text{temp} & \leftarrow \text{temp} + 2M \\
\text{edge} & \leftarrow \text{temp} + \text{edge} \\
\text{temp} & \leftarrow \text{edge} \\
\text{edge} & \leftarrow \text{edge}^{\text{West}} \\
\text{temp} & \leftarrow \text{temp}^{\text{East}} \\
\text{edge} & \leftarrow \text{edge} - \text{temp} \\
\text{edge} & \leftarrow |\text{edge}| \\
\text{edge} & \leftarrow \begin{cases} 255 & \text{if } |\text{edge}| > \text{threshold} \\ 0 & \text{otherwise} \end{cases}
\end{align}

The unprocessed input image is assigned to the parallel variable $M$ in Equation 2.1:

$$M(i, j) = \text{image}(i, j)$$

In Equations 2.2 and 2.3 the north and south neighbor pixels are assigned to the variable temp and edge respectively:

$$\begin{align}
\text{temp}(i, j) & = M(i, j + 1) \\
\text{edge}(i, j) & = M(i, j - 1)
\end{align}$$

Equations 2.4 and 2.5 perform the multiplication and additions associated with the left column of the $3 \times 3$ kernel:

\begin{align}
\text{temp}(i, j) & = \text{temp}(i, j) + 2 \cdot M(i, j) \\
& = M(i, j + 1) + 2 \cdot M(i, j) \\
\text{edge}(i, j) & = \text{temp}(i, j) + \text{edge}(i, j) \\
& = M(i, j + 1) + 2 \cdot M(i, j) + M(i, j - 1)
\end{align}
Equations 2.6 - 2.9 complete the calculation of the convolution:

\[
\text{temp}(i,j) = \text{edge}(i,j) \\
= M(i, j + 1) + 2 \cdot M(i, j) + M(i, j - 1) \\
\text{edge}(i,j) = \text{edge}(i - 1,j) \\
= M(i - 1, j + 1) + 2 \cdot M(i - 1, j) + M(i - 1, j - 1) \\
\text{temp}(i,j) = \text{temp}(i + 1,j) \\
= M(i + 1, j + 1) + 2 \cdot M(i + 1, j) + M(i + 1, j - 1) \\
\text{edge}(i,j) = \text{edge}(i,j) - \text{temp}(i,j) \\
= M(i - 1, j + 1) + 2 \cdot M(i - 1, j) + M(i - 1, j - 1) \\
- M(i + 1, j + 1) - 2 \cdot M(i + 1, j) - M(i + 1, j - 1)
\]

Equation 2.10 calculates the absolute value of the convolution sum and Equation 2.11 compares it to a threshold. White pixels are shown when the convolution sum is greater than the threshold (indicating an edge) and black pixels are shown otherwise.

The C++ instructions used to implement this algorithm are shown in Figure 2.5. Details of the syntax and code generators are discussed in Appendix C.

### 2.3 New PE Array Board

The PE Array Board designed by Gealow [4] was intended to test the performance and functionality of the PPIP chips. The goal of this Masters project is to use the PPIP chips to build a fully pixel-parallel image processing system that can be utilized in an actual intelligent vehicle control application.

#### 2.3.1 Increase Array Size to 256 × 256

The first requirement of the project is to increase the array processing size from 128 × 128 pixels to 256 × 256 pixels, using sixteen PPIP chips instead of four. Clock distribution, crosstalk, and fanout are some of the board level issues that arise when increasing the number of chips.

#### 2.3.2 Reduce Board Size

The original test board was 9.187" × 11.024" extending several inches outside the VMEbus chassis. The actual hardware to be used in the image processing system should fit on a standard VMEbus card, 9.187" × 6.299".

#### 2.3.3 Portability

In order to test the timing and power consumption of the PPIP chips, power supplies and signal generators were used. In the final system the timing and power issues should be finalized and the new PE Array Board should run off of a single +5 volt supply (provided by the VMEbus chassis). Since the final system is intended to run an actual vehicle, no test equipment should be required.
2.3. NEW PE ARRAY BOARD

```
ufield M(aC.statically(), 8);
sfield temp(aC.statically(), 9);
sfield edge(aC.statically(), 9);

sequence seqSobel;

seqSobel += imageIO(aC,M,M);
seqSobel += nbrDifference(aC,M,North,temp);
seqSobel += nbrDifference(aC,M,South,edge);
seqSobel += add(aC,M,M);
seqSobel += add(aC,temp,M);
seqSobel += add(aC,edge,temp);
seqSobel += copy(aC,temp,edge);
seqSobel += nbrDifference(aC,edge,Central,edge);
seqSobel += nbrDifference(aC,temp,East,temp);
seqSobel += subtract(aC,edge,temp);
seqSobel += write(aC,M,Ou);
seqSobel += abs(aC,edge);
seqSobel += outside(aC,edge,0,parm.threshold);
seqSobel += writeC(aC,M,255u);
```

Figure 2.5: C++ implemenation of Sobel vertical edge detection.
2.3.4 Robustness

The original PE Array test board exhibited an occasional "flicker" during real-time demonstrations. Since loss of data or processing lapses cannot be tolerated in the machine vision system, the new PE Array Board must not have such flicker problems.

2.3.5 Flexibility

The new PE Array Board needs to work with the final intelligent vehicle system shown in Figure 2.1. Since most of the components in this system are not yet built, the new PE Array board also needs to work with the current legacy system so that its operation can be validated. Protocols and conventions that support the legacy systems must be abstracted away from the core 16-chip processor which will be used in the final system. While the original system only supported data I/O for one camera, the new board must support three. Separating the functional aspects of the PE Array from the legacy system prevents future hardware from being constrained by old technology and legacy design decisions.
Chapter 3

Architectural Alternatives

This chapter explores image processing architectures and how they behave in machine vision systems applied to intelligent vehicle control tasks.

The image processing architecture proposed in this research consists of an SIMD pixel-parallel processor and a general purpose high-level processor. This architectural decision makes two assumptions:

1. Machine vision image processing tasks can be divided into two computationally different stages; namely a low-level pre-processing stage and a high-level image interpretation stage.

2. The low-level pre-processing requirements consist primarily of local neighborhood operations.

The first assumption is based on the notion that raw image data must be transformed into a format understandable to a machine. Examples of such formats include edge maps and motion vectors. Without having extensive knowledge of the images, some form of pre-processing is necessary. Even with pre-processing, imperfection in and complexities of images require noise reduction. Examples of noise reduction include median filtering and smoothing and segmentation algorithms. Pre-processing computation is typically the most intensive aspect of image processing because it computes a new value for each pixel. From the pre-processed image, higher level processing determines real-world parameters of interest. This first assumption is well-grounded and considered true for a large number of image understanding applications.

While the first assumption is generally true, the second assumption is not. That is, low-level pre-processing requirements do not necessarily consist primarily of local neighborhood operations. Instead pre-processing algorithms that use geometric estimations and other model based parameters often require more complex, global operations [2, 8]. Algorithms often use geometric models because neighborhood operations are often inadequate; they are too computationally intensive for general purpose hardware to perform in real-time.

The architecture presented in this thesis is based on the assumptions about the computational requirements for image processing applications described above. Given these assumptions, the pixel-parallel processor architecture is very efficient in handling the low-
level image processing. This claim of performance efficiency is supported by analysis done by Gealow. Chapter 5 of Gealow's thesis [4] compares the pixel parallel processor chip with other integrated circuits. In comparison with microprocessors, DSP chips, and FPGA circuits, the PPIP chip was shown to be very efficient in terms of both processing speed per unit area silicon and energy per pixel for local neighborhood processing tasks. While the pixel-parallel architecture is particularly well suited for local neighborhood operations, several alternative pre-processor architectures offer various other benefits and trade-offs.

This chapter explores some of the issues associated with using the pixel-parallel image processing system in a machine vision system for intelligent vehicle control applications. First, the methods for comparing different architectures are presented. Next a qualitative comparison is presented; a comprehensive comparative analysis is beyond the scope of this research. Image processing hardware alternatives are presented, highlighting the advantages and disadvantages of each architecture. Finally, a brief section summarizes the architectural decision and comparison analysis.

3.1 Comparison Methodology

Comprehensively comparing image processing hardware for intelligent vehicle applications is a deceptively large task. System dynamics suggest that the best individual components do not necessarily make the best overall system; it is not sufficient to evaluate a component of a system without evaluating how the component behaves in the actual system. The interaction between the individual components is vital to the operation of the overall system. Therefore, determining the best low-level image processing hardware for use in intelligent vehicle applications is a complex task.

This section describes how a comprehensive comparison might be conducted to evaluate low-level image processing hardware within a machine-vision based image processing system. In Sections 3.1.1 - 3.1.5, comparison methodologies are presented in order of increasing complexity. The following sections describe (1) methods of comparison, (2) metrics used to make those comparisons, and (3) shortcomings of the comparisons. Finally, Section 3.1.6 describes the qualitative comparison technique used in the remainder of this chapter.

3.1.1 Comparing Low-Level Hardware

The first way to make the comparison between the PPIP and other image processing architecture is within low-level processing. This comparison has already been made with respect to neighborhood operations [4]. The metrics used as the basis of comparison were processing speed (normalized for chip area) and power consumption.

From a system level, there are several problems with only comparing the low-level aspect of the image processing. One such problem is that the comparison gives no gauge of how the processor will behave in the overall system. For example, the PPIP performs certain applications faster than other chips, but transfers data to the higher-level processor much slower. In an architecture where the same memory is used to perform low and high level processing, there is no data transfer time. Another example is that processing time latency could be the same for two architectures, but processed data throughput might be faster for one. Without knowing the requirements of the high-level processor there is no way to fairly
3.1. COMPARISON METHODOLOGY

weigh the advantages of faster throughput. Processing time comparisons are meaningless without considering the other performance limiting behaviors of the PPIP within the system.

A second problem is how to compare different architectures when some run certain applications faster than others, and vice versa. Without knowing the importance or robustness of the various algorithms in the final system implementation, there is no fair way to assign a weighting to these metrics and therefore no way to make a normalized comparison.

3.1.2 Comparing Image Processing Hardware by Algorithm

A better, although more involved, method for performing comparisons exists. The method is to compare architectures at the image processing system level. This comparison would evaluate the combination of the PPIP and some high-level processor against other complete image processing systems (e.g. array of DSPs, array of FPGAs and a DSP, ASIC and a microprocessor). The same set of algorithms should be performed on each architecture to standardize the metric for performance. The basis for comparison is essentially the same as above: processing time normalized for chip area. The processing time and chip area should be calculated for both the low-level and high-level processor together.

This method is better than the previous comparison technique, because it starts to encompass system level issues such as data transfer between the low and high level processors. One problem with this method is that there is no way to standardize performance because some architecture might perform some algorithms better than others. Different algorithms might run better on different architectures, but they might also accomplish the same image processing application (e.g. finding obstacles using stereo vision vs. using optical flow).

3.1.3 Comparing Image Processing Hardware by Application

A third comparison attempts to remove algorithmic and architectural dependencies. This comparison method compares image processing architectures by the application they accomplish. The algorithm is chosen optimally for each architecture. This permits each architecture to perform the types of computation that it performs best. The metrics used for comparison would still be performance efficiency; determining which architecture performed the application most efficiently with the minimum processing time.

One problem with this comparison is that it introduces another variable: like algorithms are not being compared across hardware, so the performance of the final system varies from platform to platform.

A second problem is that while individual applications might run well on one architecture, it is not clear that the system will be able to handle multiple applications. FPGAs, for example, might perform individual applications very quickly. However, FPGAs might not be able to perform a series of different applications very quickly since it takes time in between applications to reconfigure the hardware. The PPIP, for example, might not have the required memory to perform multiple applications.

3.1.4 Comparing Image Processing Hardware by High-Level Tasks

In order to evaluate the architecture, comparisons could be made by using a high-level task such as intelligent cruise control. Intelligent cruise control could be accomplished by using a
series of image processing tasks: avoiding obstacles, maintaining a fixed distance to the car in front, and lane following. Each individual application would be specifically implemented to the architecture.

The metrics used to evaluate this comparison could still be performance efficiency; that is, what is the system cost used to perform the application in a certain amount of time. System cost could perhaps be roughly equated with the aggregate chip area of the hardware in the system.

The problem with this comparison is that different implementations of the high-level task might not perform equivalently in the same amount of processing time. There is no easy way to normalize the processing time for the efficiency and robustness of the algorithm. A secondary issue is that no weight is given to ease of development and implementation.

3.1.5 Comparing the Entire System

The most significant test of the image processing hardware is how it behaves in an actual system. This system consists of cameras, image processing hardware, a system controller, a controllable vehicle, and image processing algorithms. To determine the overall system performance, it is not enough to gauge robustness of each individual image processing application. All the applications must be integrated by the system controller, which determines the actual speed and heading of the vehicle. A comprehensive test could be made, perhaps, by simulating the entire system. Implementing the real-system is probably less complex and more accurate than performing simulations.

The metrics that could be used to perform this comparison would be broad in nature: robustness, cost, flexibility, and ease of development.

Several problems surface with this sort of comparison. The comparisons are complex and involved to perform. Each system has its own set of advantages and disadvantages. Depending on the requirement of the application each would have to be weighed accordingly. Furthermore, it is very difficult to evaluate the performance of just the image processing hardware due to the large number of variables in the system.

3.1.6 Qualitative Comparison

The comprehensive comparison described above is beyond the scope of this research. Alternatives to the PP1P architecture are presented in the remainder of this chapter. The primary criteria used for identifying the appropriate image processing hardware are flexibility and performance. As shown in Figure 3.1, there is a trade-off in image processing hardware between flexibility and performance. Systems based on microprocessors or DSP chips are extremely flexible and useful for exploring a variety of different and complex algorithms. As a result of their general purpose hardware, however, they tend to be the slowest performing systems. Conversely, Application Specific Integrated Circuit (ASIC) based systems are fast and efficient, but inflexible because they are designed specifically for a particular image processing task. It is not possible to have general purpose hardware that is as performance efficient as application specific hardware. The goal of this research is to choose or design an architecture that is fast enough to perform real-time image processing applications, yet flexible enough to be programmed for a specific class of image processing needs, enabling
3.2. APPLICATION SPECIFIC HARDWARE

Figure 3.1: Image processing hardware: Flexibility vs. Performance.

the quick and easy development of algorithms as well as the ability to run multiple applications on the same hardware. The applications of interest were defined in Section 1.2. The comparisons in the remainder of this chapter will explore the various system trade-offs.

3.2 Application Specific Hardware

Instead of designing hardware particularly well suited to a certain class of image processing tasks, dedicated hardware could be designed specifically for each pre-processing application. A real-time vision system designed by Toyota demonstrates this technique by incorporating edge segment extraction, optical flow, stereo vision, and template matching modules [9].

While this architecture is very efficient for the tasks it was designed for, it is also very inflexible. It is not clear that this hardware can handle a wide enough class of intelligent vehicle applications to make the architecture useful for commercial applications. The implicit assumption in using this particular hardware is that all the pre-processing tasks required to implement the high level applications makes use of edge segments, motion vectors, or depth maps.

A serious advantage of general purpose hardware over application specific hardware, including the PPiP chip, is a proven market demand and the associated economies of scale. Utilizing an ASIC-based system in a commercial intelligent vehicle application bears the risk of obsolescence. That is, if the demand is not great enough, hardware advances will not be able to keep pace with advancements in technology. The comparative analysis in Gealow's thesis [4], shows the PPiP has a higher performance per unit silicon area metric than even the fastest DSP chips. However, since DSP chips are in such large demand for a variety of applications, continual technological advances are taking place. Advances in process technology, architecture, and circuit design all lead to increasingly faster performing chips. From an academic standpoint, the application specific hardware might have a more efficient architecture, but on an absolute scale general purpose hardware might perform better.
3.3 Digital Signal Processor

While ASIC-based designs are limited to a particular application, a digital signal processor can be used for a wide class of image and signal processing tasks. Many machine vision algorithms are developed and tested with DSP chips since they are so flexible and application development is quite easy.

The drawback to this flexibility is performance: due to their general purpose architectures, DSPs are not very efficient at performing neighborhood operations. Texas Instrument's TMS320C80 (C80) DSP is designed for image processing applications and is capable of performing over 2 billion operations per second [10]. In spite of its impressive speed, the C80 is 5 to 10 times less efficient in terms of performance per unit silicon area for tasks such as 5 × 5 convolution and 3 × 3 median filtering than the PPIP chip [4].

A significant advantage of using a DSP, however, is that low-level and high-level operations are performed in the same architecture; low and high level operations can be intermixed and image data need only reside in one physical memory. Since the C80 is not capable of performing all the low-level image processing tasks in real-time, several DSP chips could be used together in parallel.

Some DSP chips are designed specifically to be used in parallel. An example of such a DSP is Texas Instrument's TMS320C40 (C40). The C40 is a 32-bit floating point DSP with a wide bandwidth external data interface that enables the C40 to be used in parallel with other C40 chips. In one application an array of eight C40 DSP chips is used to perform real-time stereo vision and lane finding [2]. In the lane finding algorithm, the Hough transform is used to compare candidate lane estimators with observed data. The Hough transform is used in this algorithm since other known robust techniques, like geometric modeling or deformable template matching, are computationally too intensive [8]. Neither the Hough transform nor the deformable template techniques are well suited for the pixel parallel architecture. In both cases the processing primarily consists of global operations.

Another application that makes use of an array of C40 DSP chips is real-time depth mapping system designed by CMU [11]. A multi-baseline stereo vision algorithm is used to produce depth maps at video-rate. Utilizing multiple cameras, the algorithm calculates the image disparity between cameras. The depth at each pixel can be calculated from the focal length of the cameras, the length of baseline, and the disparity. The algorithm involves: (1) Laplacian and Gaussian (LOG) filtering, (2) computing the Sum of Squared Differences (SSD) between stereo pairs, (3) summing all the SSD values (SSSD), (4) identifying the localization of the minimum SSSD, and (5) performing sub-pixel localization. Each camera has its own dedicated application specific hardware to perform the LOG and SSD processing. The output for each camera serves as the input to another piece of custom hardware that accumulates the SSD from each camera and determines the minimum difference. An array of C40 DSP chips are used to perform the sub-pixel interpolation.

