PIPELINING – AN APPROACH FOR MACHINE VISION

D. J. Foster
Balliol College.

A thesis submitted for the degree of
Doctor of Philosophy
in the University of Oxford.

Department of Engineering Science.

Michaelmas 1987.

Abstract

PIPELINING – AN APPROACH FOR MACHINE VISION

D. J. Foster, Balliol College

A thesis submitted for the degree of Doctor of Philosophy
in the University of Oxford.
Michaelmas 1987.

Much effort has been spent over the last decade in producing so called "Machine Vision" systems for use in robotics, automated inspection, assembly and numerous other fields. Because of the large amount of data involved in an image (typically $\frac{1}{4}$ MByte) and the complexity of many algorithms used, the processing times required have been far in excess of real time on a VAX-class serial processor. We review a number of image understanding algorithms that compute a globally defined "state", and show that they may be computed using simple local operations that are suited to parallel implementation. In recent years, many massively parallel machines have been designed to apply local operations rapidly across an image. We review several vision machines. We develop an algebraic analysis of the performance of a vision machine and show that, contrary to the commonly-held belief, the time taken to relay images between serial streams can exceed by far the time spent processing. We proceed to investigate the roles that a variety of pipelining techniques might play. We then present three pipelined designs for vision, one of which has been built. This is a parallel pipelined bit slice convolution processor, capable of operating at video rates. This design is examined in detail, and its performance analysed in relation to the theoretical framework of the preceeding chapters. The construction and debugging of the device, which is now operational in its hardware is detailed.

Acknowledgements

I would like to express thanks to everyone at Oxford who has helped me in the writing of this thesis, in particular to my supervisor, Mike Brady, who has kept me on the right track for the last two years, and provided much inspiration. Thanks also to David Witt, Josef Kittler and Arthur Dexter, who supervised me in my first year; also to Tony, Vaughan and Greg for many conversations, which kept my sense of humour alive.

I am also extremely grateful to IBM U.K. Laboratories Ltd. for their support of this CASE studentship, for the equipment and financial help that they have provided. Particular thanks must be made to Ivan Symonds and Steve O'Connell, who managed the project, to Pat O'Neill, Graham Caulfield and the staff of the U.K. Science Center, who provided invaluable advice, and to Nigel Ansell, who entered the PPC design onto IBM's design system.

Finally, none of this would have been possible without the support of my parents and family who provided advice and encouragement throughout. Thanks also to the members of OULRC, OUBC, Leander Club and Balliol College Boat Clubs, who have provided me with endless diversions over my years in Oxford.

Contents

1 INTRODUCTION 7

2 VISION ALGORITHMS 14
- 2.1 Introduction . . . 14
- 2.2 Low Level Algorithms . . . 15
  - 2.2.1 Signal Processing Approach . . . 17
- 2.3 Why Convolution . . . 21
  - 2.3.1 Benefits From Parallelism . . . 22
- 2.4 State of the Art Vision . . . 23
  - 2.4.1 The Geometry Viewpoint . . . 24
  - 2.4.2 The Need For Constraints . . . 24
  - 2.4.3 Regularisation . . . 28
  - 2.4.4 Relaxation . . . 29
  - 2.4.5 Global State From Local Interaction . . . 30
- 2.5 Intermediate Level Algorithms . . . 39
  - 2.5.1 Shape From Shading . . . 39
  - 2.5.2 Edge Detection . . . 44
  - 2.5.3 Lightness Computation . . . 50
  - 2.5.4 Optic Flow . . . 54
  - 2.5.5 Surface Reconstruction . . . 59
  - 2.5.6 Stereo . . . 61
  - 2.5.7 Stochastic Relaxation . . . 67
- 2.6 Chapter Summary . . . 73

3 CURRENT ARRAY PROCESSORS FOR VISION 74
- 3.1 Introduction . . . 74
- 3.2 Three Array Computers . . . 77
  - 3.2.1 UCL CLIP . . . 77
  - 3.2.2 ICL/AMT DAP . . . 78
  - 3.2.3 Goodyear MPP . . . 80
- 3.3 Three Unconventional Vision Processors. . . . 82
  - 3.3.1 The Cytocomputer. . . . 82
  - 3.3.2 NBS PIPE Processor . . . 85
  - 3.3.3 The Connection Machine. . . . 87
- 3.4 ARRAY PROCESSORS . . . 88

4 AN ALGEBRAIC PERFORMANCE MODEL 90
- 4.1 Input/Output Methods . . . 90
  - 4.1.1 Pixel-serial . . . 95
  - 4.1.2 Column-Parallel . . . 96
  - 4.1.3 Column-Parallel I/O overlapped with Processing . . . 96
  - 4.1.4 Image-parallel . . . 97
  - 4.1.5 General Case . . . 97
  - 4.1.6 Evaluation . . . 97

5 PIPELINING 100
5.1 Pipelining - A Definition 100
5.1.1 General Pipelines 102
5.2 Speed and Efficiency 106
5.3 Performance Analysis 109
5.3.1 Pipeline Speed 109
5.3.2 Pipeline Cost 111
5.4 Pipeline Problems 115
5.5 Pipeline vs. Array Computers 116
5.5.1 Classification of existing pipeline architectures. 120
5.6 Pipelining as a Technique. 124
5.7 Pipelining to Improve Memory Bandwidth 125
5.7.1 Serpentine Memory 125
5.7.2 Memory Interleaving 126
5.8 Multi-Dimensional Pipelining 132
5.9 Direct Camera Interface 135

6 DESIGN VIGNETTES 137
6.1 Introduction 137
6.2 A Serpentine Array 138
6.3 A Dynamic Memory Array 142
6.4 A Parallel-Pipelined Convolver 145

7 DESIGN OF A PARALLEL PIPELINED CONVOLVER 151
7.1 System Overview 151
7.2 PC Card 157
7.2.1 Clock Generator 157
7.2.2 PC Interface 161
7.3 Analogue Interface 163
7.3.1 Data Interface 163
7.3.2 Internal Synchronisation Signal Generation 167
7.4 Planar 1 Card 169
7.4.1 General 169
7.4.2 Program Controller 169
7.4.3 Memory Mapping 174
7.4.4 Convolver 176
7.5 Planar 2 Card 180
7.5.1 General 180
7.5.2 Memory Address Generator 180
7.5.3 Main Memory 184
7.6 System Operation 187
7.6.1 Input/Output to TV or Camera 187
7.6.2 Input or Output to Controlling PC. 189
7.6.3 Convolution using a 3x3 Window. 190
7.7 Design Summary 193
7.8 APPENDIX A 194
7.8.1 Pipeline Register Bits 194

8 PARALLEL-PIPELINED CONVOLVER – THEORETICAL FRAMEWORK 204

9 CONCLUSIONS 209
9.1 Current Work 209
9.2 Future Work 211

List of Figures

1.1 The Parallel Pipelined Convolver 13

2.1 Simple Directional Derivatives 16
2.2 Sobel and Canny Edge Finders-Original Picture 18
2.3 Sobel and Canny Edge Finders-Results of Sobel 19
2.4 Sobel and Canny Edge Finders-Results of Canny 20
2.5 Huffman and Clowes' Shape algorithm 26
2.6 Convex and Non-Convex Parameter Spaces 31
2.7 Hildreth's Optic Flow Algorithm 33
2.8 Horn and Schunck's Optic Flow Algorithm 34
2.9 Canny's Edge Finder-Original Picture 35
2.10 Canny's Edge Finder-Results 36
2.11 Ikeuchi and Horn's Shape from Shading Algorithm 37
2.12 Terzopoulos' Surface Reconstruction Algorithm 38
2.13 Ikeuchi and Horn's use of the Stereographic Plane 40
2.14 A Zero Crossing Edge 45
2.15 Laplacian of Gaussian Filter 46
2.16 Results from Laplacian of Gaussian Filter 47
2.17 Canny's Non-Maximum Supression 51
2.18 Blake's Lightness Algorithm 54
2.19 The Aperture Problem 56
2.20 Two Images Separated by a Baseline 62
2.21 Stereo Disparity Matrix 64
2.22 Poggio and Drumheller's Stereo Algorithm 66
2.23 Energy Potentials in Geman and Geman 70
2.24 Geman's Definition of U(l). 71
2.25 Geman's Definition of a Line Site 71
2.26 Geman and Geman's Reconstruction Algorithm 72

3.1 The CLIP4 Processor 79
3.2 The DAP Processor 80
3.3 The MPP Processor 82
3.4 The Cytocomputer 84
3.5 The PIPE Processor 86
3.6 The Connection Machine 88

4.1 Two Components to I/O Array Time 93
4.2 Row parallel camera output 94

5.1 Functional Structure of a Modern Computer 102
5.2 A Linear Pipeline Processor 103
5.3 Space–Time Diagram for a Linear Pipeline 104
5.4 Feedforward and Feedback in Pipelines 106
5.5 Methods to Solve Pipeline Bottlenecks 107
5.6 The Speedup and Efficiency of a Pipeline Processor 112

5.7 Pipeline Speedup with and without Vector Looping 112
5.8 Performance of Some Established Computers 119
5.9 A Serpentine Memory 127
5.10 Nishihara and Larsen's Gaussian Convolver 128
5.11 S-Access Memory Interleaving 129
5.12 C-Access Memory Interleaving 130
5.13 Relative Speedup from Overlapping I/O and Processing 130
5.14 A Two-Dimensional Pipeline 134

6.1 A Serpentine Array Computer 140
6.2 A Parallel-Pipelined convolver. 148
6.3 Parallel-pipelined convolver : memory structure. 149
6.4 Parallel-pipelined convolver : system controller. 150

7.1 Functional Block Diagram 154
7.2 Convolver Card Set 1 155
7.3 Convolver Card Set 1 156
7.4 System Clock Block Diagram 160
7.5 Memory Enables 161
7.6 PC Interface 162
7.7 Analogue Interface 165
7.8 An Instruction-data Based Architecture 171
7.9 System Microcontroller 173
7.10 Memory Multiplexers 174
7.11 The Convolver 177
7.12 Address Generation 181
7.13 Memory Layout 186
7.14 Phased Address Controllers 188

List of Tables

4.1 Comparison of I/O Methods 98

5.1 Comparison of Computer Performance 118
5.2 Handler's Pipeline Classification 121

7.1 Physical Card Split 157
7.2 System Clock Notation 159
7.3 Inverted Clock Signals 159
7.4 PC Interface 162
7.5 Video Timing Controller Loading 168
7.6 Memory Mapping Table 175
7.7 Memory Mapping in I/O Mode 175
7.8 Convolver Functions and Decoding 178
7.9 Storage of Columns in Frame Stores 184
7.10 Pipeline Register Bits 195
7.11 Pipeline Register Bits – continued 196
7.12 Pipeline Register Bits – continued 197
7.13 Column Map Proms 197
7.14 Row Map Proms 198
7.15 I/O Map Proms 198
7.16 Row Load Proms 199
7.17 Column Load Proms 199
7.18 I/O Load Prom 199
7.19 Vector Map Prom 200
7.20 Instruction Map Prom 201
7.21 Inter Board Connectors–Section 1 202
7.22 Inter Board Connectors–Section 2 203

9.1 Errors Found In Debugging 213

Chapter 1

INTRODUCTION

Vision is our most complex sense. It is also our most versatile, providing us with information about our suroundings under a wide range of conditions. Our ability to gather such information from images is matched only by our ability to form representations of these images that we may use later in visual processing. Thus we have a remarkable duality; the ability to extract information from very complex scenes, often under adverse lighting conditions, and the power to store this knowledge in a way that allows it to be rapidly accessed for use in future scenes.

In an industrial environment, vision provides one with a large range of skills; the ability to select components that are randomly orientated or obscured, to assemble these within very fine tolerances, to check for correct manufacture, and to move around in a complex and often hostile environment. For two decades now, research has been aimed at providing automated machines or robots with these capabilities, to remove the need for human operators in repetitive or hazardous tasks, and to provide increased speed, consistency and quality of manufacture.