DSP chips and application specific hardware are used in a number of applications to perform image processing tasks in real-time. The pixel-parallel processor, however, is well suited for performing much of this computation. The PPIP performs the LOG filter processing, for example, quite fast: the 5 × 5 Laplacian filter takes 197 μs and 5 × 5 Gaussian (convolution) takes 730 μs.

While the processing speed for any individual aspect of the CMU system is quite fast and
very efficient on the PPIP, it does not appear that the PPIP is capable of handling all the computational requirements for the CMU multi-camera stereo vision algorithm. The major problem is the amount of memory available for each processing element. Performing the SSD alone requires that 30 disparity calculations need to be tallied to find the minimum and perform the sub-pixel interpolation. With 128 bits of memory per PE, there is only enough memory for sixteen 8-bit words. To tally the 30 disparity calculations, an additional 192 bits of PE memory would be necessary. Unlike the PPIP, the CMU system, processes the images in a pipelined, serialized fashion. A small region of each camera goes through its respective LOG processor, then SSD processor. The results for each camera are accumulated and the minimum disparity is found. Sub-pixel localization is then performed on that region. Because data is processed in a serial fashion, the CMU system does not require the massive amount of memory as the PPIP.

As discussed above in Section 3.1.2, it is not a fair comparison to attempt to run the CMU algorithm on the PPIP architecture. The CMU system performs general purpose stereo vision at the expense of a lot of dedicated hardware. For a number of intelligent vehicle applications it is not necessary to determine the depth at each pixel in the image with 8 bits of resolution. By modifying the algorithm to preclude the need to tally the 30 disparity calculations, the PPIP system is capable of performing the CMU stereo vision algorithm and determining depth to 4–6 bits of resolution with 128 bits of memory per PE. However, a more elegant three-camera stereo vision algorithm used to determine the distance to cars has much less computational requirements. The application operates in 4.00 ms on the PPIP [6]. The distance to vehicles are accurately found by correlating vertical edges maps and performing sub-pixel approximation. This algorithm runs efficiently on the PPIP processor and accomplishes the same effective task for an intelligent vehicle application.

In summary, the PPIP provides a good alternative to DSPs and dedicated hardware for low-level processing in a number of applications. DSPs, however, are capable of performing a much wider class of image processing operations and are therefore useful for more complex algorithms.

### 3.4 Field Programmable Gate Array

Unlike the DSP, for a specific low-level application, a Field Programmable Gate Array (FPGA) based design can be extremely efficient. A single Atmel FPGA can perform a Sobel edge detection on a 64 x 64 pixel image in 47.5 μs [12]. The same operation requires 175 μs on a TMS320C80 DSP, and 72 μs on the PPIP chip [4, 13]. (Note: although the Atmel FPGA performs faster than the PPIP chip, the comparison is not normalized for chip area or process technology.)

A custom computing platform, Splash-2, based on 17 Xilinx XC4010 FPGAs performs a variety of image processing tasks in real-time [1]. The entire 17 FPGA Splash-2 system has comparable performance to a single PPIP chip. Splash-2 performs a 3 x 3 median filter on a 512 x 512 image in 27 ms, while the PPIP chip can perform the median filtering on a 64 x 64 pixel image in 105 μs. Multiple PPIP chips can be used in parallel to process larger pixel images with the same amount of processing time. While FPGAs are comparable to the pixel-parallel processor in terms of performance, two major drawbacks of using FPGAs
are (1) the design effort required and (2) the processing time spent reconfiguring the system to run a different application. For application development, gate level schematic entry or hardware description languages such as Verilog or Very High Speed ASIC Hardware Description Language (VHDL) must be used for FPGAs. In comparison, a C++ application development framework is available for the pixel-parallel processor. Furthermore, a considerable drawback of FPGAs versus the PPIP, is that FPGAs can only run one algorithm at a time. In a system running multiple applications, the PPIP system is more desirable than an FPGA based system. Since the PPIP system is programmable, no processing time is lost during reconfiguration. Reconfiguration time for Splash-2 is a few seconds, but for some single FPGAs reconfiguration is as low as a few milliseconds [1]. In the future, the impact of reconfiguration may be minimal. Some recent Atmel FPGAs have already offered the ability to partially reconfigure the FPGA at the rate of a few microseconds per cell [14].

Computation flexibility is another tradeoff between the PPIP and FPGAs. FPGAs are capable of performing column-wise or global operations, such as the Hough transform, which the PPIP is not. It is not clear, however, if FPGAs are capable of performing more complex neighborhood algorithms, like optical flow, as efficiently as the PPIP.

3.5 Other SIMD Architectures

Another alternative to the pixel-parallel structure is a less parallel SIMD architecture. While the pixel-parallel architecture is very efficient for local neighborhood operations, it is not good for pre-processing that involves global operations.

3.5.1 GOLD System

The Università di Parma's GOLD system utilizes a custom built pre-processor followed by a SPARC based high-level processor to perform lane and obstacle detection in real-time [15]. The innovative aspect of the GOLD system is its ability to detect traffic lane markings through image remapping. The Inverse Perspective Remapping is a global transformation that removes the perspective effect from an image. Lane markers have a constant width within the remapped image, and are thus easier to detect by the high-level processor. The custom designed low-level pre-processor consists of a parallel SIMD processor array and fast image remapping hardware. The processor array consists of 256 processors implemented in 16 custom ASICs, and the image remapping hardware is implemented in an FPGA.

The advantage of the GOLD system is that the pre-processor utilizes a parallel architecture for neighborhood operations and dedicated hardware for a specific computationally intensive global pre-processor operation.

One disadvantage is that the hardware was designed with the intention to implement the image remapping based algorithms. It is not clear if other algorithms, such as optical flow or stereo vision, can be performed on the GOLD system in real-time. The architecture is also not as parallel as the PPIP chip; the GOLD system has 16 PEs per chip whereas the PPIP chip has 4096 PEs per chip.
Table 3.1
Comparison of Image Processing Devices, 3 × 3 median filter.

<table>
<thead>
<tr>
<th>Device</th>
<th>Area per pixel</th>
<th>Time per frame</th>
<th>Processing Time-Area Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixel-Parallel Image Processor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096 One-bit-wide PEs One pixel per PE</td>
<td>19 222 μm²</td>
<td>105 μs</td>
<td>2.02 μs · m²</td>
</tr>
<tr>
<td>NEC IMAP-VISION [16]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256 8-bit processors, 40 MHz 256 × 240 image</td>
<td>28 520 μm²</td>
<td>1.07 ms</td>
<td>30.5 μs · m²</td>
</tr>
</tbody>
</table>

3.5.2 IMAP-VISION Chip

NEC corporation developed an integrated memory array processor for real-time image pre-processing tasks [16]. The IMAP-VISION chip has 32 processing elements, each with 8-bit wide ALU and 1 kilobyte of local memory. A system utilizing eight IMAP chips, or 256 processing elements, has 10.24 GIPS peak performance. Processing a 256 × 240 pixel image, the IMAP system performs 3 × 3 median filtering in 1.07 ms. As shown in Table 3.1, the PPIP chip has a performance per unit silicon area metric 15 times better than the IMAP chip.

Although less performance efficient for local neighborhood operations than the PPIP chip, the IMAP chip is capable of performing a wider class of image processing tasks. Since the processing elements are capable of accessing the entire image's pixel information directly from memory, global operations such as intensity histogram equalization and the hough transform can be performed.

3.6 Summary

The PPIP is a fast, efficient processor that holds great potential for use with image understanding and intelligent vehicle applications. The strength of the PPIP lies in its ability to efficiently perform extremely intensive low-level local neighborhood operations. Employing a flexible, programmable architecture, developing applications with the PPIP proves to be relatively easy when compared to FPGAs and application specific hardware.

However, the PPIP is not without its shortcomings. It is, for example, limited to a certain class of image processing computation. Because of its massively parallel architecture, the PPIP is not capable of efficiently performing complex algorithms that require serial or global operations. Moreover, because the processing must be complete before data is transferred, the application must execute entirely within the on-chip memory. With only 128-bits of memory per processing element the PPIP is limited to the number of input images it can process in real-time. The PPIP requires all the data to be transferred before
it can begin processing. This results in the data-path limiting the PPIP from utilizing external memory effectively. Currently, 16 chips are required to process a $256 \times 256$ pixel array. Halving the number of PPIP chips more than doubles the effective processing time because output data is only available after all the processing is complete. In terms of throughput, pipelined serial processors scale better. In terms of data communication over chip boundaries and processing larger pixel images, however, the PPIP chip scales better.

Although the PPIP architecture is limited to certain type of computation, a PPIP based image processing system could indeed prove to be more efficient than the other architectures presented. One difficulty in performing such a comparison is that equivalent application development effort has not taken place for the PPIP. Currently, the majority of image understanding algorithms have been developed for implementation on a serial processor. Real-world knowledge and models can be exploited to simplify the image processing computation for these processors. Further application development, however, could show that applications designed to run on a PPIP based system perform better than the equivalent application implemented on a different architecture.

Used in conjunction with a high-level processor, the PPIP shows great promise for use in an intelligent vehicle control system. In a number of vision based intelligent vehicle applications the low-level image processing computation is the performance bottleneck. The PPIP offers a solution to the mismatch between serial processing power and the massively parallel nature of local neighborhood operations. In order to fairly evaluate the PPIP architecture in an intelligent vehicle application, however, the entire system needs to be considered. First, to compare processing power in terms of chip area and processing time, the PPIP chip must be fabricated in the same aggressive process technologies as commercially available chips. This would enable the entire $256 \times 256$ array of PEs to be integrated onto a single chip. Second, the data-path interface must be taken into consideration: a faster more efficient method of transferring data could be devised. Third, algorithms must be developed to fully utilize the processing power of the PPIP. As has been the case with DSPs, the computational needs of the algorithms based on its performance in a system will drive improvements and evolution of the PPIP architecture.
Chapter 4

Design

The Pixel-Parallel Image Processor (PPIP) chip consists of an array of $64 \times 64$ (4096) Processing Elements. As shown in Figure 4.1, each processing element incorporates $128 \times 1$ bit of DRAM memory, a three input 256 function boolean generator, and the ability to communicate with the north, south, east, and west neighbors. The processing element communication extends over chip boundaries, so that PPIP chips can be cascaded together to form larger processing element arrays. Serial access memories are used to transfer image data into and out of the PE Array. Table 4.1 lists the pin names and functions of the PPIP chip. A full chip pin-out is included in Appendix A of Gealow's thesis [4].

This chapter discusses the implementation details of how to use the PPIP chip in an actual image processing system. Section 4.1 presents the operation of the processing element. Section 4.2 details image input and output. Section 4.3 discusses the board level logic used to communicate with the Data-Path and Controller boards. Sections 4.4 describes the interchip communication and Sections 4.5 and 4.6 describe the power supplies and clocks necessary to drive the PPIP chips.

<table>
<thead>
<tr>
<th>Function</th>
<th>Pin Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power supplies</td>
<td>$v_{ss}$, $v_{dd}$, $v_{pp}$, $v_{il}$, $v_{hh}$</td>
</tr>
<tr>
<td>Global chip enable</td>
<td>$i3$</td>
</tr>
<tr>
<td>PE instruction clocks</td>
<td>$clk0$, $clk1$</td>
</tr>
<tr>
<td>PE instruction signals</td>
<td>$i_2 - i_0$, $a_6 - a_0$, $f_r - f_0$, $f_W$, $f_E$, $f_5$, $f_N$, $l_a$, $l_b$, $l_c$, $l_d$, $l_e$</td>
</tr>
<tr>
<td>Interchip communication clocks</td>
<td>$nck0$, $nck1$</td>
</tr>
<tr>
<td>Interchip communication signals</td>
<td>$n_0 - n_{15}$, $s_0 - s_{15}$, $e_0 - e_{15}$, $w_0 - w_{15}$</td>
</tr>
<tr>
<td>Serial data transfer and control signals</td>
<td>$sck1$, $sck2$, $str$, $sld$</td>
</tr>
<tr>
<td>Serial input/output signals</td>
<td>$s_{00}$, $s_{01}$, $s_{000}$, $s_{01}$</td>
</tr>
</tbody>
</table>

Table 4.1

PPIP pin names and functions.
Figure 4.1: Processing Element. Solid lines represent data signals. Dashed lines represent control signals.

| 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| i_2 | i_1 | i_0 | a_6 | a_5 | a_4 | a_3 | a_2 | a_1 | a_0 | f_7 | f_6 | f_5 | f_4 | f_3 | f_2 | f_1 | f_0 | f_w | f_E | f_S | f_N | l_a | l_b | l_c | l_d | l_e |

Figure 4.2: Instruction Format.
4.1 Processing Element Instruction

Figure 4.2 lists the signals that form the processing element instruction. In addition to a global chip enable pin (\(i_3\)), these 27 signals control the 4096 PEs. Five types of signal comprise/compose the PE instruction word:

- Operation code \((i_2 - i_0)\)
- Address bits \((a_6 - a_0)\)
- Function generator \((f_7 - f_0)\)
- Interconnection function \((f_W, f_E, f_S, f_N)\)
- Latch load \((l_a, l_b, l_c, l_d, l_e)\)

This section describes the operation of each of these signals.

4.1.1 Operation Code

The op-code signals dictate the PE activity. As can be seen in Table 4.2, the op-code governs reads from memory and writes to memory—including DRAM refreshes—and interchip communication.

4.1.2 Address Bits

The seven address bits \(a_0 - a_6\) directly address the PE's 128 x 1 bit DRAM memory column.

4.1.3 Function Generator

The Boolean function generator is used to perform the actual computation in the PE. Conceptually, the function generator can be thought of as an 8-bit selector as shown in Figure 4.3. The three inputs \(A, B, C\) are used to select one of the eight signals \(f_0 - f_7\) as the output.

<table>
<thead>
<tr>
<th>Table 4.2 Operation Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>Zero</td>
</tr>
<tr>
<td>Array</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Refresh</td>
</tr>
<tr>
<td>North</td>
</tr>
<tr>
<td>South</td>
</tr>
<tr>
<td>East</td>
</tr>
<tr>
<td>West</td>
</tr>
</tbody>
</table>
Table 4.3

Function Generator Truth Table

<table>
<thead>
<tr>
<th>Latch C</th>
<th>Latch B</th>
<th>Latch A</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>f0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>f1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>f2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>f3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>f4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>f5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>f6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>f7</td>
</tr>
</tbody>
</table>

Table 4.3 shows the truth table that governs the operation of the function generator. For the inputs shown in the table, the result is assigned to the corresponding function bit. Some examples of typical function generator operations include: output $0 \Rightarrow \text{x00}$, output $1 \Rightarrow \text{xFF}$, output $A \text{ AND } B \text{ AND } C \Rightarrow \text{x80}$, and output the contents of Latch A $\Rightarrow \text{AA}$.

In order for the function generator to output the contents of Latch A, the output must be 0 when Latch A contains 0, and 1 when Latch A contains 1. According to Table 4.3, when Latch A contains 0, the function generator outputs $f_0$, $f_2$, $f_4$, or $f_6$. When Latch A contains 1, the function generator outputs either $f_1$, $f_3$, $f_5$, or $f_7$. Therefore, by setting $f_0 = f_2 = f_4 = f_6 = 0$ and $f_1 = f_3 = f_5 = f_7 = 1$, the function generator outputs the contents of Latch A. Similarly, to output the contents of Latch B, $f_7 - f_0 = \text{xCC}$; for Latch C, $f_7 - f_0 = \text{xFO}$. 
4.1.4 Interconnect Function

The contents of Latch A are communicated to the neighboring processing elements. The interconnection function bits are used to access this data. Specifically, the \( f_n, f_s, f_e, f_w \) signals enable access to the north, south, east, and west neighbor's Latch A contents, respectively.

4.1.5 Latch Load

The signals \( l_s, l_b, l_c, l_o, \) and \( l_e \) are used to synchronously load the Latches A, B, C, D, and E. The latches are opened after the rising edge of \( CLKI \) and closed after the falling edge of \( CLK0 \).

All memory reads must be stored in Latch A. Results of arithmetic operations can be written to memory through Latch D or can be used in subsequent arithmetic operations by utilizing Latches B or C. Latch E is used as a local write enable. When writing to memory, the op-code signals \( i_2-i_0 \) that are used to perform a memory write are broadcast to the entire processing element array. The contents of Latch E can be used to perform conditional operations based on local PE information.

For example, in a thresholding application it might be desirable to output a white pixel when the input pixel is greater than some threshold C and black when it is less. The input pixel is compared to the threshold value C. The output of the comparative arithmetic operation is stored in Latch E. If the comparison is true (i.e. the pixel value is greater than C) then the local write enable option is enabled. If the comparison is false and the output of the function generator is 0, the local write enable option is disabled and nothing is written to memory.

4.1.6 Bit-serial Arithmetic Example

A simple bit-serial arithmetic operation serves as a good example of how the PE instruction signals are used. This example shows the first few PE instructions used to perform an 8-bit addition. This example does not make use of the interconnect communication signals. An example in Table C.1 of Gealow's thesis demonstrates their use.

Two integers, \( a \) and \( b \), are stored in memory locations \( x01-x08 \) and \( x09-x10 \) as shown in Figure 4.4. The sum of \( a \) and \( b \), denoted as \( s \), will be stored in memory locations \( x11-x18 \). The 8-bit sum is performed one bit at a time. Essentially, three instructions are used to perform the sum for each bit: (1) reading input bits from memory, (2) calculating the sum and writing it to memory, (3) calculating and storing the carry bit. First, Latches A and B read in one bit of integer \( a \) and \( b \) from memory. Next, the boolean function generator uses the contents of Latches A, B, and C to calculate a 1-bit sum. The sum is stored in Latch D and then written into memory. Last, the carry bit is calculated and stored in Latch C. This operation is repeated for each of the 8 bits.