The first algorithms written to perform these tasks were primitive in the extreme, and often very slow to run. They were based around the use of a grey-level (monochrome) image, and the application of a threshold to this image. The aim was then to extract information about potentially interesting objects in the scene by segregating them from the background via the threshold operation. This method was soon proved to be virtually useless in a real environment, where differing lighting

conditions, and the obscuration of the objects prevented the program from running reliably.

It was soon realised that it was necessary to adopt a more scientific approach to machine vision, by using, for instance, methods from signal processing. It was also realised that it would be necessary to take account of the geometry of the scene, and also to use, if possible, some a priori knowledge about the image. This led to the development of more complex algorithms, which performed better in real (non-laboratory) conditions.

However, the gap between the algorithms used in industrial practices today, and those being developed in the now flourishing research base is widening. A major obstacle to technology transfer from laboratories to industry is the requirement for rapid execution. The laboratories have concentrated on the competence of the algorithms that they develop, and are perfectly prepared (if unhappy) to wait an hour for the results of such algorithms. This is a luxury in which industry cannot indulge: its demand is for systems that work both quickly and reliably.

Originally the solution to this performance problem was to use images with very small resolution and/or few grey levels, with the first vision systems operating on a 64×64 binary image. Useful applications for such systems using backlit tables and simple algorithms have now largely been exhausted. Producing a reliable binary image, for example, is the job of edge detection, which is a surprisingly tough task. Resolutions of between 256×256 up to 1024×1024 at 256 grey levels are now the norm for vision systems. The combination of such large amounts of data, with the need for algorithms that provide accurate results under a wide variety of conditions, and that will operate at sufficient speed to make them of use in an industrial environment, poses a tough challenge.

In this thesis, we examine one particular aspect of the problems outlined above; that of increasing the speed at which some of the latest vision algorithms can be executed, without significantly increasing the hardware cost. In the next Chapter,

we provide an overview of a number of vision algorithms which represent state of the art research. These span diverse problems, from the detection of edges in an image, to the calculation of shape from shading, or computing stereo images. We look first at established vision algorithms, and note that most of their computation is linear and shift invariant, being performed with local convolutions over a small neighbourhood. These are often combined with thresholding, to produce the final results. The discussion then proceeds to advanced algorithms, and we show that the mathematical framework under which most of these problems are solved is similar. Most of the problems are underconstrained, and are solved by the addition of a regulariser, which is an extra constraint (often one of geometry) imposed upon the system. The problem is then solved using local iterative methods which frequently use repeated convolution-type operations. We see that convolution is thus a generic operation for all levels of visual processsing. This is an unusual discovery, as traditionally algorithms which perform on the intermediate level aspects of vision processing have been thought of as requiring different processing methods to those that perform low level operations.

We proceed in Chapter 3 to review some of the hardware that has been designed to perform local convolutions at the processing rates that industry requires. These are mainly array computers, which consist of large matrices (often many thousands of elements) of processing elements, each capable of acting individually on a single image pixel or group of such pixels. The processors usually act in SIMD (Single Instruction Multiple Data) mode, with all processors performing the same operation at once. Array processors such as these are ideally suited for providing the sort of local computational support that many of the latest algorithms require. The type of parallelism found in these machines provides greatly increased performance over the serial computers that were originally used for vision algorithms, by, as we show in Chapter 2, reducing the time taken to find the edges in a single image from 42 minutes on a serial computer to 2.5 seconds on a relatively small (32×32)

element processing array.

We proceed to show in Chapter 4 that, in spite of the hugely increased performance that such array computers provide, there are often very significant bottlenecks in the processing. These arise through the time taken to input and output images to and from the processing array. We show, by an algebraic analysis of a general processing array when used for vision, that two components exist to the I/O time, as well as the time taken to actually process the image. These two components are:

1. The time taken to store an image after it is output pixel-serially from a camera.

2. The time taken to transfer this image into a (usually spatially smaller) processing array.

We conclude that the total I/O time taken in such operations can be significantly greater than the processing time in a number of potential operations, and that the first component of the I/O time may often outweigh the second component.

The problem of keeping a processor supplied with data at sufficient rate has been faced by the designers of serial computers, for example to increase the performance of arithmetic units. A generic solution that was used, and which is widely adopted in industry today, is pipelining, where elements in the processing chain are separated by registers. Data flows from stage to stage in the pipeline, with each stage concurrently executing either consecutive functions or the same function.

We analyse pipelining in general terms, and perform a detailed analysis on the performance gains that we may expect when employing pipeline techniques. The problems that affect pipeline processors are assessed, and the potential applications of pipelining to vision processors are discussed. A number of ways in which pipelining has already been used are analysed, and the ways that these methods can be used in vision machines are discussed. We also present some new ideas on the ways in which pipelining can be of use in a vision processor.

In Chapter 6, three design vignettes for vision processors are presented to overcome I/O bandwidth problems. These are designed to support algorithms of the type mentioned in Chapter 2, and all of the machines utilise pipelining in one or more ways to increase their performance. One of these machines is optimised for interfacing the processor array with a pixel-serial camera, in an attempt to reduce the I/O time taken in loading the processing array. The second device is a semi-intelligent frame store, which can be programmed to format video data between the camera and the processing array. The third device is a small processing array, designed to provide the local convolution support that emerged in Chapter 2 as a generic operation for visual processing. It utilises pipelining in three key areas to increase performance; in the controlling microprocessor, in the arithmetic unit itself, and in the data pathways between the ALU and the two frame stores, each of which holds a 512×512 image.

This particular design for a *Parallel-Pipelined Convolver* resulted from collaboration with I.B.M. U.K. Laboratories, who sponsored this research on a CASE studentship. It has been designed with specific industrial applications in mind, and had a low target price (of the order of £5000). It is designed to provide video rate convolution support for a 256×256 pixel image, and quarter video-rate local support for a 512×512 image. The design is described in detail in Chapter 7. A picture of the device is shown in *Figure 1.1.* It features pipelining in the controller, the convolver, and in the data pathways between the frame stores and the convolver.