Table 4.4 shows the actual PE instructions used to perform the operation. Table 4.5 shows the functional activity of the processing element. Note that \( c0 \) is used to denote the carry bit of \( a0 + b0 \), and \( c1 \) is used to denote the carry bit of \( a1 + b1 + c0 \).
\[ a + b = s \]

| a_0 | a_1 | a_2 | a_3 | a_4 | a_5 | a_6 | a_7 | b_0 | b_1 | b_2 | b_3 | b_4 | b_5 | b_6 | b_7 | s_0 | s_1 | s_2 | s_3 | s_4 | s_5 | s_6 | s_7 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 01  | 02  | 03  | 04  | 05  | 06  | 07  | 08  | 09  | 0A  | 0B  | 0C  | 0D  | 0E  | 0F  | 10  | 11  | 12  | 13  | 14  | 15  | 16  | 17  | 18  |

**Figure 4.4:** Memory allocation for two 8-bit integers and their sum. The bottom 24 bits of the 128-bit memory structure are shown.

<table>
<thead>
<tr>
<th>i2 – i0</th>
<th>a6 – a0</th>
<th>f7 – f0</th>
<th>fn, fs, fe, fw</th>
<th>le</th>
<th>ld</th>
<th>lc</th>
<th>lb</th>
<th>la</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.</td>
<td>zero</td>
<td>x09</td>
<td>xFF</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1.</td>
<td>zero</td>
<td>x01</td>
<td>xAA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>write</td>
<td>x11</td>
<td>x66</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3.</td>
<td>zero</td>
<td>x0A</td>
<td>x88</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4.</td>
<td>zero</td>
<td>x02</td>
<td>xAA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5.</td>
<td>write</td>
<td>x12</td>
<td>x96</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6.</td>
<td>zero</td>
<td>x0B</td>
<td>xE8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### Table 4.5
Processing element activity for bit-serial addition.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>0&lt;sub&gt;a&lt;/sub&gt;</td>
<td>1 → E</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>0&lt;sub&gt;b&lt;/sub&gt;</td>
<td>b0 → A</td>
<td>1</td>
<td>b0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>1&lt;sub&gt;a&lt;/sub&gt;</td>
<td>A → B</td>
<td>A</td>
<td>b0</td>
<td>b0</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>1&lt;sub&gt;b&lt;/sub&gt;</td>
<td>a0 → A</td>
<td>A</td>
<td>a0</td>
<td>b0</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>2&lt;sub&gt;a&lt;/sub&gt;</td>
<td>A + B → D</td>
<td>A + B</td>
<td>a0</td>
<td>b0</td>
<td>x</td>
<td>a0 + b0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>2&lt;sub&gt;b&lt;/sub&gt;</td>
<td>D → s0</td>
<td>A + B</td>
<td>a0</td>
<td>b0</td>
<td>x</td>
<td>a0 + b0</td>
<td>1</td>
<td>a0 + b0</td>
</tr>
<tr>
<td>3&lt;sub&gt;a&lt;/sub&gt;</td>
<td>carry(A + B) → C</td>
<td>carry(A + B)</td>
<td>a0</td>
<td>b0</td>
<td>c0</td>
<td>a0 + b0</td>
<td>1</td>
<td>a0 + b0</td>
</tr>
<tr>
<td>3&lt;sub&gt;b&lt;/sub&gt;</td>
<td>b1 → A</td>
<td>carry(A + B)</td>
<td>b1</td>
<td>b0</td>
<td>c0</td>
<td>a0 + b0</td>
<td>1</td>
<td>a0 + b0</td>
</tr>
<tr>
<td>4&lt;sub&gt;a&lt;/sub&gt;</td>
<td>A → B</td>
<td>A</td>
<td>b1</td>
<td>b1</td>
<td>c0</td>
<td>a0 + b0</td>
<td>1</td>
<td>a0 + b0</td>
</tr>
<tr>
<td>4&lt;sub&gt;b&lt;/sub&gt;</td>
<td>a1 → A</td>
<td>A</td>
<td>a1</td>
<td>b1</td>
<td>c0</td>
<td>a0 + b0</td>
<td>1</td>
<td>a0 + b0</td>
</tr>
<tr>
<td>5&lt;sub&gt;a&lt;/sub&gt;</td>
<td>A + B + C → D</td>
<td>A + B + C</td>
<td>a1</td>
<td>b1</td>
<td>c0</td>
<td>a1 + b1 + c0</td>
<td>1</td>
<td>a0 + b0</td>
</tr>
<tr>
<td>5&lt;sub&gt;b&lt;/sub&gt;</td>
<td>D → s1</td>
<td>A + B + C</td>
<td>a1</td>
<td>b1</td>
<td>c0</td>
<td>a1 + b1 + c0</td>
<td>1</td>
<td>a0 + b0 a1 + b1 + c0</td>
</tr>
<tr>
<td>6&lt;sub&gt;a&lt;/sub&gt;</td>
<td>carry(A + B + C) → C</td>
<td>carry(A + B + C)</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>a1 + b1 + c0</td>
<td>1</td>
<td>a0 + b0 a1 + b1 + c0</td>
</tr>
<tr>
<td>6&lt;sub&gt;b&lt;/sub&gt;</td>
<td>b2 → A</td>
<td>carry(A + B + C)</td>
<td>b2</td>
<td>b1</td>
<td>c1</td>
<td>a1 + b1 + c0</td>
<td>1</td>
<td>a0 + b0 a1 + b1 + c0</td>
</tr>
</tbody>
</table>
The processing element instructions are executed in two sequential phases: First, function generator and latch activity take place and second, memory reads and writes are performed. Details of the timing are presented in Section 2.5 and Appendix D of Gealow’s thesis [4]. In instruction 0a, the function generator writes a 1 to Latch E, thereby activating the local write enable of all the PEs. Instructions 0b and 1a, make use of Latch A to read the LSB of integer b into Latch B. Instruction 1b, reads the LSB of a into Latch A. Instruction 2a calculates the sum a + b and writes it into Latch D. Table 4.6 shows the truth table used to determine the function bits for this instruction. Notice that the contents of Latch C do not matter for this operation.

Table 4.6
Function generator bits for 2-bit addition.

<table>
<thead>
<tr>
<th>A</th>
<th>+</th>
<th>B</th>
<th>(C) =</th>
<th>Result</th>
<th>+</th>
<th>Carry</th>
<th>Function Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>f0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(0)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>f1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(0)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>f2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(0)</td>
<td>0</td>
<td>1</td>
<td>f3</td>
<td>f3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>(1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>f4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(1)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>f5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(1)</td>
<td>1</td>
<td>0</td>
<td>f6</td>
<td>f6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(1)</td>
<td>0</td>
<td>1</td>
<td>f7</td>
<td>f7</td>
</tr>
</tbody>
</table>

\[ \times66 \quad \times88 \quad (f7 - f0) \]

Table 4.7
Function generator bits for 3-bit addition.

<table>
<thead>
<tr>
<th>A</th>
<th>+</th>
<th>B</th>
<th>+</th>
<th>C =</th>
<th>Result</th>
<th>+</th>
<th>Carry</th>
<th>Function Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>=</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>f0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>=</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>f1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>=</td>
<td>1</td>
<td>0</td>
<td>f2</td>
<td>f2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>=</td>
<td>0</td>
<td>1</td>
<td>f3</td>
<td>f3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>=</td>
<td>1</td>
<td>0</td>
<td>f4</td>
<td>f4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>=</td>
<td>0</td>
<td>1</td>
<td>f5</td>
<td>f5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>=</td>
<td>0</td>
<td>1</td>
<td>f6</td>
<td>f6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>=</td>
<td>1</td>
<td>1</td>
<td>f7</td>
<td>f7</td>
</tr>
</tbody>
</table>

\[ \times96 \quad \timesE8 \quad (f7 - f0) \]

Next, instruction 2b writes the sum (held in Latch D) into memory. Instruction 3a writes the carry bit of the sum into Latch C. Instructions 3b – 4b read the second LSB of a and b into Latches A and B respectively. Instruction 5a writes the sum of the second LSBs and the carry bit from the sum of the first LSBs (a1 + b1 + c0) into Latch D. The truth table
used to determine the function generator bits is shown in Table 4.7. In this case, taking into the account the carry bit from the previous operation, the contents of Latch C do impact the output of the function generator.

Instruction 5ₖ, like instruction 2ₖ, writes the sum stored in Latch D into memory. Instruction 6ₖ writes the carry bit into Latch C. Instruction 6ₖ starts reading in the third LSB. The remaining bits of the 8-bit sum are handled in a similar fashion.

4.2 Image I/O

Because the PPIP chip is capable of processing 64 × 64 pixels in parallel, 16 chips are required to process a 256 × 256 pixel image in parallel. Each chip processes a different 64 × 64 pixel region as shown in Figure 4.5.

The PPIP chips have two serial input and two serial output lines lines with dedicated Serial Access Memories (SAM) for transferring data to and from the processing element array. Figure 4.6 shows a diagrammatic representation of the SAM structures and how they access the processing elements. Functionally the chip is comprised of two sets of 128 columns of 16 PEs. A single SAM sits at the bottom of a column of 16 PEs. Each serial line accesses half the PEs in a PPIP chip, so that 32 input and 32 output data signals transfer data to and from the 16 chips.

Figure 4.7 shows the transmission format of the image data over each serial input line. The notation for the pixel data is \( (R_{x,y,z})^{(n)} \), where \( x \) is the horizontal pixel position, \( y \) is the vertical pixel position, and \( z \) represents the bit location. The \( (n) \) is used to indicate the order of transmission. \( (R_{62,1,1})^{(2238)} \), for example, represents the 2239th bit sent. It should be written into address bit-1 of pixel location (62,1) denoted by a black square in Figure 4.8.

As shown in Figure 4.7, image data are transferred by bit-planes; a bit-plane consists of a single bit of each pixel in the image. Packets of 128 bits are sent at a time. Once the 128 bits are stored in the SAM cells, they are transferred into the PE Array memory. PE control signals are used to shift the image data up through the PE column. Since each SAM cell has access to a column of 16 PEs, 16 such transfers are required. Transferring a bit-plane, therefore requires 16 transfers of 128-bit packets of image data.
Figure 4.6: Serial Access Memories.
Figure 4.7: Data Transmission Format.
There are two distinct stages to the image input/output process: (1) transferring data to and from the PPIP chip through the serial I/O lines, and (2) exchanging data between the serial access memory cells and the processing element array memory. Image transfers take place by executing the following sequence repeatedly: (1) Read processed data from PE Array memory into SAM cells. (2) Shift processed data out of SAMs onto serial output lines, while shifting in new data from serial input line into SAMs. (3) Transfer new data from SAM cells into PE Array Memory. Section 4.2.1 shows how image data is transferred into and out of the serial access memories. Section 4.2.2 gives a detailed explanation of how data is exchanged between the processing elements and the SAMs.

4.2.1 Transferring Data

Independent of processing element operation, image data is transferred into and out of serial access memories. Figure 4.9 shows the functional representation of the SAM cell. When $SCK1$ is high, Latch 1 captures the data from Latch 2 of the preceding SAM cell. When $SCK2$ is high, Latch 2 captures the data from Latch 1. Data is shuffled through the series of SAM cells by alternating pulses of $SCK1$ and $SCK2$. $STR$ and $SLD$ are used to direct image data to and from the PE Array. When $STR$ is high, input data stored in Latch 1 is transferred to the PE Array through $N_{out}$. When $SLD$ is high, output data in Latch A of the PE is loaded into Latch 2 of the SAM.

Figure 4.10 shows how input (RAW) and output (COOKED) data are transferred into and out of the PPIP chip. First, $SLD$ is asserted to load processed data from the PE into Latch 2 of the SAMs. Although shown as a separate instruction in the figure, $SCK1$ is asserted simultaneously as $SLD$ so that the COOKED data read in from the PEs is also stored in Latch 1. RAW data is also read in from the serial input line at this time. Next, $SCK2$ is asserted to shift the next COOKED pixel out onto the serial output line and shift the RAW data into Latch 2. The transfer continues in this fashion. $SCK1$ and $SCK2$ are
asserted alternately shifting COOKED data onto the serial output line and shifting RAW data into the SAM cells. At the end of the serial shift process, all the COOKED data has been shifted out onto the serial line and all the RAW data has been shuffled into the SAM cells.

4.2.2 Exchanging Data

While the previous section describes how image data is transferred through the serial lines to the serial access memories, this section describes how pixel data is transferred between the processing elements and the SAMs. The sequences of PE instructions that govern this data transfer are called Exchange Sequences or Exchange Vectors. There are three types of Exchange Vectors: Pre-Exchange Vectors, Typical Exchange Vectors, and Post-Exchange Vectors.

After an image has been processed, the first exchanged performed is the Pre-Exchange Vector. This moves processed output data from the PE memory to the SAMs. After the processed data is shuffled out and new input data is transferred into the SAMs, a Typical Exchange Vector is executed. This moves both processed output data from the PE memory to the SAMs and input data from the SAMs to the PE memory. After all the processed data for a bit-plane has been shuffled out onto the serial output line, the last exchange performed is a Post-Exchange Vector. This moves the last set of input data from the SAMs into the PE memory.

To clarify how the pixel data is transferred between the serial access memories and the processing elements, the entire exchange sequence for a single bit-plane is presented. The example will show the interaction between one SAM cell and one column of 16 processing elements.

Figure 4.11 shows a more detailed representation of a serial access memory cell and a column of 16 processing elements. The output data, the input data, the contents of Latch A, and the contents of Latch D are shown for each processing element. The contents of Latch 1 and Latch 2 of the SAM cell that accesses those 16 PEs are shown as well. The output data $M[\text{out}]$ is the resulting output data that has been processed from the last input image. The input data $M[\text{in}]$ is the new input data to be transferred into the array for processing. $A$ and $D$ indicate the contents of Latch A and Latch D, and 1 and 2 indicate the contents of Latch 1 and Latch 2 of the SAM cell, respectively.
Figure 4.10: Transferring image data through serial access memories. The notation $C_x$ is used to denote the processed (COOKED) pixel data that is read into SAM cell $x$. $R_x$ is used for input (RAW) pixel data.
Figure 4.11: Processing Element Column.

Figure 4.12: Pre-Exchange.

Since this example shows only a single column of PEs and a single bit-plane, only one variable will be used to reference pixel data. That is, \( C1 \) will be used rather than \( C_{(63,1,0)} \) to refer to the output pixel data that belongs in pixel location \( (63,1) \), address bit 0. This will indicate which PE the data belongs to while the other variables will be assumed to be x-coordinate 63 and address bit 0.

Before any exchanges take place, processed image data already resides in the output memory location. The first block in Figure 4.12 shows that \( M[\text{out}] \) is filled with processed data \( C0 - C15 \). After the output data is shuffled out and the new input data is transferred in, \( M[\text{out}] \) will be empty and \( M[\text{in}] \) will be filled with input data \( R0 - R15 \) as shown in Figure 4.15.

The Pre-Exchange vector sequence consists of two instructions:
1. \( M[\text{out}] \rightarrow A \)  
   Reading the contents of PE output memory into Latch A.

2. \( SLD \)  
   Loading the contents of Latch 2 of the SAM with the contents of Latch A of PE15.

After the Pre-Exchange, Latch 2 of each of the 128 SAMs contains processed data \( C(127, 15, 0) - C(0, 15, 0) \). This, processed data gets shuffled out while input data \( R(127, 0, 0) - R(0, 0, 0) \) gets shuffled in.

Once the input data is shuffled into the SAMs, a typical exchange sequence transfers input data from Latch 1 of the SAMs into PE memory and loads Latch 2 of the SAMs with processed output data from PE memory.

The Typical Exchange Vector consists of the following instructions:

1. \( D \rightarrow M[\text{in}] \)  
   Writing the contents of Latch D into \( M[\text{in}] \)

2. \( A_{\text{north}} \rightarrow D \)  
   Reading the North neighbors Latch A into Latch D

3. \( M[\text{in}] \rightarrow A \)  
   Reading the contents of \( M[\text{in}] \) into Latch A

4. \( D \rightarrow M[\text{out}] \)  
   Writing the contents of Latch D into \( M[\text{out}] \)

5. \( A_{\text{south}} \rightarrow D \)  
   Reading the South neighbor’s Latch A into Latch D

6. \( M[\text{out}] \rightarrow A \)  
   Reading the contents of \( M[\text{out}] \) into Latch A

7. \( SLD \)  
   Loading Latch 2 of the SAM with the contents of Latch A

At the end of the exchange sequence, the output pixel bit sits in Latch 2 of the SAM cell and the new input pixel bit sits in Latch D of the PE. At the beginning of the exchange, the input pixel from the previous transfer—residing in Latch D—is written into PE memory.

Figure 4.13 shows the first three exchange sequences. Figure 4.14 shows the last three regular exchange sequences. Notice that after the 15 exchanges, all the processed data has been transferred out of memory.

Since all processed data has been transferred out onto Latch 2, an exchange sequence simpler than the Typical Exchange Vector can be utilized. The Post-Exchange Vector is used to transfer \( R14 \) and \( R15 \) into memory. It consists of the following instructions:
Figure 4.13: First three Typical Exchange Sequences.
Figure 4.14: Last three Typical Exchange Sequences.
1. $D \rightarrow M[in]$  
   Writing the contents of Latch D into $M[in]$ 

2. $M[in] \rightarrow A$  
   Reading the contents of $M[in]$ into Latch A 

3. $A_{south} \rightarrow D$  
   Reading the South neighbor's Latch A into Latch D 

4. $D \rightarrow M[in]$  
   Writing the contents of Latch D into $M[in]$ 

Notice that after the Post-Exchange Sequence is complete all the output data has been transferred out onto the serial lines and all the input data has been transferred into the PE Array memory.
4.3 Data Transmission Handshaking

While the PE instruction signals are primarily governed by the software system and issued by the Controller, the signals that handle image data transfer are determined by a combination of control instructions and data-path signals. This section describes the handshaking protocol between the PE Array, Data-Path, and Controller boards. Board-level logic used to handle image I/O for the PPIP chips is also discussed.

The existing Data-Path board sends 128-bit packets in the order described above. The Controller board sends instructions to the processing element array to perform the exchange sequences. The board-level logic on the PE Array board governs the overall data-path handshaking and decoding of PE Array instruction signals.

4.3.1 Communication Between Boards

When transferring an image, the Data-Path board sends 8 sets of 16 packets of 128 bits of data in a pre-determined format. The PE Array board and Data-Path boards use a frame ready (X), frame acknowledge (XA), and a row ready (H), row acknowledge (HA) handshaking protocol to administer the data transmission as shown in Figure 4.16.

The Controller dictates how the image data actually gets shifted into and out of the PE Array memory. Data is transferred one bit-plane at a time, and each bit-plane is transferred into the PE memory through several exchange sequences. The Controller sends array instructions to address the appropriate memory locations and handle read and write instructions as shown in exchange sequences above.

There are two main communication signals between the Controller and the PE Array board: The DPATN (Data-Path Attention) signal goes from the PE Array to the Controller and the DPACK (Data-Path Acknowledge) signal goes from the Controller to the PE Array. A third signal, AR (Array Ready), is used by the Controller to indicate to the state machine that it is ready to start an application sequence.

The PE Array board can request the Controller to execute an exchange sequence by
sending the DPATH signal. The Controller, having intimate knowledge of the data transmission format, keeps track of the Exchange requests in order to determine which type of exchange it has to perform (i.e. Pre-Exchange, Exchange, or Post-Exchange).

The Controller uses the DPACK signal to indicate when it has completed an exchange sequence. The PE Array board decodes SR to be 1 when DPACK goes high. When the PE Array board requests an exchange sequence by asserting DPATH, the state machine waits for SR to go high to know when the Controller has finished executing the exchange sequence. Since the Controller only sends one bit (DPACK), the PE Array board resets the SR signal when it initiates a request for an exchange sequence from the Controller (i.e. SR goes low when DPATH is asserted).

The first time the Controller receives the DPATH signal in the Exchange State, it executes a Pre-Exchange; the last time (17th) it executes a Post-Exchange. The other 15 times it executes a Typical Exchange. After all 17 exchanges have taken place, the Controller starts again with a Pre-Exchange and moves on to the next bit-plane (i.e. uses the next set of address bits).

The board level logic on the PE Array ties the Controller and the Data-Path together. The unfortunate part of this design is that the Controller software must have intimate knowledge of the Data-Path handshaking protocol and data transmission format. In the future an improved design would allow the software to be developed with greater abstraction of the hardware implementation, perhaps by having the Data-Path and Controller communicate directly.

### 4.3.2 Data-Path State Machine

Figure 4.17 shows the data-path handshaking state machine. This state machine is implemented on the PE Array board. In addition to handling board-to-board communication, the state machine generates the serial shift clocks (SCK1 and SCK2) used to transfer data into and out of the PPIP chips.

### 4.3.3 Idle State

When the Controller is ready to start a sequence, it sends the signal DPSET, which gets decoded as AR = 1 by the PE Array board. When AR is high (Array Ready) and the Data-Path board is ready to send a frame, the signal X (Frame Ready) is sent to the PE Array board.

### 4.3.4 Acknowledge Frame Ready (AckFrame) State

The PE Array board acknowledges that a frame is ready (X) by sending the signal XA. When the Data-Path board sees that the PE Array has acknowledged the frame ready signal (XA) and when it is ready to send a row of data, it sends the signal H (Row Ready).

### 4.3.5 Exchange State

The PE Array asserts the DPATH (Data-Path Attention) and XFR (Transfer) signals in this state. The DPATH signal is sent to the Controller to indicate that the Data-Path board is
ready to start transferring RAW data into the array. The XFR signal is an internal state variable used to differentiate between the AckFrame and Transfer States. The DPATN signal is the only communication from the PE Array to the Controller. The Controller, in essence, keeps tracks of the state of the PE Array and Data-Path handshaking by the number of DPATN signals it receives.

The first time the Controller enters the Exchange State, it executes a Pre-Exchange sequence (i.e. transfers processed output data from the PE Memory to the SAM output latches). When finished with the Pre-Exchange, the Controller asserts the DPACK (Data-Path Acknowledge) signal. Board level logic decode SR as 1 when DPACK is high, and as 0 when DPATN is low. SR going high causes the state machine to exit the Exchange State and enter the AckLine State.

### 4.3.6 Acknowledge Line Ready (AckLine) State

The PE Array acknowledges that a row is ready to be sent with the signal HA. The PE Array brings DPATN low indicating to the Controller that it will be ready to perform a data exchange again soon. The Controller waits for the PE Array to bring DPATN high again to know when to issue the instructions to exchange data between the SAMs and PE memory.

### 4.3.7 Transfer State

When the Data-Path board sees the HA signal go high, it starts sending a row of input image data. It brings the H signal low to indicate that data is no longer ready to be sent, but rather is being sent. When the data is finished being sent, the Data-Path board will bring H high, and the state machine will exit the state.
When the PE Array sees \( H \) go low, the \( HA \) signal is brought low again too. The \( HD \) signal gets translated to go low, and a pixel clock later, the \( HQ \) signal gets translated to go low as well. The \( HD \) and \( HQ \) signals start a series of alternating pulses of \( SCK1 \) and \( SCK2 \) (25% duty cycle) based on \( \{/2PCK\} \) twice the Data-Path pixel clock \( (/PCK) \). Figure 4.18 shows the timing of the data transfer.

In the each Exchange State (except for the post-exchange), the signal \( SLD \) is asserted to load COOKED data from the PE Memory into Latch 2 of the SAMs. Board level logic ensures that \( SCK1 \) goes high when \( SLD \) is asserted so that the COOKED data read in to Latch 2 is also stored in Latch 1 of the SAMs. When the data shuffling starts, valid COOKED data already sits in Latch 2 of the last SAM at the end of the serial line. \( SCK2 \) is asserted to shift COOKED data from Latch 1 to Latch 2. The Data-Path board sends RAW data on the rising edge of \( /PCK \). The PE Array board registers the data on the falling edge of \( /PCK \) so that the RAW data is valid when \( SCK1 \) is asserted half a pixel clock later.

\( SR \) is decoded as going low, by \( DPATH \) going low in the AckLine State. When the Data-Path board is finished transferring a row, it bring \( H \) high again causing the state machine to go to the Transfer State. When the Data-Path board is finished transferring the entire frame it brings \( X \) low causing the state machine to exit into the Finish State.

### 4.3.8 Exchange State

The Data-Path board indicates that all the RAW data has been transferred over to the PE Array board by bringing \( H \) high again. This causing the signals \( HD \) and \( HQ \) to go high, stopping the \( SCK1 \) and \( SCK2 \) pulse train. Since all the RAW data has been transferred into
the SAMs and all the COOKED data has been transferred out of the SAMs during the Transfer State, another exchange sequence needs to be performed. In this state, \( DPATN \) is brought high again, signaling the Controller that there is a new set of RAW data in the SAMs to be transferred to memory. The Controller executes the appropriate exchange instructions, transferring input data from the SAMs into the PE memory and processed data from the PE memory to the SAMs.

### 4.3.9 Finish State

The Data-Path board brings \( X \) low when the entire frame has been transferred. This causes the state machine to exit into the Finish State. \( DPATN \) is brought back high to tell the Controller that the image transfer is complete. The Controller then asserts \( DPACK \) (which gets translated into \( SR \)) causing the state machine to return to the initial Idle State.

### 4.4 Interchip Communication

Four perimeter processing elements share one bi-directional interchip communication signal. With 64 processing elements on a side, there are 16 interchip communication signals per side; with four sides, there are 64 in total.

- North \( n_0 - n_{15} \)
- South \( s_0 - s_{15} \)
- East \( e_0 - e_{15} \)
- West \( w_0 - w_{15} \)

Interchip connections should be made between chips as shown in the Figure 4.19.

### 4.5 Power Supplies

There are three power supplies for the PPIP chip. Power supplies \( V_{DD}, V_{PP}, \) and \( V_{HH} \) are each 3.3 volts. They are used to power internal circuitry, wordline drivers and platelines, and interface circuitry respectively. Power supplies \( V_{SS} \) and \( V_{LL} \) serve as ground for the internal circuits and interface circuits, respectively.

### 4.6 Clocking

Four clocks govern processing element operation: two clocks for memory reads and writes, latch and function generator operation (\( CLK0 \) and \( CLK1 \)) and two clocks for interchip communication (\( NCK0 \) and \( NCK1 \)). The four clocks are shown in Figure 4.20. \( CLK0 \) and \( CLK1 \) are used to generate four internal clocks signals with pulse widths of \( t_{p00}, t_{p01}, t_{p10}, \) and \( t_{p11} \). Similarly, \( NCK0 \) and \( NCK1 \) are used to generate four internal clocks with equal pulse widths \( t_{n00}, t_{n01}, t_{n10}, \) and \( t_{n11} \). Sections 2.5 and 2.7 of Gealow's thesis describe the processing element timing in more detail.
Figure 4.19: Interchip Communication.

Figure 4.20: PPIP clock timing.
Table 4.8

PPIP clock timing. $T$ denotes the clock period.

<table>
<thead>
<tr>
<th>General Clocking Scheme</th>
<th>Clock</th>
<th>High</th>
<th>Low</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NCK0</td>
<td>$T/2$</td>
<td>$T/2$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>NCK0</td>
<td>$T/2$</td>
<td>$T/2$</td>
<td>$T/4$</td>
</tr>
<tr>
<td></td>
<td>CLK0</td>
<td>$t_{p10} + t_{p00}$</td>
<td>$t_{p01} + t_{p11}$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>CLK1</td>
<td>$t_{p00} + t_{p01}$</td>
<td>$t_{p11} + t_{p10}$</td>
<td>$t_{p10}$</td>
</tr>
</tbody>
</table>

$T = 100\text{ns}$

<table>
<thead>
<tr>
<th>Clock</th>
<th>High</th>
<th>Low</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCK0</td>
<td>50</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>NCK1</td>
<td>50</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>CLK0</td>
<td>50</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>CLK1</td>
<td>50</td>
<td>50</td>
<td>25</td>
</tr>
</tbody>
</table>

$T = 66\text{ns}$

<table>
<thead>
<tr>
<th>Clock</th>
<th>High</th>
<th>Low</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCK0</td>
<td>33</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>NCK1</td>
<td>33</td>
<td>33</td>
<td>16.5</td>
</tr>
<tr>
<td>CLK0</td>
<td>33</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>CLK1</td>
<td>23</td>
<td>43</td>
<td>17</td>
</tr>
</tbody>
</table>

$T = 50\text{ns}$

<table>
<thead>
<tr>
<th>Clock</th>
<th>High</th>
<th>Low</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCK0</td>
<td>25</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>NCK1</td>
<td>25</td>
<td>25</td>
<td>12.5</td>
</tr>
<tr>
<td>CLK0</td>
<td>25</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>CLK1</td>
<td>18</td>
<td>32</td>
<td>13</td>
</tr>
</tbody>
</table>

In order to distribute these clocks with minimal noise and buffering problems to the 16 PPIP chips, a clock buffer with integrated source-termination resistors was utilized. The clock buffer chosen is an inverting buffer, so the clock scheme needed to be generated is actually inverted from what is seen in Figure 4.20. Table 4.8 shows how the clocks are determined.

The scheme can be arranged so that $t_{p10} + t_{p00} = T/2$. There are then only three design issues to consider: (1) creating a 50% duty-cycle signal (for CLK0, NCK0, and NCK1), (2) creating a signal with duty-cycle greater than 50% (CLK1), and (3) delaying the signals with respect to each other.

Precision silicon delay lines and combinational logic were utilized to implement a flexible
clocking circuit. Limitations and constraints of the delay blocks required that two unique circuits be used to generate the 50% and non-50% duty-cycle clock signals. The two notable constraints were that (1) the minimum delay of the adjustable delay lines is 10 ns and (2) the input signal could not be delayed by more than half the clock period. Once the 50% and greater than 50% duty-cycle clocks are created, two additional delay lines are used to delay the clocks relative to each other.

Figure 4.21 shows the circuit used to generate the 50% duty cycle clock. The circuit consists of an off-the-shelf crystal or CMOS oscillator, a delay line, and two asynchronously resettable edge-triggered latches. Figure 4.22 shows how the output signal generated has 50% duty-cycle even when the oscillator does not. When CLK goes high, the output of Latch 1, OUT, goes high. The delayed output, DLY, clocks Latch 2 causing the output to go high. The output of the latch also serves as the reset to the latch. Therefore when Latch 2 is triggered by DLY the output signal goes high for a brief period of time (i.e. the propagation delay of the reset to output circuitry of the latch). This reset pulse, RST, is used to reset the the first latch, causing OUT to go low. The delay can be precisely adjusted to create the exact pulse width desired:

\[ width = \frac{T}{2} = delay + t_{cq} + t_{rq} \]

Figure 4.23 shows the circuitry used to generate the clock signals with duty-cycle greater than 50%. The circuit consists of a delay block and a set-reset latch. The delay can be adjusted to create a pulse width greater than half the period of the input signal as shown in Figure 4.24:

\[ width = T - delay - t_{sq} + t_{rq} \]
**Figure 4.22:** Timing diagram for 50% duty-cycle circuit. Timing parameters are as follows: $t_{cq}$ represents the propagation delay from clock to output of the latch, $t_{rq}$ represents the propagation delay from the reset input to output of the latch, delay represents the delay of the silicon delay line.

**Figure 4.23:** Greater than 50% duty-cycle clock circuit.

**Figure 4.24:** Timing diagram for greater than 50% duty-cycle circuit. Timing parameters are as follows: $t_{sq}$ represents the propagation delay from set to output of the latch, $t_{rq}$ represents the propagation delay from the reset input to output of the latch.
Chapter 5

Test Board

In order to ensure the proper operation of the full $256 \times 256$ processing element array it was necessary to evaluate and consider several design issues in a simpler test setup. A second generation PE Array test board was designed and built to explore the various design concerns. Specifically, the purposes of the test board were to:

- Test the new PPIP chips
- Test the logic integration
- Test the on-board power supplies and clock generation scheme
- Test the clock, signal buffering, and other board-level problems
- Eliminate the data path flicker

If additional PPIP chips are fabricated in the future, the test board can be used to verify the functionality of the new chips. The software and test procedure used to evaluate the chips is also described.

5.1 Purpose

The final goal of this project is a 16 PPIP chip PE array board that is small enough to fit within a VMEbus chassis and makes use of no test equipment. The goal of the test board is to explore some of the design issues associated with building the final board.

5.1.1 Testing the New PPIP Chips

The first issue that necessitated the need for a test board was to determine which of the PPIP chips worked correctly. In order to reduce the chip size from its original Pin Gray Array (PGA) package and preclude the need for some additional board level components, a revision of the PPIP chips was produced and packaged in a space economical surface mount Quad Flat Pack (QFP) package. Since these chips were produced in small quantity for prototyping, it was not known how many chips would have functional problems due to fabrication variations. Thirty-eight chips were produced of which 16 would be used on the final PE array board.
Soldering and desoldering the 16 chips is cumbersome and could damage the chips. A 16 chip board with sockets for each of the PPIP chips would be too large for a high speed digital board. In order to test the PPIP chips fully, four chips were used so that both vertical and horizontal interchip communication lines could be tested. Zero Insertion Force (ZIF) clam-shell type sockets were placed on the board so that chips could be easily inserted and removed.

5.1.2 Testing the Logic Integration

The board that was used to test the original PPIP chips utilized five 22x10 Programmable Array Logic (PAL) chips and five 16-bit registers. In order to reduce the board size further, all these devices can be integrated into one or two small footprint Complex Programmable Logic Devices (CPLD). The second purpose of the test board, then, was to test the CPLD implementation of the logic and timing used for decoding controller instructions and governing data-path handshaking. Logic test points were placed around the CPLD so that handshaking signals and array instruction timing could be analyzed.

5.1.3 Testing Power Supplies and Clocks

In the original PE Array Test Board, an HP6628A dual channel power supply and an HP6629A quad channel power supply were used to generate the power supplies for the boards and the chips. A Tektronix HFS9003 quad output stimulus system (with minimum pulse widths of 100ps) was used to generate the four clocks. Since one of the goals of the final project is to preclude the need for all test equipment, another function of this test board is to test the implementation of the on-board power supplies and clock generation circuit. Keeping in mind that the PPIP chips themselves might have problems, the test board enables the chips to be tested with the on-board regulators and the clock generation circuit or with the Hewlett-Packard and Tektronix test equipment. The clock circuit used for the test board was a simplified version of the circuit presented in Section 4.6 fixed to operate with a period of 100 ns.

5.1.4 Testing for Board Level Problems

One of the biggest board-level concerns was fanout. Since each of the PPIP chips receive the same clocks and the same instructions signals, the final PE Array board requires that each of these signals drive 16 chips. The CPLD that generates the PE instructions and the circuit that generate the clock signals probably do not source enough current to drive these signals fast enough. From a buffering standpoint, the most conservative design buffers each signal and clock going to each chip. This would involve a large number of clock and signal buffers chips though which would make the board bigger than desired and possible add extra distance between the clock signals and the chips. The function of the test board is to help to determine the necessary amount of signal buffering.