The controller pipeline comprises an instruction lookahead which doubles the operating speed of this part of the system. A threefold increase in speed is provided by the pipelining of the data paths between the memory and the convolver. This data pathway may be used in either byte-parallel mode, or as a high speed serial link. Finally, the convolver unit itself is internally pipelined, with a latency of one clock cycle, the net effect being to make the entire system six times faster than a similar serial device would be, for minimal extra hardware expense.

In Chapter 8, the Parallel-Pipelined Convolver design is analysed with the framework established in Chapters 4 and 5. We show that performance gains of between two and three times over a serial computer are made with each of the three applications of pipelining in the device. Although this machine was designed to be a strictly practical vision processor, rather than a pure testbed for pipelining in the context of machine vision, we have shown that pipelining is a potential solution to many of the problems currently affecting the design of vision processors, where the tradeoff between the increasing accuracy and robustness of algorithms is offset by the increased time taken to process them.

We finish this thesis with an overview of the work presented, concluding that pipelining can play a major role in increasing the speed of the latest generation of vision processors. We add some suggestions for further research. These include a description of the construction and debugging of the parallel-pipelined convolver, which is now largely complete. Work on the system software will continue, and benchmarking the system with the algorithms detailed in Chapter 2 is planned. We also present a new design, which would provide a more flexible environment for testing the applications to which pipelining can be applied in vision.

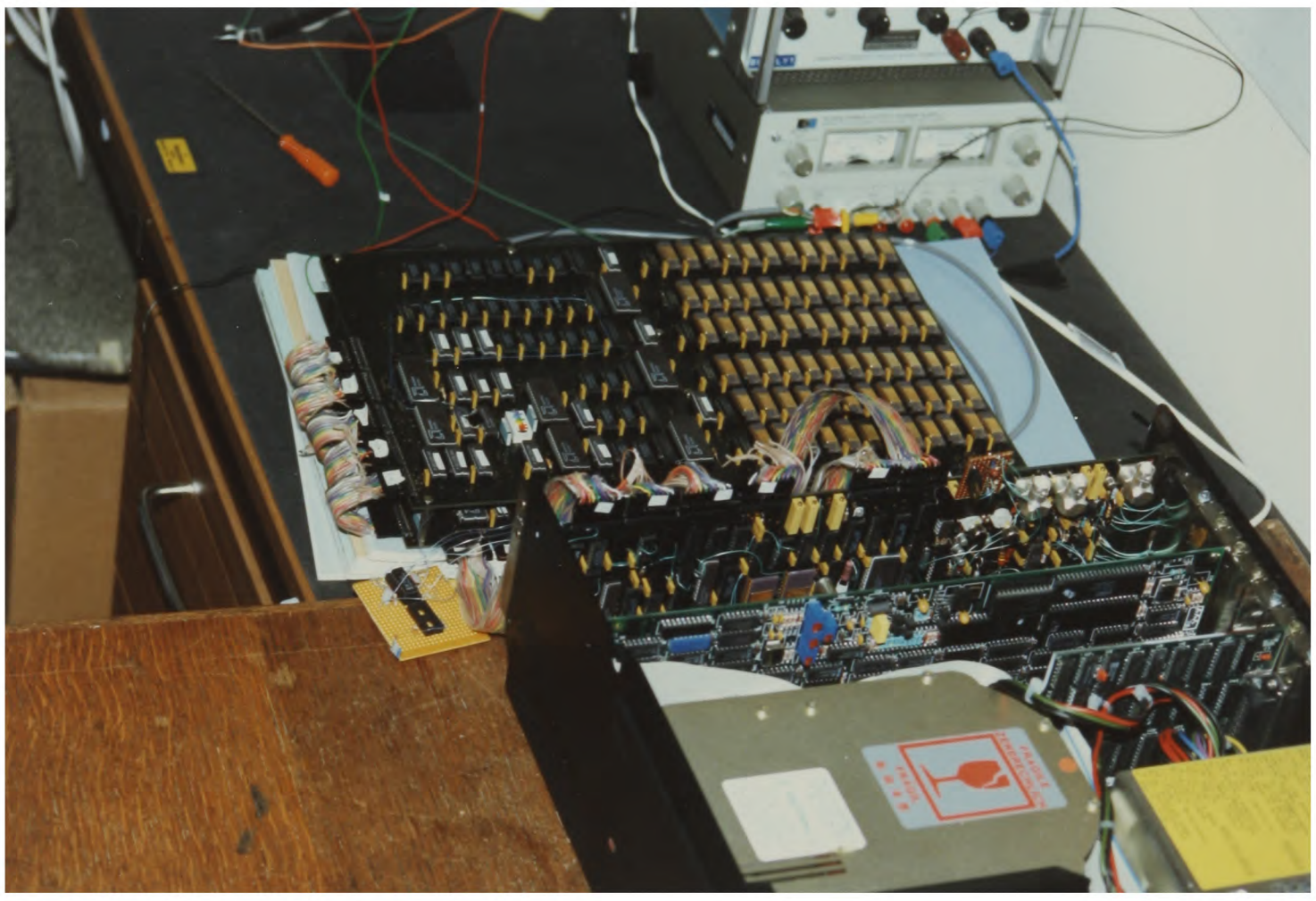


Figure 1.1: The Parallel Pipelined Convolver installed in its host PC.

Chapter 2

VISION ALGORITHMS

2.1 Introduction

We start this chapter with a discussion of low level vision algorithms that currently form the main basis of industrial vision. These are linear shift-invariant operations, which are computed by local convolutions applied to a small pixel window. The use of such convolutions in low level vision has made industry aware of the need for parallelism in vision computers, so that the calculations may be performed at the required speed. However, convolution is often regarded as an end in itself, with the idea that a fast convolution board, with a limited library of kernels, can perform all of the tasks necessary for an industrial vision system.