Secondary board-level concerns include crosstalk and ringing. The test board was designed to be as small as possible so that possibilities of crosstalk could be explored. Long traces on the test board can also be examined for any potential signal ringing.
5.1.5 Data-Path Flicker

The last objectives of the test board was to explore the occasional data-path flicker that occurs in the previous PE Array test board. Equipped with logic and clock test points, the new test board facilitates exploration of the data path handshaking to determine where frames might be being skipped and how to go about correcting the problem in the final board.

5.2 Description

5.2.1 Logic Design

Cypress Semiconductor CPLDs were used to implement the board level logic. Cypress parts were chosen since they have a fixed timing model, are “In-System-Reprogrammable” through a JTAG interface, and have simple, easily accessible software development tools. In specific the Cypress CY7C3751-125AC CPLD was chosen since it had a large number of resources (128 Macrocells, 133 I/O pins) and a fast (10ns) propagation delay.

The logic was designed using Cypress's WARP tools. Cypress's Galaxy editor and compiler was utilized to develop the VHDL code, and Cypress's functional simulator, NOVA, was used to test and debug the code.

5.2.2 Board Design

The test board, HDPP-TEST, was designed using Accel Technologies EDA Schematic and Layout tools. (Note: “HDPP” stands for High Density Parallel Processors. It is a legacy name that was originally used to refer to the PPIP chips. Various aspects of the software system and hardware documentation refer to the PPIP chips as HDPP.) Layout was performed manually with the aid of the Accel QUICK-ROUTE auto router tool. The board has 10 layers; 6 signal layers and 4 power supply planes. The size of the board is $9.187" \times 7.700" \times 0.062"$. A photograph of the test board is shown in Figure 5.1.

5.3 Debugging Procedure

A series of software tests were performed on the test board to debug the logic and board problems as well as to determine the functionality of the chips. Each section below describes the software test performed and includes tables below to detail the problem encountered and the solution implemented.

5.3.1 Serial Access Memory

The first program, SAMTEST, tests the functionality of the serial access memory cells. This test merely checks that the data-path handshaking is taking place correctly and shifts the input data through the SAMs. The data does not get written to or read out of the actual processing element memory.
Figure 5.1: Test Board.
### 5.3. DEBUGGING PROCEDURE

<table>
<thead>
<tr>
<th>Symptom:</th>
<th>The controller would not reset.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem:</td>
<td>The controller did not receive the system clock generated on the test board and therefore did not recognize that the test board was active. The problem turned out to be a misinterpretation of the clock driver data-sheets. The polarity of the output enable pins on the clock driver was reversed.</td>
</tr>
<tr>
<td>Solution:</td>
<td>Since these signals are generated in the CPLD, the problem was corrected by inverting the signals in VHDL.</td>
</tr>
<tr>
<td>Result:</td>
<td>The Controller reset okay.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symptom:</th>
<th>The output image is all white or all black regardless of the input image.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem:</td>
<td>The CPLD was not going through the data-path handshaking correctly. This was a result of a startup problem due to an RC time-constant based reset circuit.</td>
</tr>
<tr>
<td>Solution:</td>
<td>The reset circuit was disabled. After closer examination of the state machine, no real startup issues existed, precluding the need for the reset circuit.</td>
</tr>
<tr>
<td>Result:</td>
<td>The output images consisted of random data.</td>
</tr>
</tbody>
</table>
### Symptom
Output images appear to contain random data regardless of the input image.

### Problem 1
Looking at serial clocks used to transfer data into and out of array \( SCK1, SCK2, STR, SLD \) there appeared to be a timing problem. The clock buffers used to distribute \( SCK1, SCK2, STR, SLD \) were actually inverting buffers rather than non-inverting buffers. This inversion was not documented in the data-sheets.

### Solution 1
The serial clocks are generated in the CPLD and were therefore inverted.

### Problem 2
There still appeared to be a timing problem. The falling edge (instead of the rising edge) of \( /PCK2 \) was used to generate the serial clocks.

### Solution 2
Corrected an error in the VHDL.

### Result
The serial access memory test works fine.

---

### 5.3.2 Image I/O

The second program, IMGIO, is a fully functioning image input/output test. Input image data gets shuffled into the SAMs and then transferred into the PE memory. Output image data gets loaded into the SAMs from the PE memory and shuffled out through the SAMs.
5.3. DEBUGGING PROCEDURE

<table>
<thead>
<tr>
<th>Symptom:</th>
<th>Images were not read from or written to memory correctly. Output images contain random image data. The value of the random data would be the same for an entire column of 16 PEs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1:</td>
<td>Latch Instructions ($l_9...l_1$) were decoded from the Array Instructions in reverse order.</td>
</tr>
<tr>
<td>Solution 1:</td>
<td>Corrected error in the VHDL.</td>
</tr>
<tr>
<td>Problem 2:</td>
<td>Latch A was inverted with respect to the behavior expected from the application sequence.</td>
</tr>
<tr>
<td>Solution 2:</td>
<td>There was a logic error on the previous generation PPIP chips that caused the $l_9$ logic block to be inverted. A software patch was put in place to work around this. Since the logic decoding error was corrected in the revised PPIP chips, the software workaround needed to be reversed. (Note: In order to use the previous generation test board, the software workaround needs to be put back in place. The procedure to do this is documented in Appendix C.4).</td>
</tr>
<tr>
<td>Problem 3:</td>
<td>There appeared to be a timing problem with Op-Code $i_2,i_1,i_0$. The signals appear to be delayed by a full clock cycle.</td>
</tr>
<tr>
<td>Solution 3:</td>
<td>Corrected an error in the VHDL that caused these signals to be registered by an extra clock cycle.</td>
</tr>
<tr>
<td>Result:</td>
<td>Able to store and retrieve images from memory okay.</td>
</tr>
</tbody>
</table>

5.3.3 Interconnect Function

The third program, MOVETEST, simply moves pixel data from one processing element to its neighbor. Its function is to test the communication between PEs as well as the interchip communication lines between chips to see if image data can be passed correctly over chip boundaries.
Symptom: A few adjacent pixels appear to be swapped.

Problem: Two interchip communication signals were crossed.

Solution: Board level traces were manually rerouted.

Result: Interconnect function test works fine.

5.3.4 Boolean Function Generator

This test simply tests the basic functions of the PE by adding a scalar pixel value to each input pixel (ADDTEST).

Symptom: The processing appears to work fine in applications where little processing takes place (such as IMGIO and MOVETEST). In applications with more complex processing (such as ADDTEST and MEDIANTEST), the output images contain all black pixels.

Problem: The problem was traced to a short between two pins of a PPIP socket \(f_0\) and \(f_1\). Since these signals are well correlated in a number of applications the error was not easily found. The problem only arises in applications where \(f_0 = 1\) and \(f_1 = 0\) or vice versa. To complicate this issue further, there was some small non-zero resistance between the pins. When the signals were being driven with different values (e.g. \(f_0 = 1\) and \(f_1 = 0\)) both signals would rise up approximately 700-800mV. It turned out that this voltage level was high enough to trip a logic high on the logic analyzer, but not enough for the PPIP chips. The problem was finally found after analyzing the signals with the oscilloscope.

Solution: Removed a stray piece of solder barely connecting two adjacent pins on one of the chip’s sockets.

Result: ADDTEST, MEDIANTEST and the other applications work okay.
Table 5.1
PPIP Memory Test.

<table>
<thead>
<tr>
<th>Refresh Interval</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>≥ 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.27 ms</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>1.64 ms</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>819 µs</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>410 µs</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>205 µs</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>102 µs</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>51.2 µs</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>25.6 µs</td>
<td>11</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

5.4 Data-Path Flicker

When running an image processing sequence in real-time, an occasional flicker is noticed on the output display. By examining the data-path handshaking and the controller signals it was discovered that the controller would sometimes not send its handshaking signals fast enough. As a result frames were occasionally skipped. The problem was traced back to the host computer. Sequences of array instructions are stored in memory on the Controller board and are called by the host computer in real-time. Of the host cannot call the sequences fast enough the handshaking sequence does not finish correctly and a frame is skipped. The handshaking sequence drops the current frame in order to be able to process the next valid frame. This problem was effectively fixed by running the host (SPARC IPX station) in single-user mode (Appendix C.8 provides instructions for this). By running the workstation in this mode, only the executed sequence runs on the computer's CPU; no other daemons or processes can run. Freeing up the host computer's resources fixed the data-path flicker problem. In the future, this problem will be prevented by replacing the host with a fast personal computer or by modifying the controller architecture.

5.5 PPIP Chip Test Results

The functionality of the PPIP chips was verified by testing the various aspects of the processing element and by performing a comprehensive memory test. It was not possible to obtain 16 fully functional PPIP chips. Of the 38 chips fabricated, the 16 best chips were chosen for the final board. An additional 12 chips exhibited visual defects in the processed image, and the remaining 10 chips did not function at all.

All 128 bits of memory of each processing element were tested. Table 5.1 shows the characteristics of the memory with respect to various refresh intervals. The relatively fast refresh interval of 102 µs was used in final operation.

Three applications were used to test the functionality of the processing element: (1) straight image I/O, (2) smoothing and segmentation, and (3) median filtering. Table 5.2 shows the
Table 5.2
PPIP Functionality Test.

<table>
<thead>
<tr>
<th>Test Sequence</th>
<th>Number of Discrepancy Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Image I/O</td>
<td>16</td>
</tr>
<tr>
<td>Smooth &amp; Segmentation</td>
<td>14</td>
</tr>
<tr>
<td>Median Filter</td>
<td>5</td>
</tr>
</tbody>
</table>

number of discrepancies between the output of the PE Array and the actual image in terms of bits. The median filter application utilizes all aspects of the processing element. A problem in one processing element causes errors in the surrounding 9 pixels—hence the large number of discrepancy bits.

The chips were tested with operating clocks of 10MHz, 15MHz, and 20MHz: The chips operated fine with the clock of 10MHz. Chip performance was degraded at 15MHz, and extremely degraded at 20MHz.
Chapter 6

Final Board

After testing the various components and aspects of the processing element array test board, a final 16 chip board was designed. The objective of this board is to be able to process $256 \times 256$ pixel images for the machine vision based intelligent vehicle control system described in Section 1.2. The most notable feature of the new final board is the data-path flexibility. While the current Data-Path board is limited to one NTSC image in and one NTSC image out, the next generation Data-Path board will take three cameras as input and will output processed images to the high-level processor as well as to a display. In order to facilitate both legacy and future designs, the new PE Array board must have flexible data and control paths.

6.1 Description

The 16 chip board is designed to work in three phases of operation:

- Housed in the VMEbus chassis, working with the existing Data-Path and Controller boards.

- Housed in the VMEbus chassis, working with the existing Controller board and a new multiple input/output, Data-Path board.

- Housed independent of the VMEbus chassis, working with a new digital multiple I/O Data-Path board, and a new VMEbus independent Controller board.

In order for the PE Array to interface with the existing systems and works with future systems effectively, a two board design was employed. The first board, HDPP-16, contains 16 PPIP chips, a CPLD for logic decoding and handshaking, clock and signal buffers. HDPP-16 also contains headers to connect power, the JTAG interface, data-path and controller interfaces. The second board, HDPP-CON, connects the HDPP-16 to the existing Data-Path and Controller boards. It contains the power supplies, differential line drivers and receivers, twisted pair cable connectors to connect to the existing system, and
data-path and controller interfaces to HDPP-16. Given the successful operation of the logic on the HDPP-TEST board, most of the design emphasis for the final board was placed in board-level design.

6.1.1 Board Design

Both the boards, HDPP-16 and HDPP-CON, were designed using Accel Technologies EDA Schematic and Layout tools. The layout was performed manually and with the aid of the Cadence CCT SPECCTRA autorouter tool. The HDPP-CON board has 8 layers; 6 signal layers and 2 power supply planes. The HDPP-16 board has 14 layers; 10 signal layers and 4 power supply planes.

The HDPP-CON board is the size of a standard double-height VMEbus card, 9.187" × 6.299" × 0.062". The HDPP-16 board is 8.550" × 4.900" × 0.078" in size. A photograph of the boards can be seen in Figure 6.1 and 6.2. The 16 chip PPIP board plugs into the connector board.

6.2 Design Issues

A few additional board-level design issues needed to be considered in expanding the four chip test board to the final sixteen chip board. The first concern is a potential buffering problem that could arise in attempting to drive instruction signals and clocks to 16 PPIP chips. The second concern is based on the long signal traces introduced by the two board design. Long signal traces could potentially introduce undesired signal reflections.

6.2.1 Buffering

The first issue is whether the outputs of the CPLD would be able to drive the sixteen PPIP chips. The test board was utilized to understand the implications of driving sixteen chips. Because instruction signals only drive four PPIP chips, no real buffering problems exist on the test board. The timing characteristics of the instruction signal can be better understood by removing one PPIP chip at a time and observing the impact on rise-time. By understanding the loading effect of one PPIP, the behavior of instruction signals on the 16 chip board can be anticipated.

From the test board, the observed rise time of a typical instruction signal driving four PPIP chips was approximately 5.5 ns. Each PPIP chip was found to contribute 0.2 - 0.4 ns of rise-time. Rise-time is proportional to the total capacitance seen by the signal. The rise time is calculated by

\[ t_{\text{rise}} = (C_1 + C_2 + C_3 + C_4 + C_o) \times K = 5.5 \text{ ns} \]

where the capacitance for HDPP1–HDDP4 is denoted \( C_1 - C_4 \) and all the other board-level capacitance and PLD capacitance is lumped into \( C_o \). Given \( C_i \times K > 0.2 \text{ ns} \), the largest possible loading affects due to \( C_o \) are:

\[ C_o \times K = 5.5 \text{ ns} - 4 \cdot (0.2 \text{ ns}) = 4.7 \text{ ns}. \]
Figure 6.1: 16 PPIP Board.
Figure 6.2: Connector Board.
6.2. DESIGN ISSUES

With 16 chips, the worst-case rise-time for the instruction signals is:

\[ t_{\text{rise}} = (C_1 + \ldots + C_{16}) \cdot K + C_0 \cdot K < 16 \cdot (0.4 \text{ ns}) + 4.7 \text{ ns} = 11.1 \text{ ns} \]

Although this is quite a slow rise-time, it would probably not cause functional problems. The PE instruction signals are stored by internal latches. The timing for the latches is designed so that the signals going to the chip should be stable before the rising edge of \( CLK_1 \) and remain stable until after the falling edge of \( CLK_0 \). The instruction signal starts rising up 14 ns before the rising edge of \( CLK_1 \) so that a worst-case rise-time of 11 ns would still ensure stability before the rising edge of \( CLK_1 \). The instruction signal is also valid for several tens of nanoseconds after the falling edge of \( CLK_0 \).

The only potential concern with such a long rise-time is power dissipation. The instruction inputs on the PPIP chip are connected to buffers formed using CMOS inverters. When the input to the inverter is higher than than the \( V_{tn} \) (threshold on the n-channel device) but lower than \( V_{dd} - V_{tp} \) (threshold on the p-channel device) a large DC current may flow from \( V_{dd} \) through the devices to ground. Average power dissipation due to this “short-circuit” current is generally relatively small if the input transitions are reasonably fast. Since the transitions could be quite long—as long as 11 ns—power dissipation could be significant. It was worth exploring whether or not the signal rise-time could be decreased without too much difficulty. By simply buffering each signal from the CPLD with one buffer, the rise-time can be decreased. The output current drive of the buffer is greater than the CPLD, so that the signal does not slew for as long. Rise-time, and hence power dissipation, is effectively reduced.

While the instruction signals are buffered 1:16 (1 signal to 16 PPIP chips), the eight clocks (two for processing, two for interchip communication, and four for data transfer) are buffered in a one-to-one fashion.

6.2.2 Ringing

A second board level concern is signal reflections. Signals passed between the two boards (HDPP-CON to HDPP-16) tend to be quite long. The critical signal length to be concerned about for ringing is determined by the rise-time of the signals:

\[ \text{length}_{\text{critical}} > \frac{1}{6} \cdot \frac{\text{risetime}}{\text{delay}} \]

The \( \text{delay} \) is the propagation delay of the printed circuit board material. For standard FR-4 material, \( \text{delay} \approx 150\text{ps/in} \). A conservative estimate of rise-time between 5.0 – 10 ns yields

\[ 5.6'' < \text{length}_{\text{critical}} < 11.1''. \]

Several signals in the the final board design have trace lengths in this range. Most of these, however, are data signals. Because data signals are registered, the impact of small signal reflections is minimal.

Unlike the registered data signals, the behavior of the clock signals are sensitive to reflections. In order to determine which clock signals would require termination it was necessary to look at the trace lengths to see if they exceed \( \text{length}_{\text{critical}} \). The only clock
signals on HDPP-16 whose signal lengths might be long enough are the clocks that drive the PPiP chips (CLK0, CLK1, NCK0, NCK1, SCK1, SCK2, STR, SLD). These clocks do not exhibit signal reflections because the the clock buffers that generate them have internally source terminated outputs. The other clock signals on HDPP-16 all have short trace lengths.

The signals that that go between the HDPP-16 and HDPP-CON boards, however, are quite long. Three particularly long signals (DCK, X, H) from HDPP-CON to HDPP-16 were source terminated. One signal from HDPP-16 to HDPP-CON (CCK, T) was source terminated. Source terminating these signals signals was probably an unnecessarily conservative design measure, but the decrease in rise-time does not negatively impact the system. The value of the termination resistor is 50Ω. The value was chosen to match typical impedance numbers.

6.3 Debugging Procedure

The same software testing procedure used to test HDPP-TEST was used for HDPP-16. Two mostly working PPiP chips (few bad memory cells) were soldered down onto HDPP-16 for testing. In the event of any board mistakes, the errors could be fixed on an unpopulated board which would then be populated with the 16 good chips.

Optimistically the Image I/O test was attempted first but did not work. The Serial Access Memory test was unsuccessfully attempted next.

6.3.1 Serial Access Memory

| Symptom:       | Output images contain all white pixels. |
| Problem:       | One of the clocks drivers output enable pins was not routed. This turned out to be a mistake due to a software bug with the SPECCTRA autorouter. This software problem is under investigation with the vendor. |
| Solution:      | A manual route was created. |
| Result:        | Serial Access Memory Test appeared to work for part of the chips. |
6.3. DEBUGGING PROCEDURE

| Symptom: | Only half of the serial lines appear to work. |
| Problem: | The input serial lines were not getting the right data. The two connectors used to receive the input data from the Data-Path board were swapped in the layout. This turned out to be an oversight with the layout software when renumbering connectors. |
| Solution: | Switch the cables used to receive the input data. |
| Result: | SAMTEST worked |

### 6.3.2 Image I/O

| Symptom: | Output image contains all white pixels. |
| Problem: | Turned out that $f_w$ was floating high. After probing around and soldering desoldering the appropriate connections, the problem “disappeared”. |
| Solution: | The source of the problem was never confirmed. It appears to have been a solder problem that was fixed by touching up the connections. When a new set of boards were carefully populated this problem did not manifest. |
| Result: | Image I/O and all processing tests work fine. |

### 6.3.3 PE Functionality Tests

After the small problems found through the Serial Access Memory and Image I/O tests were corrected, new HDPP-CON and HDPP-16 boards were populated. When tested, all the processing appeared to work correctly. Two subtle problems, however, were noticed:

1. Most applications worked fine except median filtering. It turned out that two PPiP chip pins were soldered together. The solder short was corrected and then all applications seemed to work.

2. After careful examination of certain applications there also appeared to be 12 “bad spcts”. It was quickly evident that two interchip communication lines had been crossed. The source of error was in the land patterns used for the QFP, and was therefore duplicated.
for all the chips. The PPIP land pattern (ZIF socket) for the test board did not have this problem.

6.3.4 Board Revision

A revision of the HDPP-16 board was fabricated. Modifications included: correcting the interchip communication error; providing 3.3 Volts to the data-path and controller headers to provide the power supplies for future designs; adding the flexible clock circuit described in Section 4.6.

The revised board, HDPP-16-002, had no design or fabrication problems. Since there were not 16 flawless PPIP chips, the best chips were chosen and placed on the board. The PE Array board operates successfully with a clock speed of 10MHz and a refresh interval of 0.1 ms.
Chapter 7

Conclusion

This thesis has described the design and implementation of a pixel-parallel image processing system for use in an intelligent vehicle control system. This chapter summarizes the project and presents suggestions for future work.

7.1 Summary

Digital image processing is computationally very intensive. In many intelligent vehicle applications, the majority of the computation used for image processing is not based on performing very complex operations to a few pixels, but rather, repeatedly performing a few relatively simple operations to every pixel. This premise motivated a machine vision system designed specifically for the computational requirements of an intelligent vehicle applications. The image processing system consists of a low-level pre-processor and a high-level processor. A Pixel-Parallel Image Processor (PPIP) chip integrating 4096 Processing Elements (PE) on a single integrated circuit is proposed as the basis for the low-level pre-processor.

A four chip test board was built to test the PPIP chips and to explore design issues for the final board. Thirty-eight chips were tested. The 16 best chips operate with a refresh interval of 0.1 ms and a clock speed of 10MHz with minor errors. If a new run of PPIP chips is fabricated, the test board can also be used to verify the functionality of the new chips.

A 16-chip 256 × 256 PE Array board was implemented using a two-board design. The PE Array board operates without test equipment and is smaller in size than the original goal. All the logic used to govern array instruction decoding and data-path handshaking is implemented in one CPLD. The CPLD is “In-System Reprogrammable” through the JTAG interface. This reprogrammability facilitates modifications to the handshaking protocols as the system evolves. The flexible data and control path design enables the PE Array to use existing legacy hardware as well as future Data-Path and Controller boards.

A qualitative comparison between the PPIP and various other image processing architectures was presented. The comparison suggested that the PPIP is very efficient for local neighborhood operations and could, in a number of cases, replace lots of dedicated application specific hardware. However, the PPIP is not flexible for some algorithms requiring more
complex low-level processing. Without having utilized the PPIP in a real intelligent vehicle application, it is not clear what system-level limitations are imposed by the data-path limitations and memory capacity.

7.2 Future Work

The value of the pixel-parallel image processing system for intelligent vehicle applications can only be assessed by its performance in an actual system. Work is currently underway to demonstrate such a machine vision system. The first stage of the project includes the integration of three wide-dynamic range CMOS imagers with the processing element array.

In the future, a high-level serial processor, such as a DSP, could take the output of the pre-processed image and extract real-world parameters. A system-wide controller needs to be designed to tie the high-level and low-level processors together. Receiving output data from the high-level image processor, the system controller also issues control instructions to the electro-mechanical interface of a controllable vehicle.

In order to make the machine vision system hardware practical for actual vehicle tests, a new portable controller could be designed. A simplified controller architecture would preclude the need for the VMEbus chassis and host computer. The controller would consist of SRAM and four finite state machines (FSMs). Basically, it would operate as follows: A laptop personal computer would compile image processing applications into sequences of array instructions which would, in turn, be downloaded into SRAM from the computer; the first FSM would be used to issue instructions directly to the PE Array board; the second FSM would be used to handle image I/O; the third FSM would be used to handle memory refreshes; the fourth FSM would be used to orchestrate the timing of the first three state machines.

The software application development framework must be extended to incorporate the high-level processing. Ideally, a “smart” compiler could be designed to divide the image processing computation appropriately between the low and high level processors. For a stereo vision algorithm, for example, high-level code could be developed. Bit-serial sequences would be compiled for the PE Array and high-level global operations would be compiled for the high-level processor. Applications need to be developed, run, and tested on the hardware.

While the pixel-parallel image processor is very efficient at performing local operations, addressing some of the drawbacks of the PPIP in a revised chip could greatly improve its utility in a machine vision system. One of the drawbacks of the PPIP is that the memory requirements for performing multiple applications could potentially exceed the 128-bits available per processing element. This problem is compounded by the fact that data transfer is slow. External memory cannot be utilized efficiently because it takes 655 \( \mu \)s to transfer an 8-bit image. Accessing memory 20 times alone would require over 13 ms.

Another limitation of the current data-path is that data is not available until the entire image has been processed. The low-level/high-level architecture cannot be efficiently pipelined since, the high-level processor must wait for the PE Array to finish processing the entire image before it can start processing any new data.

Some suggestions for improving the PPIP chip include: (1) integrating more PEs on
a single chip, (2) utilizing a wider data-path, and (3) improved facilities for performing processing while data is being transferred.

Improvements in semiconductor process technology will enable the full $256 \times 256$ array of PEs to be integrated onto a single chip. To improve the data-path, perhaps, the PPIP chip could transfer an entire column of processing elements' data at a time. Repackaged in a Ball Grid Array (BGA) package, 256 pins could be dedicated to image I/O. A wide data-path would offer a number of benefits. First, it would facilitate column-wide processing, such as the Hough transform, in an off-chip co-processor. Second, the wide data-path would enable the usage of external memory since it would only take $256 \times 8 \times 40$ ns = 81.92 $\mu$s to transfer an entire image to and from memory. Lastly, software improvements in the controller architecture would enable better facilities for performing image processing while data is being transferred.

In order to truly assess the value of this system, algorithms must be developed to exploit the pixel-parallel architecture. It is not fair to make system-level comparisons utilizing image processing applications that have been developed for serial processors. A better comparison can be made by comparing applications that have been developed for the pixel-parallel architecture against image processing applications developed for other architectures.
References


Appendix A

Test Board

This section provides details about the test board including VHDL for on-board logic and full schematics.

A.1 VHDL

The VHDL files used for the test board are located on COLOB in:

```
./homes/zubair/src/vhdl/hdpp-test/
    hdpp.vhd
    control_dec.vhd
    dp_handshaking.vhd
    hdpp_clock.vhd
```

A.2 Clocks

A signal generator connected through the SMA connectors can be used to generate the clocks by removing the solder that ties the clock buffers to the clock generator circuit. A script has been written to generate the appropriate clocks given the $t_{p00}$, $t_{p01}$, $t_{p11}$, $t_{p10}$ parameters. The script generates the inverted clocks from what the chips. The signals are reinverted by the inverting clock buffers. To operate the Tektronix signal generator through the GPIB bus, use the following commands filling in the appropriate timing parameters:

```
./homes/zubair/bin/sparc/findinst
cd /homes/zubair/src/xapp-4/rtdemo
./488/hfs9000-ns.sh tp00 tp01 tp11 tp10
```

The on-board clock generator implements a simple 100ns timing scheme.

A.3 Board Errors

The nets V4 and V5 on Figure A.4 are crossed due to an error in the schematic symbol definition. The traces were manually swapped to fix the error.
A.4 Schematics

The board schematics, library files, and PC board layout are located on COLOB in ACCEL binary format:

```
/homes/zubair/hdpp/hdpp-test/
    hdpp-test.sch
    hdpp-test.pcb
    hdpp-test.lib
```

Schematics are shown in Figures A.1 - A.7.
Figure A.1: Controller Interface.
Figure A.2: Data Path Receiver Interface.
Figure A.3: Data Path Transmitter Interface.
Figure A.4: HDPP Devices.
Figure A.5: CPLD and Data Path Handshaking.
Figure A.6: Clock Drivers.
Figure A.7: Power Supplies.
Appendix B

Final Board

B.1 VHDL

The VHDL files used for the final board are located on COLOB in:

```
/homes/zubair/src/vhdl/final_hdpp_16/
    hdpp.vhd
    ctrl.vhd
    dp_hs.vhd
    clocks.vhd

/homes/zubair/src/vhdl/hdpp-clocks/
    clocks.vhd
```

B.1.1 Top Level VHDL

Location on COLOB:
```
/homes/zubair/src/vhdl/final.hdpp.16/hdpp.vhd
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.all;

---------------------------
-- ENTITY DECLARATION --
---------------------------

entity hdpp is port ( 
-----------------------------

--Control Path Inputs
-----------------------------

103
CCK, NCCK: in std_logic;
RstIn: in std_logic;
AI: in std_logic_vector(31 downto 0);

------------------------
--Data Path Inputs
------------------------

nPCK, nPCK2: in std_logic;

nX: in std_logic;
nH: in std_logic;
nB: in std_logic;
PCKLKLK: in std_logic;
CCKLKLK: in std_logic;

DpRx: in std_logic_vector(3 downto 0);

------------------------
--Control Path Outputs
------------------------

nSCKEN: out std_logic;
nCLKEN: out std_logic;
nRAWEN: out std_logic;

I3: buffer std_logic;
I: buffer std_logic_vector(2 downto 0);
LA, LB, LC, LD, LE: buffer std_logic;
FN, FS, FE, FW: buffer std_logic;
P: buffer std_logic_vector(7 downto 0);
A: buffer std_logic_vector(6 downto 0);

AS4r: out std_logic;
PCKLKr: out std_logic;
CCKLKr: out std_logic;
DPERRr: out std_logic;
DPATNr: out std_logic

;

------------------------
--Data Path Outputs
------------------------

DPATN: buffer std_logic;
nXA: buffer std_logic;
nXA: buffer std_logic;
XFR: buffer std_logic;
DPERR:       buffer std_logic;
SCK1:        buffer std_logic;
SCK2:        buffer std_logic;
STR:         buffer std_logic;
SLD:         buffer std_logic;
nAR:         buffer std_logic;
DPACK:       buffer std_logic;
SR:          buffer std_logic;

DpTx:        buffer std_logic_vector(2 downto 0);

--Extra I/O's
ExInput:     in std_logic_vector(11 downto 0);
ExOutput:    buffer std_logic_vector (18 downto 0)
);

USE WORK.my_control_dec_pkg.ALL;
USE WORK.my_dp_handshaking_pkg.ALL;
USE WORK.my_hdpp_clk_pkg.ALL;
ATTRIBUTE pin_avoid of hdpp: ENTITY is "6 46 76 116";
ATTRIBUTE part_name of hdpp:ENTITY is "C375i";
ATTRIBUTE pin_numbers of hdpp:ENTITY IS
"A(0):115 " &
"A(1):107 " &
"A(2):106 " &
"A(3):105 " &
"A(4):104 " &
"A(5):97 " &
"A(6):32 " &
"AI(0):147 " &
"AI(1):148 " &
"AI(10):25 " &
"AI(11):26 " &
"AI(12):42 " &
"AI(13):43 " &
"AI(14):44 " &
"AI(15):45 " &
"AI(16):63 " &
"AI(17):64 " &
"AI(18):65 " &
"AI(19):66 " &
"AI(2):149 " &
"AI(20):82 " &
"AI(21):83 " &
"AI(22):84 " &
"AI(23):85 " &
"AI(24):136 " &
"AI(25):137 " &
"AI(26):138 " &
"AI(27):11 " &
"AI(28):12 " &
"AI(29):13 " &
"AI(3):150 " &
"AI(30):14 " &
"AI(31):157 " &
"AI(4):2 " &
"AI(5):3 " &
"AI(6):4 " &
"AI(7):5 " &
"AI(8):23 " &
"AI(9):24 " &
"AS4R:37 " &
"CCK:19 " &
"CCKLK:132 " &
"CCKLKR:98 " &
"DPACK:47 " &
"DPATN:73 " &
"DPATNR:112 " &
"DPERR:67 " &
"DPERR:110 " &
"DPRX(0):33 " &
"DPRX(1):34 " &
"DPRX(2):35 " &
"DPRX(3):15 " &
"DPTX(0):16 " &
"DPTX(1):17 " &
"DPTX(2):18 " &
"EXINPUT(0):39 " &
"EXINPUT(1):144 " &
"EXINPUT(10):58 " &
"EXINPUT(11):59 " &
"EXINPUT(2):156 " &
"EXINPUT(3):155 " &
"EXINPUT(4):158 " &
"EXINPUT(5):153 " &
"EXINPUT(6):152 " &
"EXINPUT(7):146 " &
"EXINPUT(8):145 " &
"EXINPUT(9):154 " &
"EXOUTPUT(0):123 " &
"EXOUTPUT(1):124 " &
"EXOUTPUT(10):118 " &
"EXOUTPUT(11):114 " &
"EXOUTPUT(12):113 " &
"EXOUTPUT(13):109 " &
"EXOUTPUT(14):108 " &
"EXOUTPUT(15):51 " &
"EXOUTPUT(16):53 " &
"EXOUTPUT(17):56 " &
"EXOUTPUT(18):57 " &
"EXOUTPUT(2):125 " &
"EXOUTPUT(3):126 " &
"EXOUTPUT(4):127 " &
"EXOUTPUT(5):128 " &
"EXOUTPUT(6):129 " &
"EXOUTPUT(7):131 " &
"EXOUTPUT(8):103 " &
"EXOUTPUT(9):119 " &
"F(0):92 " &
"F(1):88 " &
"F(2):93 " &
"F(3):87 " &
"F(4):94 " &
"F(5):86 " &
"F(6):95 " &
"F(7):96 " &
"FE:8 " &
"FN:89 " &
"FS:7 " &
"FW:9 " &
"I(0):79 " &
"I(1):78 " &
"I(2):77 " &
"I3:52 " &
"LA:30 " &
"LB:29 " &
"LC:28 " &
"LD:27 " &
"LE:75 " &
"NAR:69 " &
"NCCK:22 " &
"NCLKEN:49 " &
"NH:134 " &
"NHA:68 " &
"NPCK2:102 " &
"NPCK:99 " &
"NRAWEN:36 " &
"NSCKEN:48 " &
"NX:133 " &
"NXA:72 " &
"PCKLK:159 " &
"PCKLKR:91 " &
"RSTIN:135 " &
"SCK1:143 " &
"SCK2:122 " &
"SLD:74 " &
"SR:117 " &
"STR:55 " &
"XFR:70"
end hdpp;

-----------------------------

-- ARCHITECTURE DECLARATION --
-----------------------------

architecture hdpp_arch of hdpp is
  signal RST: std_logic;
  signal HD,HQ: std_logic;
  signal AIEN: std_logic;

-----------------------------

--Control Path Signals
-----------------------------

  signal OP_CODE: std_logic_vector(2 downto 0);
  signal PE_LATCHES: std_logic_vector(0 to 4);
  signal PE_INTERCHIP: std_logic_vector(0 to 3);
  signal PE_ALU_FUNC: std_logic_vector(7 downto 0);
  signal PE_MEM_ADDRESS: std_logic_vector(6 downto 0);
  signal INST DECODE: std_logic_vector(4 downto 0);
  signal AI_R: std_logic_vector(31 downto 0);
  signal OP_CODE_R: std_logic_vector(2 downto 0);
  signal AS4: std_logic;
--Data Path Signals
----------------------------------
signal SLDSET,SLDCLR,STRSET,STRCLR: std_logic;
signal DPSET,DPCLR: std_logic;
----------------------------------

--Signals that need inverting
----------------------------------
signal X: std_logic; --Input
signal H: std_logic; --Input

signal AR: std_logic; --Output
signal XA: std_logic; --Output
signal HA: std_logic; --Output
signal SCKEN: std_logic; --Output
signal CLKEN: std_logic; --Output
signal RAWEN: std_logic; --Output

signal Dummy_In: std_logic_vector(21 downto 0);
signal Dummy_Out: std_logic_vector(21 downto 0);

----------------------------------

--MAIN VHDL CODE
----------------------------------

begin
----------------------------------

--Reset Signal
----------------------------------

RST <= '0';

----------------------------------

--Negate/Invert Signals
----------------------------------

--Negate Input Signals
X <= not(nX);
H <= not(nH);

--Negate Output Signals
nXA <= not(XA);
nHA <= not(HA);
nAR <= not(AR);

----------------------------------
nRAWEN <= '0';
nCLKEN <= '1';
nSCKEN <= '1';

--Control Path
--Register Array Instructions (AI)

CP_AI_register: process
begin
  wait until CCK='1';
  AI_R <= AI;
end process;

OP_CODE       <= AI_R(26 downto 24);
PE_MEM_ADDRESS <= AI_R(23 downto 17);
PE_ALU_FUNC    <= AI_R(16 downto 9);
PE_INTERCHIP   <= AI_R(8 downto 5);
PE_LATCHES     <= AI_R(4 downto 0);

INST_DECODE   <= AI_R(21 downto 17);

--Control Path
--Decode Instructions for Data Path
--Handshaking and for STR/SLD Clocks

control_decode: control_decode PORT MAP (  
  CCKi=>CCK,
  OP_CODEi=>OP_CODE,
  INST_DECODEi=>INST_DECODE,
  RSi=>RST,
  OP_CODE_Ro=>OP_CODE_R,
  I3o=>I3,
  DPSETo=>DPSET,
  DPCLRo=>DPCLR,
  DPAko=>DPAK,
  STRSETo=>STRSET,
  STRCLRo=>STRCLR,
  SLDSETo=>SLDSET,
  SLDCLRo=>SLDCLR,
  SCKNo=>SCKEN,
  CLKNo=>CLKEN,
);
RAWENo=>RAWEN,
AIENo=>AIEN
);

-- Data Path
-- Handshaking State Machine

---
dp_hs0: dp_handshaking PORT MAP (  
nPCKi=>nPCK,
nPCK2i=>nPCK2,
Xi=>X,
Hi=>H,
DPSETi=>DPSET,
DPACKi=>DPACK,
DPCLRi=>DPCLR,
HQo=>HQ,
HDo=>HD,
ARo=>AR,
SRo=>SR,
DPATNo=>DPATN,
XAo=>XA,
HAo=>HA,
XFRo=>XFR,
DPERRo=>DPERR
);
---

-- Data Path
-- Generation of SCK1, SCK2, STR, SLD
-- Serial Access Memory Clocks

---
hdpp_clk0: hdpp_clocks PORT MAP (  
nPCKi=>nPCK,
nPCK2i=>nPCK2,
STRSETi=>STRSET,
STRCLRi=>STRCLR,
SLDSETi=>SLDSET,
SLDCLRi=>SLDCLR,
SRi=>SR,
ARi=>AR,
HDi=>HD,
HQi=>HQ,
STRo=>STR,
SLDo=>SLD,
SCKio=>SCK1,
SCK2o=>SCK2
);

------------------------------
--Control Path
--Register Handshaking Signals
--Back to Controller Board
------------------------------

CP_OUT_register: process
begin
  wait until NCCK='1';
  DPATNr <= DPATN;
  DPERRr <= DPERR;
  CCKLKr <= CCKLK;
  PCKLKr <= PCKLK;
  AS4r <= AS4;
end process;

------------------------------
--Reassign Controller Instruction
--Word Signals to Output Pins
------------------------------

I <= OP_CODE when A1EN='0' else "ZZZ";
A <= PE_MEM_ADDRESS when A1EN='0' else "ZZZZZZZ";
F <= PE_ALU_FUNC when A1EN='0' else "ZZZZZZZZ";
LE <= PE_LATCHES(0) when A1EN='0' else 'Z';
LD <= PE_LATCHES(1) when A1EN='0' else 'Z';
LC <= PE_LATCHES(2) when A1EN='0' else 'Z';
LB <= PE_LATCHES(3) when A1EN='0' else 'Z';
LA <= PE_LATCHES(4) when A1EN='0' else 'Z';
FW <= PE_INTERCHIP(0) when A1EN='0' else 'Z';
FE <= PE_INTERCHIP(1) when A1EN='0' else 'Z';
FS <= PE_INTERCHIP(2) when A1EN='0' else 'Z';
FN <= PE_INTERCHIP(3) when A1EN='0' else 'Z';
--Extra Signals