This is far from being the case, and we move on to discuss algorithms which compute more complex visual operations such as stereo, shape from shading and motion. These are in use and development in research laboratories, but infrequently in industry. Their basis is a mathematical analysis of the geometry (and noise statistics) associated with visual operations like stereo. This makes such vision problems highly non-linear, in contrast to low level vision algorithms.

Such problems are underconstrained, and may not be solved without the addition of external constraints. These are often in the form of some global image parameter, such as the position of the light source with respect to the scene and the viewer. The problem may then be solved by techniques such as regularisation, and these are discussed.

Regularisation leads to a set of equations which may be solved by iterative methods such as relaxation, that involve very large numbers of local calculations. Surprisingly, the local operations are often linear, and may be performed by convolution. In this way a global approach to the vision problem, designed to provide far more accuracy and detail in the amount of visual information available, leads to a solution largely composed of convolutions or other simple operations. Such convolutions may be thought of as being generic to all levels of vision processing, rather than, as is the case today, as an operation which is only useful in a small number of primitive vision algorithms.

We illustrate this with a description of a number of state of the art vision algorithms, few of which are yet in use outside of a research environment. These must, however, be the basis on which the next generation of industrial vision systems are to be based, and we highlight the widespread use of convolutions in such algorithms.

2.2 Low Level Algorithms

In this section we discuss some of the algorithms used in current industrial vision processors. They are local operations. For example, we replace the pixel upon which the operation is centered by the average of the 8 or 24 surrounding pixel grey levels (this is a crude smoothing filter) thus:

$$x_{i,j} = \frac{1}{24} \sum_{\delta,\mu} x_{i+\delta,j+\mu}$$

Most of the algorithms that are used in vision processing to detect edges, highlight features etc, have been developed from the idea of taking a simple directional derivative across an image, as we show in *Figure 2.1.*

These led to the most commonly used edge detectors, the Sobel (Sobel et al, [1969]) and Prewitt filters (see Kittler [1983] for a discussion). These are simple directional masks, defined as follows:

$$\begin{pmatrix} -1 & 0 & 1 \\ -k & 0 & k \\ -1 & 0 & 1 \end{pmatrix}$$

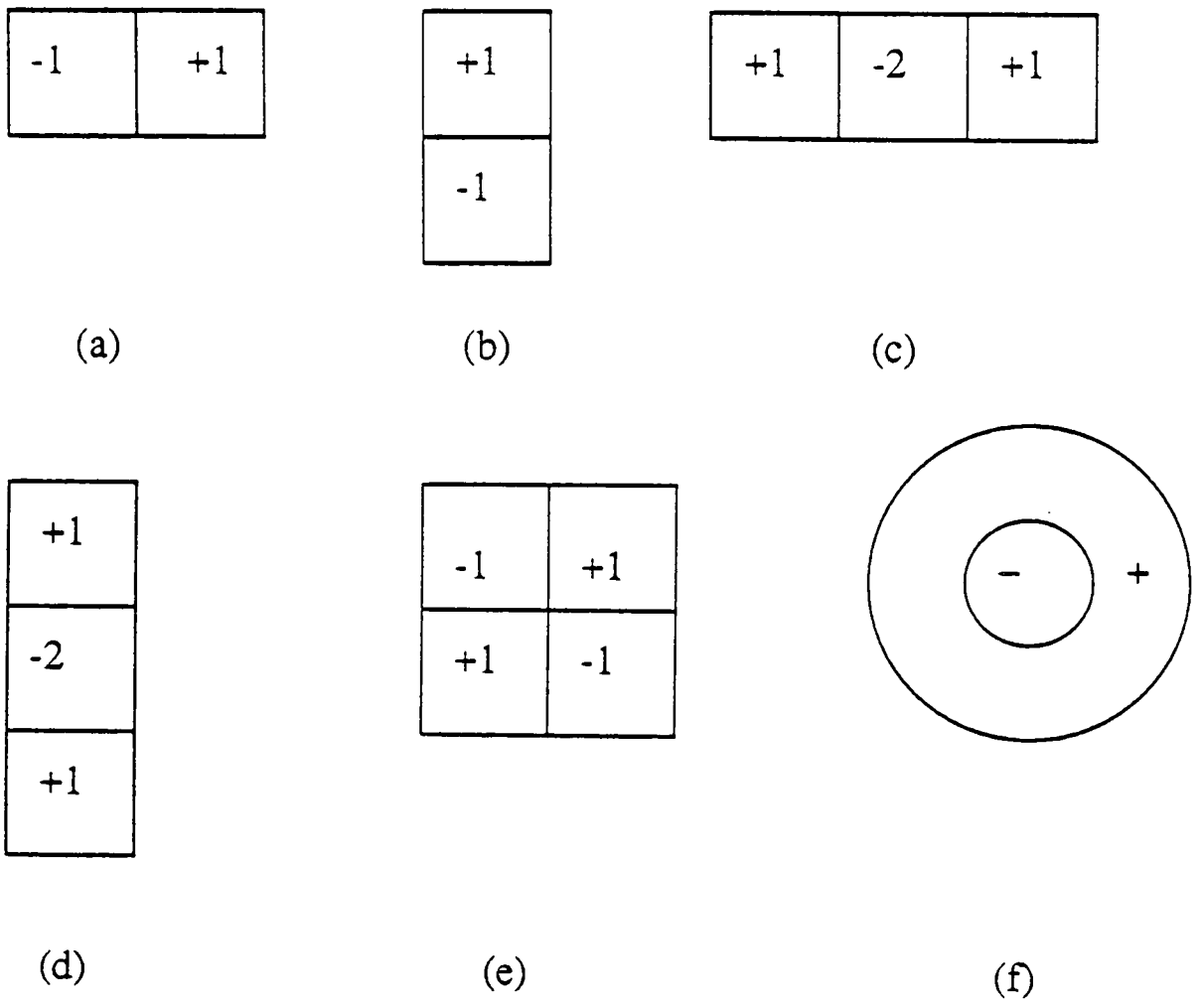


Figure 2.1: Approximations to directional derivatives. (a)and (b) show approximations to $\frac{\partial}{\partial x}$ and $\frac{\partial}{\partial y}$. The operator $\frac{\partial^2}{\partial x^2}$ may be considered the difference between two neighbouring values of $\frac{\partial}{\partial x}$, and so may be represented as in (c). The other two second-order operators appear in (d) and (e). Lastly, the Laplacian ∇^2, which is the lowest order isotropic operator is shown in (f) in circularly symmetric form.