```
Dummy_In(21 downto 10) <= ExInput(11 downto 0);
Dummy_In(9 downto 5)  <= AI(31 downto 27);
Dummy_In(4)           <= RstIn;
Dummy_In(3 downto 0)  <= DpRx(3 downto 0);
```

```
Dummy_register: process
begin
    wait until CCK='1';
    Dummy_Out <= Dummy_In;
end process;
```

```
ExOutput(18 downto 0) <= Dummy_Out(21 downto 3);
DpTx(2 downto 0)     <= Dummy_Out(2 downto 0);
```

end hdpp_arch;

B.1.2 Decoding Controller Instruction

Location on COLOB:
/homes/zubair/src/vhdl/final_hdpp_16/crl.vhd

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.all;

ENTITY control_decode IS PORT (  
  CCKi: IN std_logic;
  OP_CODEi: IN std_logic_vector(2 downto 0);
  INST_DECODEi: IN std_logic_vector(4 downto 0);
  RSTi: IN std_logic;
  OP_CODE_Ro: OUT std_logic_vector(2 downto 0);
  I3o: BUFFER std_logic;
  DPSETo: OUT std_logic;
  DPCLRo: OUT std_logic;
  DPACKo: BUFFER std_logic;
  STRSETo: OUT std_logic;
  STRCLRo: OUT std_logic;
  SLDSETo: OUT std_logic;
  SLDCLRo: OUT std_logic;
```
SCKENo: OUT std_logic;
CLKENo: OUT std_logic;
RAWENo: OUT std_logic;
AIENo: BUFFER std_logic
);
END control_decode;

ARCHITECTURE arch_control_decode of control_decode IS
signal CP_DECODE: integer range 0 to 31;
signal CP_WORD_PRE: std_logic_vector(6 downto 0);
signal CP_WORD: std_logic_vector(6 downto 0);
signal CLK_ENABLES: std_logic_vector(3 downto 0);
signal I3_PRE: std_logic;
BEGIN
CP_DECODE <= to_integer(INST_DECODi);

With CP_DECODE Select
CP_WORD_PRE <= "0000001" when 1, --DPACK A17
 "0000010" when 2, --DPSET A18
 "0000100" when 3, --DPCLR A18, A17
 "0001000" when 4, --STRSET A19
 "0010000" when 5, --STRCLR A19, A17
 "0100000" when 6, --SLDSET A19, A18
 "1000000" when 7, --SLDCLR A19, A18, A17
 "0000000" when others;

CP_WORD <= CP_WORD_PRE when OP_CODEi="001" else "0000000";

CLK_ENABLES <= "0000" when (OP_CODEi="001" and CP_DECODi=10) else
 "1111" when (OP_CODEi="001" and CP_DECODi=11) else
 CLK_ENABLES;

I3_PRE <= '1' when (OP_CODEi="001" and CP_DECODi=8) else
 '0' when (OP_CODEi="001" and CP_DECODi=9) else
 I3_PRE;

control_path_process: process (RSTi, CCKi)
begin
if RSTi = '1' then
------------------------
--Asynchronous Reset
------------------------
  --positively asserted
DPSETo <= '0';
--negatively asserted
SCKEnO <= '1';
RAWEnO <= '1';
CLKEnO <= '1';
AIEnO <= '1';

elsif (CCKi'event and CCKi='1') then

--Registered Signals

--Assign 7 signals to each bit of CP_WORD;
DPACKo <= CP_WORD(0);
DPSETo <= CP_WORD(1);
DPCLRo <= CP_WORD(2);
STRSETo <= CP_WORD(3);
STRCLRo <= CP_WORD(4);
SLDSETo <= CP_WORD(5);
SLDCLRo <= CP_WORD(6);

SCKEnO <= CLK_ENABLes(0);
RAWEnO <= CLK_ENABLes(1);
CLKEnO <= CLK_ENABLes(2);
AIEnO <= CLK_ENABLes(3);

I3O    <= '1';
OP_CODE_Ro <= OP_CODEi;
end if;
end process;

END arch_control_decode;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
PACKAGE my_control_dec_pkg IS

COMPONENT control_decode PORT (  
  CCKi: IN std_logic;
  OP_CODEi: IN std_logic_vector(2 downto 0);
  INST DECODEi: IN std_logic_vector(4 downto 0);
  RSTi: IN std_logic;
  OP_CODE_Ro: OUT std_logic_vector(2 downto 0);
);
I3o: BUFFER std_logic;
DPSETo: OUT std_logic;
DPCLR0: OUT std_logic;
DPACKo: BUFFER std_logic;
STRSETo: OUT std_logic;
STRCLR0: OUT std_logic;
SLDSETo: OUT std_logic;
SLDCLR0: OUT std_logic;
SCKENo: OUT std_logic;
CLKENo: OUT std_logic;
RAWENo: OUT std_logic;
AIENo: BUFFER std_logic
);
END COMPONENT;
END my_control_dec_pkg;

B.1.3 Data-Path Handshaking

Location on COLOB:
/homes/zubair/src/vhdl/final.hdpp.16/dp.hs.vhd

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.std_arith.all;

ENTITY dp_handshaking IS PORT ( 
nPCKi: in std_logic;
nPCK2i: in std_logic;
Xi: in std_logic;
Hi: in std_logic;
DPSETi: in std_logic;
DPACKi: in std_logic;
DPCLRi: in std_logic;
HQo: buffer std_logic;
HDo: buffer std_logic;
ARo: buffer std_logic;
SRo: buffer std_logic;
DPATNo: buffer std_logic;
XAo: buffer std_logic;
HAo: buffer std_logic;
XFRo: buffer std_logic;
DPERRo: buffer std_logic
);
END dp_handshaking;
ARCHITECTURE arch_dp_handshaking of dp_handshaking IS
  signal ARI, SRI: std_logic;
  signal state: std_logic_vector(3 downto 0);
  constant Idle: std_logic_vector(3 downto 0) := "0000";
  constant AckFrame: std_logic_vector(3 downto 0) := "1000";
  constant Exchange: std_logic_vector(3 downto 0) := "1011";
  constant AckLine: std_logic_vector(3 downto 0) := "1101";
  constant Transfer: std_logic_vector(3 downto 0) := "1001";
  constant Finish: std_logic_vector(3 downto 0) := "1010";

begin
  XAo <= state(3);
  H Ao <= state(2);
  DPATNo <= state(1);
  XFRo <= state(0);

  ARI <= '1' when DPSETi='1' else '0' when DPCLRi='1' else ARI;
  SRI <= '1' when DPACKi='1' else '0' when DPATNo='0' else SRI;
  HDo <= Hi or not(XFRo) or not(Xi);

  data_path_process: process
  begin
    wait until nPCKi='1';

  end process;

  -- Registered Signals

  HQo <= HDo;
  ARo <= ARI;
  SRo <= SRI;

  -- DataPath Handshaking State Machine
  case state is
    begin
      -- Idle


when Idle =>
    if not(ARo) = '1' then
        state <= Idle;
        DPERRo <= '0';
    elsif Hi = '1' then
        state <= Idle;
        DPERRo <= '1';
    elsif Xi = '1' then
        state <= AckFrame;
        DPERRo <= DPERRo;
    else
        state <= Idle;
        DPERRo <= DPERRo;
    end if;

--------
--AckFrame
--------

when AckFrame =>
    if not(ARo) = '1' then
        state <= Idle;
        DPERRo <= '0';
    elsif (not(Xi) and Hi) = '1' then
        state <= Idle;
        DPERRo <= '1';
    elsif not(Xi) = '1' then
        state <= Idle;
        DPERRo <= DPERRo;
    elsif (Hi and not(SRo)) = '1' then
        state <= Exchange;
        DPERRo <= DPERRo;
    else
        state <= AckFrame;
        DPERRo <= DPERRo;
    end if;

--------
--Exchange
--------

when Exchange =>
    if not(ARo) = '1' then
        state <= Idle;
        DPERRo <= '0';
    elsif not(Xi) = '1' then
        state <= Idle;
        DPERRo <= '1';
    elsif not(Hi) = '1' then
state <= Idle;
DPERRo <= '1';
elsif SRo = '1' then
    state <= AckLine;
    DPERRo <= DPERRo;
else
    state <= Exchange;
    DPERRo <= DPERRo;
end if;

--- AckLine

when AckLine =>
    if not(ARo) = '1' then
        state <= Idle;
        DPERRo <= '0';
    elsif not(Xi) = '1' then
        state <= Idle;
        DPERRo <= '1';
    elsif not(Hi) = '1' then
        state <= Transfer;
        DPERRo <= DPERRo;
    else
        state <= AckLine;
        DPERRo <= DPERRo;
    end if;

--- Transfer

when Transfer =>
    if not(ARo) = '1' then
        state <= Idle;
        DPERRo <= '0';
    elsif (not(Xi) and Hi) = '1' then
        state <= Idle;
        DPERRo <= '1';
    elsif (Hi and not(SRo)) = '1' then
        state <= Exchange;
        DPERRo <= DPERRo;
    elsif (not(Xi) and not(SRo)) = '1' then
        state <= Finish;
        DPERRo <= DPERRo;
    else
        state <= Transfer;
        DPERRo <= DPERRo;
end if;

----------
--Finish
----------

when Finish =>
  if not(ARo) = '1' then
    state <= Idle;
    DPERRo <= '0';
  elsif (Xi or Hi) = '1' then
    state <= Idle;
    DPERRo <= '1';
  elsif SRo = '1' then
    state <= Idle;
    DPERRo <= DPERRo;
  else
    state <= Finish;
    DPERRo <= DPERRo;
  end if;
when others =>
  if not(ARo) = '1' then
    state <= Idle;
    DPERRo <= '0';
  else
    state <= Idle;
    DPERRo <= '0';
  end if;
end case;
end process;

----------
--End Data Path Process
----------
end arch_dp_handshaking;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
PACKAGE my_dp_handshaking_pkg IS
  COMPONENT dp_handshaking PORT (  
    nPCKi: in std_logic;
    nPCK2i: in std_logic;
    Xi: in std_logic;
    Hi: in std_logic;
    DPSETi: in std_logic;
    DPACKi: in std_logic;
    DPCLRi: in std_logic;

    state: out std_logic;
    DPERRo: out std_logic;
  );
B.1. VHDL

    HQo:    buffer std_logic;
    HDo:    buffer std_logic;
    ARo:    buffer std_logic;
    SRo:    buffer std_logic;
    DPATNo: buffer std_logic;
    XAo:    buffer std_logic;
    HAo:    buffer std_logic;
    XFRo:   buffer std_logic;
    DPERRo: buffer std_logic
    --
    DPERRo: buffer std_logic
    --
    DP_DEBUGo: buffer std_logic
    
    END COMPONENT;
END my_dp_handshaking_pkg;

B.1.4 Generation of Clocks

Location on COLOB:
/homes/zubair/src/vhdl/final.hdpp.16/clocks.vhd

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY hdpp_clocks IS PORT (    
    nPCKi:    IN std_logic;
    nPCK2i:   IN std_logic;
    STRSETi:  IN std_logic;
    STRCLRi:  IN std_logic;
    SLDSETi:  IN std_logic;
    SLDCLRi:  IN std_logic;
    ARi:      IN std_logic;
    SRi:      IN std_logic;
    HDi:      IN std_logic;
    HQi:      IN std_logic;
    SLDo:     BUFFER std_logic;
    STRo:     BUFFER std_logic;
    SCK1o:    BUFFER std_logic;
    SCK2o:    BUFFER std_logic
    
    );
END hdpp_clocks;

ARCHITECTURE arch_hdpp_clocks of hdpp_clocks IS
    signal SCK1_SET:        std_logic;
    signal SCK2_SET:        std_logic;
    signal SLDp,STRp:       std_logic;
TYPE sck_states IS (Idle_0, SCK1_ON, SCK2_ON);
SIGNAL sck_state: sck_states;

BEGIN

-- SLD and STR signal derived from SET/CLR

SLDp <= '1' when SLDSETi='1' else '0' when SLDCLRi='1' else SLDp;
STRp <= '1' when STRSETi='1' else '0' when STRCLRi='1' else STRp;
SLDo <= not(SLDp);
STRo <= not(STRp);

-- Glitch-Free SCK1 and SCK2 (25% duty cycle)

SCK_process_set: process
begin
wait until nPCK2i='1';
case sck_state is
when Idle_0 =>
if HDi='0' then
  SCK2_SET <= '1';
  SCK1_SET <= '0';
  sck_state <= SCK1_ON;
else
  SCK2_SET <= '0';
  SCK1_SET <= '0';
  sck_state <= Idle_0;
end if;
when SCK2_ON =>
if HDi='1' then
  SCK2_SET <= '0';
  SCK1_SET <= '0';
  sck_state <= Idle_0;
else
  SCK2_SET <= '1';
  SCK1_SET <= '0';
  sck_state <= SCK1_ON;
end if;
end case;
end process;

end;
when SCK1_ON =>
    SCK2_SET <= '0';
    SCK1_SET <= '1';
    if HDi='1' then
        sck_state <= Idle_0;
    else
        sck_state <= SCK2_ON;
    end if;
when others =>
    SCK2_SET <= '0';
    SCK1_SET <= '0';
    sck_state <= Idle_0;
end case;
end process;

SCK1o <= not(SLDp) and (nPCK2i or HQi or not(SCK2_SET));
SCK2o <= nPCK2i or HDi or SCK2_SET;

END arch_hdpp_clocks;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
PACKAGE my_hdpp_clk_pkg IS
    COMPONENT hdpp_clocks PORT ( 
        nPCKi: IN std_logic;
        nPCK2i: IN std_logic;
        STRSETi: IN std_logic;
        STRCLRi: IN std_logic;
        SLDSETi: IN std_logic;
        SLDCLRi: IN std_logic;
        ARi: IN std_logic;
        SRi: IN std_logic;
        HDi: IN std_logic;
        HQi: IN std_logic;
        SLD0: BUFFER std_logic;
        STR0: BUFFER std_logic;
        SCK1o: BUFFER std_logic;
        SCK2o: BUFFER std_logic
    );
END COMPONENT;

END my_hdpp_clk_pkg;
B.2 Programming Clock Generation Scheme

Location on COLOB:

/home/zubair/src/vhdl/hdpp-clocks/clocks.vhd

Library ieee;
Use ieee.std_logic_1164.all;

--This VHDL is for the 374i device on HDPP-16-002.
--It is used to generate the clock signals for the HDPP chips.
--The duty cycles and relative delays of this clocking scheme are inverted
--from what they should actual be doing. This is due to the inverting clock
--drivers used,

Entity clocks IS PORT (
  icrystal: IN std_logic; --Crystal
  iclock: IN std_logic; --Input Clock wrapped around
  idelay1: IN std_logic; --Clock for 50% Duty-Cycle Latch
  idelay2: IN std_logic; --Clock for One-Shot
  idelay4: IN std_logic; --Clock to Set Duty-Cycle of CLK1

  idelay_params: IN std_logic_vector(7 downto 0);
  idelay_select: IN std_logic_vector(1 downto 0);
  iset_delay: IN std_logic;

  slatch_q: BUFFER std_logic;
  ocrystal: BUFFER std_logic;
  oclk0: OUT std_logic;
  oclk1: OUT std_logic;
  onck0: OUT std_logic;
  onck1: OUT std_logic;

  odelay1: BUFFER std_logic_vector(7 downto 0);
  odelay2: BUFFER std_logic_vector(7 downto 0);
  odelay3: BUFFER std_logic_vector(7 downto 0);
  odelay4: BUFFER std_logic_vector(7 downto 0)
);

ATTRIBUTE pin_numbers of clocks: entity is
"icrystal:11 " &
"iclock:15 " &
"idelay1:61 " &
"idelay2:65 " &
"idelay4:36 " &
"iDelay_Params(0):3 " &
"iDelay_Params(1):4 " &
"iDelay_Params(2):5 " &
"iDelay_Params(3):6 " &
"iDelay_Params(4):7 " &
"iDelay_Params(5):8 " &
"iDelay_Params(6):9 " &
"iDelay_Params(7):10 " &
"iDelay_Select(0):18 " &
"iDelay_Select(1):20 " &
"iSet_Delay:22 " &
"oCrystal:16 " &
"oCLK0:72 " &
"oCLK1:68 " &
"oNCK0:58 " &
"oNCK1:55 " &
"oDelay1(0):28 " &
"oDelay1(1):29 " &
"oDelay1(2):30 " &
"oDelay1(3):31 " &
"oDelay1(4):32 " &
"oDelay1(5):33 " &
"oDelay1(6):34 " &
"oDelay1(7):35 " &
"oDelay2(0):41 " &
"oDelay2(1):42 " &
"oDelay2(2):43 " &
"oDelay2(3):44 " &
"oDelay2(4):45 " &
"oDelay2(5):46 " &
"oDelay2(6):47 " &
"oDelay2(7):48 " &
"sLatch_Q:53 " &
"oDelay3(0):78 " &
"oDelay3(1):79 " &
"oDelay3(2):80 " &
"oDelay3(3):81 " &
"oDelay3(4):82 " &
"oDelay3(5):83 " &
"oDelay3(6):84 " &
"oDelay3(7):85 " &
"oDelay4(0):91 " &
"oDelay4(1):92 " &
"oDelay4(2):93 " &
"oDelay4(3):94 " &
"oDelay4(4):95 " &
"oDelay4(5):96 " &
"oDelay4(6):97 " &
"oDelay4(7):98 ";

End clocks;

USE WORK.std_arith.ALL;
USE WORK.int_math.ALL;

ARCHITECTURE behavior OF clocks IS

signal sOneShot: std_logic;
signal sLatch_R: std_logic;
signal Delay1: std_logic_vector(7 downto 0);
signal Delay2: std_logic_vector(7 downto 0);
signal Delay3: std_logic_vector(7 downto 0);
signal Delay4: std_logic_vector(7 downto 0);

BEGIN

--Delay lines 1 and 4 are swapped due to a wiring mistake
oDelay1 <= Delay4;
oDelay2 <= Delay2;
oDelay3 <= Delay3;
oDelay4 <= Delay1;

--THIS WAS USED TO PROGRAM THE DELAY LINES THROUGH THE DIP SWITCH
--CONNECTOR BOARD

--PROCESS
--BEGIN
--WAIT UNTIL iClock='1';
--IF iSet_Delay='1' THEN
--CASE iDelay_Select IS
--   WHEN "00" => Delay1 <= iDelay_Params;
--   WHEN "01" => Delay2 <= iDelay_Params;
--   WHEN "10" => Delay3 <= iDelay_Params;
--   WHEN OTHERS => Delay4 <= iDelay_Params;
--END CASE;
--ELSE
-- Delay1 <= Delay1;
-- Delay2 <= Delay2;
-- Delay3 <= Delay3;
-- Delay4 <= Delay4;
--END IF;
--END PROCESS;

--Pins 1 and 2 of H14 (the 14-pin surface mount 2mm connector)
--choose the clock scheme
--00 --> 10MHZ
--10 or 01 --> 15MHZ
--11 --> 20MHZ

WITH iDelay_Params(1 downto 0) SELECT
  Delay1 <= "00111100" when "00",
            "00001100" when "11",
            "00011000" when others;

WITH iDelay_Params(1 downto 0) SELECT
  Delay2 <= "01110000" when "00",
            "00011000" when "11",
            "00110101" when others;

WITH iDelay_Params(1 downto 0) SELECT
  Delay3 <= "00110110" when "00",
            "00001100" when "11",
            "00011000" when others;

WITH iDelay_Params(1 downto 0) SELECT
  Delay4 <= "10001111" when "00",
            "00101000" when "11",
            "00111110" when others;

--10Mhz Scheme
-- Delay1 <= "00111100";
-- Delay2 <= "01110000";
-- Delay3 <= "00110110";
-- Delay4 <= "10001111";

--20Mhz Scheme
-- Delay1 <= "00001100";
-- Delay2 <= "00011000";
-- Delay3 <= "00001100";
-- Delay4 <= "00101000";

--15Mhz Scheme
-- Delay1 <= "00011000";
-- Delay2 <= "00110101";
-- Delay3 <= "00011000";
-- Delay4 <= "00111100";

--16Mhz Scheme
-- Delay1 <= "00010010";
-- Delay2 <= "00101110";
-- Delay3 <= "00010110";
-- Delay4 <= "00111001";

--18Mhz Scheme
-- Delay1 <= "00010011";
-- Delay2 <= "00100010";
-- Delay3 <= "00010001";
-- Delay4 <= "00110010";

PROCESS (sLatch_R,iDelay1)
BEGIN
IF sLatch_R='1' THEN
  sLatch_Q <= '0';
ELSIF iDelay1'event AND iDelay1='1' THEN
  sLatch_Q <= '1';
END IF;
END PROCESS;

PROCESS (sOneShot,iDelay2)
BEGIN
IF sOneShot='1' THEN
  sOneShot <= '0';
ELSIF iDelay2'event AND iDelay2='1' THEN
  sOneShot <= '1';
END IF;
END PROCESS;

cCrystal <= iCrystal;
sLatch_R <= sOneShot;
ocLK0 <= sLatch_Q;
ocLK1 <= iDelay4 or not(iClock);
ocNCK0 <= sLatch_Q;
ocNCK1 <= sLatch_Q;

END behavior;

B.3 Schematics

The board schematics, library files, and PC board layout for both the connector board (HDPP-CON) and the 16 chip board (HDPP-16-002) are located on COLOB in ACCEL binary format.

```
/home/zubair/hdpp/hdpp-final-002/
    hdpp-con.sch
    hdpp-con.pcb
    hdpp-16-002.sch
    hdpp-16-002.pcb
    hdpp-final.lib
```

Schematics for HDPP-CON are shown in Figures B.1 - B.4. Schematics for HDPP-16-002 are shown in Figures B.5 - B.11.
Figure B.1: Controller Interface.
Figure B.2: Data Path Receiver Interface.
Figure B.3: Data Path Transmitter Interface.
Figure B.5: Connectors.
Figure B.6: Clock Drivers.
Figure B.7: CPLD and Data Path Handshaking.
Figure B.8: HDPP Devices Northwest Quadrant.
Figure B.9: HDPP Devices Northeast Quadrant.
Figure B.10: HDPP Devices Southwest Quadrant.
Figure B.11: HDPP Devices Southeast Quadrant.
Appendix C

Software System

The source code for the software system resides on COLOB in:

\[ /\text{homes/zubair/src/xapp-4} \]

Hall's Master's Thesis documents the structure of the \textit{xapp} directory.

C.1 Writing Applications

An application development manual is located in:

\[ /\text{homes/zubair/hdpp/Software/manual.ps} \]

C.1.1 Sequences

Some sample sequences are shown in the following directories:

\[ /\text{homes/zubair/src/xapp-4/hdpphw/rtdemo/} \]
\[ /\text{homes/zubair/src/xapp-4/hdpphw/test/} \]

Some applications make use of pre-defined sequences defined in another directory:

\[ /\text{homes/zubair/src/xapp-4/libxapp/seqlib/} \]
\begin{verbatim}
edge.cc
median.cc
smseg.cc
laplace.cc
\end{verbatim}

C.1.2 Pre-Sequences

Some good examples of applications that use pre-sequences are:
/homes/zubair/src/xapp-4/hdp phw/rtdemo/rtwriteC.cc
/homes/zubair/src/xapp-4/hdp phw/test/addtest.cc
/homes/zubair/src/xapp-4/libxapp/hdp pCG/hdppsys.cc

This is an example of a pre-sequence:

```c
presequence preseq;
hdpp::vector v;

v += hdpp::inst::pe(hdpp::hdpp_zero, 0, boole3_set, 0, hdpp::lb);
v += hdpp::inst::pe(hdpp::hdpp_write, 1, boole3_and, 0, hdpp::la
    | hdpp::lb);
v += hdpp::inst::pe(hdpp::hdpp_write, 2, boole3_and, 0, hdpp::ld);
v += hdpp::inst::pe(hdpp::hdpp_write, 3, boole3_and, 0, hdpp::ld);
v += hdpp::inst::pe(hdpp::hdpp_write, 4, boole3_and, 0, hdpp::ld);
v += hdpp::inst::pe(hdpp::hdpp_write, 5, boole3_and, 0, hdpp::ld);
v += hdpp::inst::pe(hdpp::hdpp_write, 6, boole3_and, 0, hdpp::ld);
v += hdpp::inst::pe(hdpp::hdpp_write, 7, boole3_and, 0, hdpp::ld);
v += hdpp::inst::pe(hdpp::hdpp_write, 8, boole3_and, 0, hdpp::ld);

v.optimize();
pres eq += v;
seqAdd += preseq.assemble();
```

The arguments to the `hdpp::inst::pe` command are:

1. op-code
2. address bit
3. function generator
4. interconnect communication
5. latch load

The actual hexadecimal values can be used for the function generator, however, pre-defined constants are defined in:

```
/homes/zubair/src/xapp-4/libxapp/include/boolefun.h
```

C.1.3 Compilation

Each time a new application is created, a new entry needs to be added to the Makefile to facilitate easy compilation. Since the makefiles are automatically generated by the autoconf utility when the whole system is compiled, the Makefile.in file needs to be altered as well. For temporary applications it is sufficient to modify the Makefile. When the whole software system is recompiled the temporary entries will be overwritten by the automatically
generated Makefile. Appendix C in Hall’s thesis goes into more detail about software maintenance.
To generate a new Makefile, modify the appropriate Makefile.in file. Then:

```
cd /homes/zubair/src/xapp-4
```

and run:

```
./configure-sh/configure-sparc-CC_0_SCCS
```

To remake the whole system:

```
cd /homes/zubair/src/xapp-4
make distclean
./configure-sh/configure-sunos5_CC_0_SCCS
```

```
cd /homes/zubair/src/xapp-4/libxapp
make
```

```
cd /homes/zubair/src/xapp-4/hdpphw
make
```

C.1.4 Revision Control

The GNU Revision Control Software (RCS) is utilized extensively throughout the software system. Some useful commands are included below:

To add a new file to RCS: `rcs -i filename`
To check out a file of RCS: `co -l filename`
To check in a file to RCS: `ci -u filename`
To review RCS log: `rlog filename`
To edit an old RCS file: `co -l -rev# filename`
To compare RCS files: `rcsdiff -rev# filename`

C.2 Testing PPIP Chips

Functional tests were performed using the median filtering. The command used was:

```
cd /homes/zubair/src/xapp-4/hdpphw/test
```

```
./median-16 -v -n 4 -x 16 -R -H -R -V -R -S 0 -R -S 255 -R -i /homes/zubair/src/xapp-4/images/misc/san_fran256.pgm -R -R -S 0 -S 255 -S 0 -R -d /homes/zubair/tmp/good/median-diff.pgm -o /homes/zubair/tmp/good/median-out.pgm | &
```

tee /homes/zubair/tmp/good/median-out.txt
Comprehensive memory tests were performed by executing the following command:

```
cd /homes/zubair/src/xapp-4/hdpphw/test

./memtest-16 -v -n 4 -x 16 -R -H -R -V -R -S 0 -R -S 255 -R -i /homes/zubair/src/xapp-4/images/misc/san_fran256.pgm -R -R -S 0 -S 255 -S 0 -R -o /homes/zubair/tmp/mem-4.pgm | &
tee /homes/zubair/tmp/mem-4.txt
```

The \texttt{-n} parameter specifies the refresh interval.

### C.3 Changing the Refresh Interval

The default refresh interval is set in:

```
xapp-4/libxapp/mit20sys/hardware.cc
```

A number of applications have interval as command-line parameters. To set the refresh interval for a particular application use the following line:

```
hwSystem sys(interval);
```

instead of:

```
hwSystem sys;
```

### C.4 Software Patch for Previous Generation Test Board

To reinstate the software workaround for \texttt{t}:

```
cd /homes/zubair/src/xapp-4/libxapp/hdpp
co -r4.1 hdppinst.y
```

Then make distclean, make, etc...

To get the code without the workaround:

```
cd /homes/zubair/src/xapp-4/libxapp/hdpp
co hdppinst.y
```

Then make distclean, make, etc...

### C.5 Expanding to Three Cameras

The following files govern how the Controller deals with Image I/O:

```
/homes/zubair/src/xapp-4/libxapp/hdppCG/hdppdpio.cc
/homes/zubair/src/xapp-4/libxapp/hdppCG/hdppsosyv.cc
```

The following line specifies 8-bit image transfers

```
for (unsigned plane = 0; plane < 8; plane++)
```

The 8 could be replaced with a 24 so that 3 images worth of data get transferred at a time.
C.6 Removing Bad Memory Cells

This is the file that specifies the available memory cells to be allocated by compiler:

```
/libxapp/hdpp/hdpparch.cc
```

To remove bad memory cells from the allocatable table of memory cells simply replace:

```
hfield hf(arch::array::FreeField());
allocContext aC(hf);
```

with something like:

```
hfield hf1(arch::array::FreeField());
hfield hf2
    = hf1.before(5)  // cells 0-4
    + hf1.at(6,10)  // cells 6-15
    + hf1.at(20);  // cells 20-127
allocContext aC(hf2);
```

To remove the cells from the allocatable table for all applications and preclude the need to explicitly state which cells to remove in each applications, remove the cells from:

```
/homes/zubair/src/xapp-4/libxapp/hdpp/hdpparch.cc
```

The bad cells can be removed from this piece of code:

```
const hdpp::hfield& hdpp::hfield::AllCells() {
    static const cellPos hardv[] = {
        cellPos( 0), cellPos( 1), cellPos( 2), cellPos( 3),
        cellPos( 4), cellPos( 5), cellPos( 6), cellPos( 7),
        .
        .
        cellPos(120), cellPos(121), cellPos(122), cellPos(123),
        cellPos(124), cellPos(125), cellPos(126), cellPos(127),
        NullCellPos()  
    };
}
```

In order for the changes to take effect the entire system needs to be recompiled.

C.7 Generating the Bit-Code for a New Controller

The bit-code for a particular sequence can be sent to standard output from an application simply by printing the sequence. An example is shown in:

```
/homes/zubair/src/xapp-4/hdpphw/test/addtest.cc
```
The lines of code that print out the bit-code are:

```cpp
    cerr << "seqAdd: \n" << flush;
    cerr << seqAdd;
    cerr << "\n" << flush
```

This bit-code could be simply stored in an SRAM and sent to the PE Array. Image transfer and memory refreshes must be handled separately.

Pre-sequence code for refreshes and for exchange vectors are included in:

```
    /homes/zubair/src/xapp-4/libxapp/hdppCG/hdppsys.cc
```

### C.8 Demonstrations

In order to run flicker-free demos, it is necessary to use the host computer BIRD in single-user mode:

1. Halt the computer:
   ```
   wheel
   sync; sync; sync;
   halt
   ```

2. Turn on the VMEbus chassis

3. Boot in single user mode. From 40-character Sun screen type:
   ```
   b -s
   ```

4. Enter password (when system reboots it’ll ask for root password for BIRD)

5. Run tcsh (when prompt comes up):
   ```
   /usr/local/bin/tcsh
   ```

6. Load aliases:
   ```
   cd /var/apps
   source aliases
   ```

7. Reset controller:
   ```
   ./reset-gold
   ```

8. Run applications:
   ```
   edge, flow, smseg, motion, median
   ```

The command switch can be used to switch between an application and the unprocessed input image. The command exit can be used to kill an application that is currently running.
THESIS PROCESSING SLIP

FIXED FIELD: ill. name

index biblio

COPYES: Archives Aero Dewey Eng Hum

Lindgren Music Rotch Science

TITLE VARIES: □

NAME VARIES: □

IMPRINT: COPYRIGHT

COLLATION: 146 p

ADD: DEGREE: □ DEPT.: □

SUPERVISORS: 

NOTES: 

cat': date: page: 

DEPT: E.E. F41

YEAR: 1999 DEGREE: M.Eng.

NAME: TALIB, Zubair Aman