The mask shown is one to detect an edge in the x-direction. The y-mask is simply rotated through 90 deg. The value of k is a variable, but if $k = 1$ then the mask is known as a Prewitt mask, and if $k = 2$ then it is a Sobel mask. Results from the applications of filters in the x and y-directions are usually combined by squaring, summing, and square rooting the result. This is typically followed by a thresholding operation, to produce a binary image which, in theory, contains all of the edge information, with all of the noise filtered out. The idea is then to perform some sort of feature matching on this binary edge map, often by comparing it with a stored set of "ideal" image data. Edge filters such as those of Sobel and Prewitt work well in some circumstances, particularly if imagery can be carefully controlled as is occasionally possible in industry. They perform poorly on uncontrolled grey level images, however.

If we assume that the image is 512×512 pixels, then there are $\frac{1}{4}$ M pixels in the image. A simple edge filter, such as the Sobel operator (Sobel et al [1969]) requires 20 operations per pixel to perform; even with this simple convolution, the number of operations required is large.

We show an example of using the Sobel in *Figure 2.2*, which shows an image, the results of the Sobel operation upon it, and the results of a more recent algorithm for edge finding, developed by Canny [1985]. As one an see, the Canny operator produces an image with much less noise than the Sobel filter, without losing too much of the edge detail. However, this is not without extra computational cost.

2.2.1 Signal Processing Approach

This approach has been fully discussed by Pratt [1978]. We discuss it briefly here for comparison with the algorithms mentioned in the last section. It relies on the fact that a large class of image processing algorithms may be linearised; an output image is formed from linear combinations of pixels of an input image, by operations that may include convolution, superposition and discrete linear filtering. Considering an

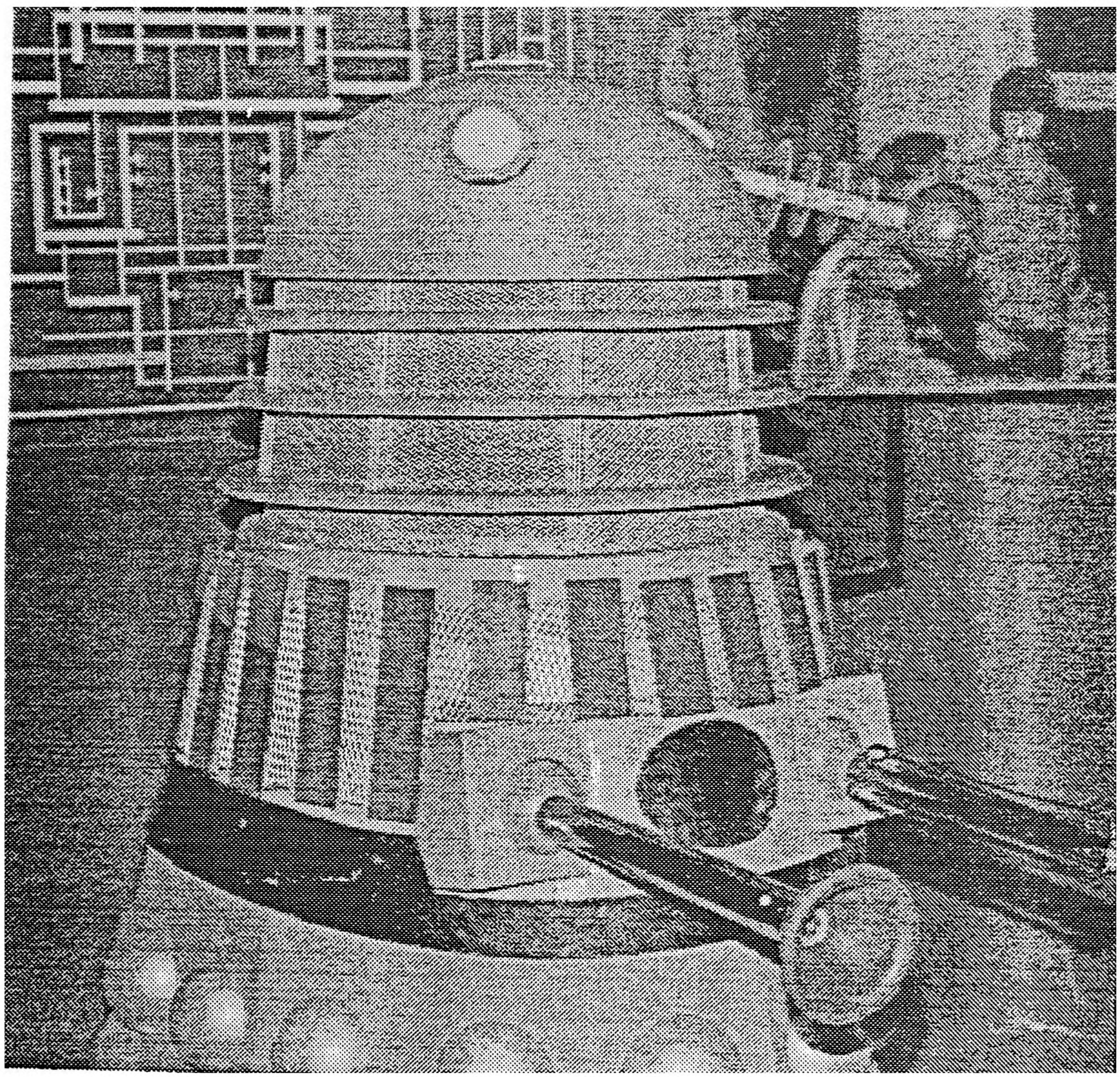


Figure 2.2: This shows the original image of a dalek.The next two figures show the results of Sobel and Canny filters on this image.

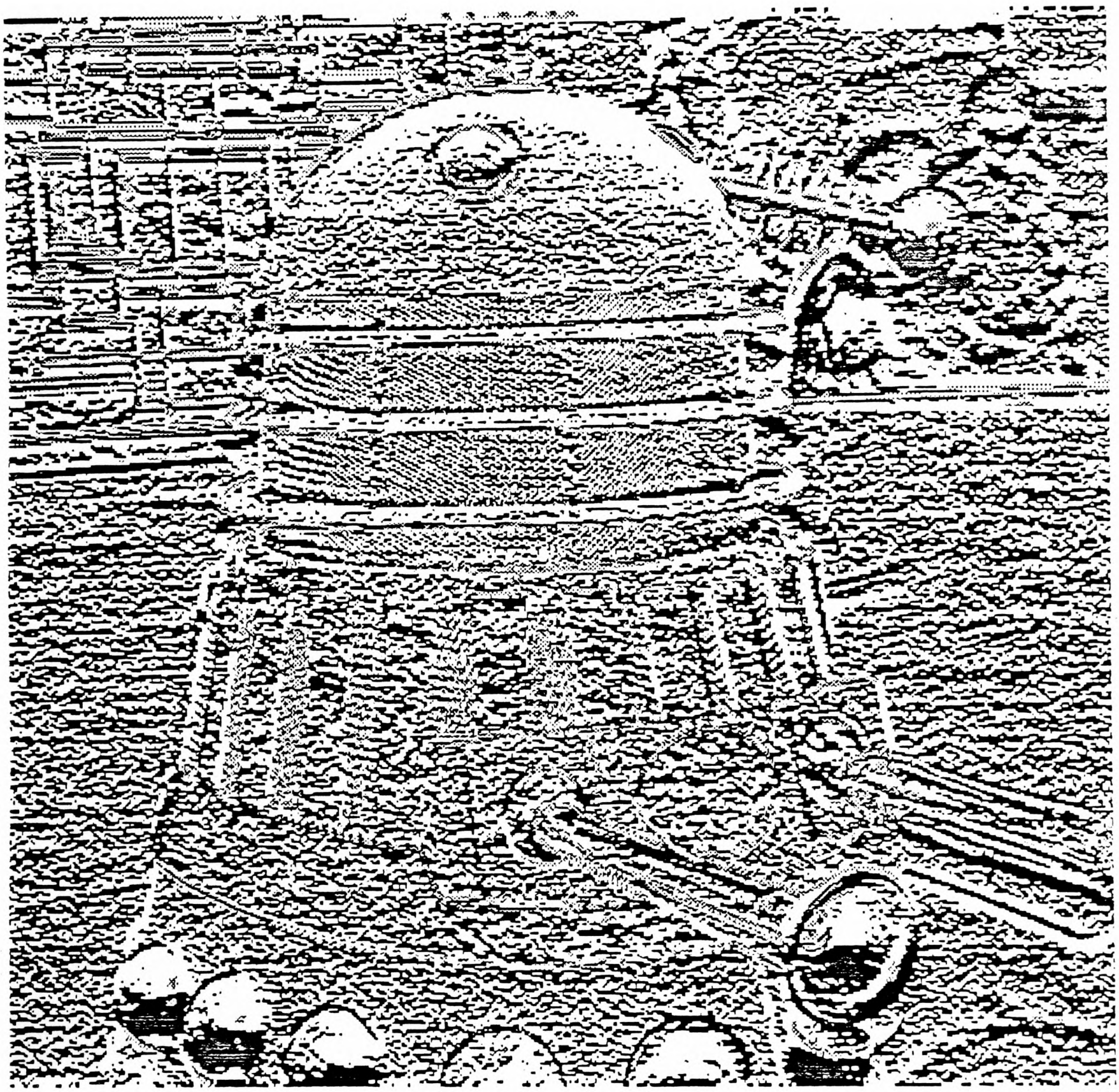


Figure 2.3: The dalek image after application of the Sobel filter, and thresholding. The resulting image is very noisy, but has a lot of detail.

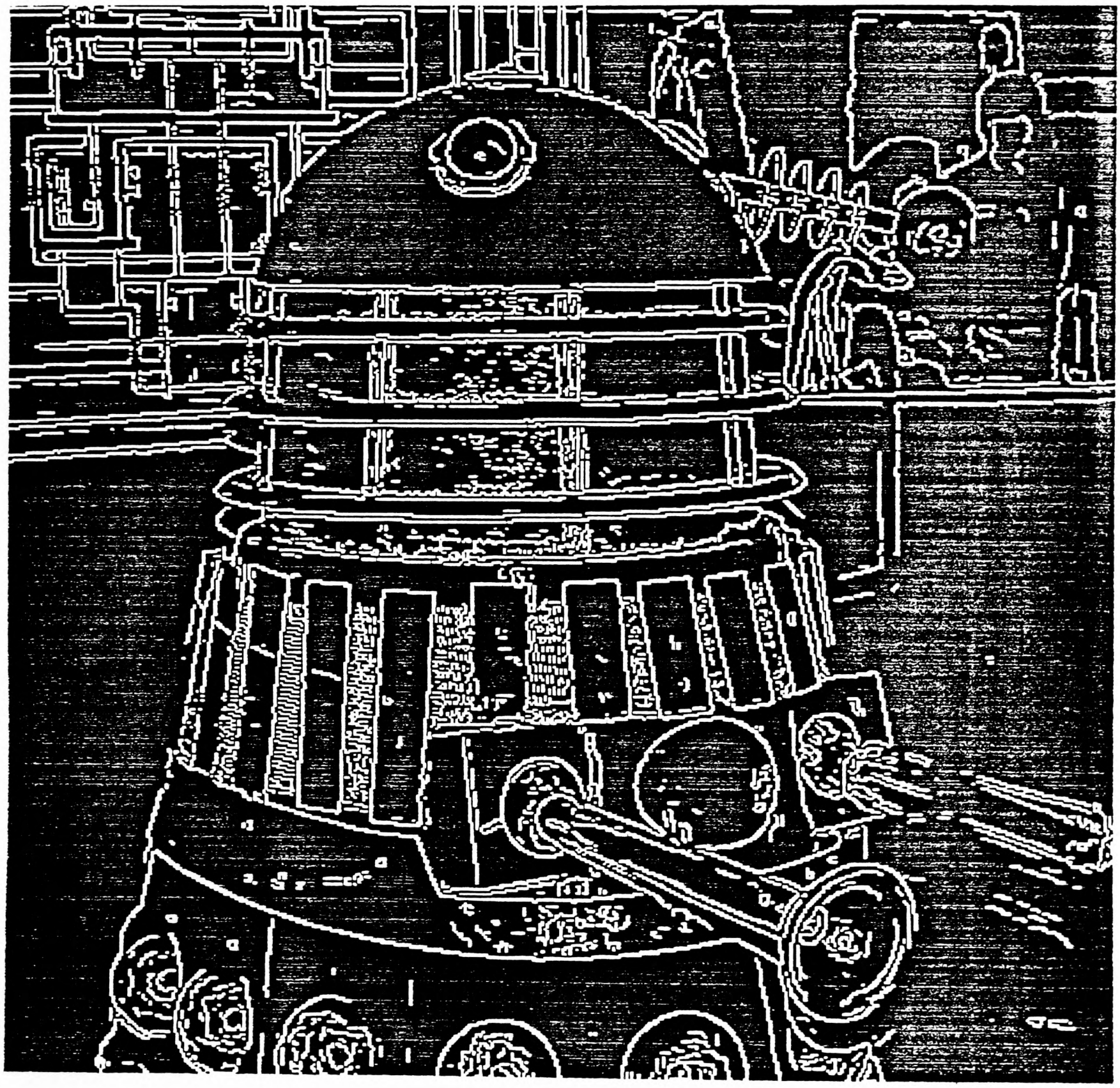


Figure 2.4: The dalek image after application of the Canny edge finder. The results have far better noise characteristics than those obtained from the Sobel filter, but at far higher computational cost.

$N_1 \times N_2$ element input image $F(n_1, n_2)$; a generalised linear operation on this image field results in an $M_1 \times M_2$ output array $P(m_1, m_2)$ as defined by:

$$P(m_1, m_2) = \sum_{n_1=1}^{N_1} \sum_{n_2=1}^{N_2} F(n_1, n_2) O(n_1, n_2; m_1, m_2) \tag{2.1}$$

where the operator kernel $O(n_1, n_2; m_1, m_2)$ represents a weighting constant which is, in general, a function of both input and output image coordinates.

Pratt shows that in the case where the image transformation $\mathbf{T}$ is a separable two dimensional linear transform, the output array may be calculated by sequential one dimensional row and column operations on the input array:

$$\mathbf{P} = \mathbf{T}_C \mathbf{F} \mathbf{T}_R^T \tag{2.2}$$

This approach is often used with the Laplacian of Gaussian filters used in many recent algorithms. Because of the separability of the two dimensional Gaussian, it is often more computationally efficient to split the filter into two one-dimensional ones, in particular when the computer is a serial one.

At this point we will leave general signal processing ideas, but we must first mention the Median filter, which has found widepread use in the last decade in industrial image processing. Median filtering (see Huang [1979], Oflazer [1983], and Narenda [1978] for examples) is a non-linear signal processing technique developed by Tukey, where the output image pixel is taken as being the median of the $n \times n$ pixels in a subwindow around the input pixel, for an $n \times n$ filter. It proves to be a very effective technique for removing random spike noise in an image, but is poor at preserving thin edges, where the edge pixels may be completely erased by taking the median of a large surrounding window. Thus the filter is often of limited use in real situations.

2.3 Why Convolution

We have seen that a typical low level vision algorithm uses local operations. This means that the same local operation is applied over the whole image, using each

pixel as a center for the calculations. The mathematical operations performed in this way are also usually linear. The combination of these two facts means that the operations performed are usually convolutions.

It would seem that convolution, and other local operations form the basis of industrial vision systems. In many cases, the convolution is often seen as an end in itself, and a limited variety of convolution kernals are assumed to provide the solution to all vision problems.

We look later in this chapter at a number of intermediate level vision algorithms which take a highly non-linear approach to solving vision problems. It would thus seem very unlikely that convolution, a very common operation in low-level vision processing, would be of any use in the solution of such problems. As we shall see, this is not the case, in much the same way as a complex curve may be approximated by a series of piecewise linear segments. We then show that convolution may indeed be a generic operation for all types of vision algorithms.

2.3.1 Benefits From Parallelism

Having seen that convolution plays an important role in early vision calculations, what effect does this have on the computation of such algorithms? If we take the Canny algorithm for edge finding as an example, this takes of the order of ten thousand operations per pixel to perform. Hence, for the Canny operator, using an image that is 512×512 pixels, the number of operations required is

$$\frac{10^6}{4} \times 10^4 = \frac{10^{10}}{4} \text{operations in total}$$

If the computer has a performance of one million operations per second *(MOPS)*, then it will take it

$$\frac{10^{10}/4}{10^6} = \frac{10^4}{4} \text{seconds}$$

that is about 42 minutes ! This calculation is confirmed by personal experience on a SUN-2 computer, which took almost three quarters of an hour to run the algorithm. Obviously, such performance is of little or no use in an industrial environment, even

if the results eventually obtained are the desired ones. What this illustrates is that, even with low level vision, the numbers of calculations involved are great. However, the latest generation of video processors, such as the Datacube, can perform such operations at video rates. If more complex algorithms are to be executed, to provide better optimised solutions, then there can often be a cost/benefit tradeoff, as the processing time becomes too great.

Particularly in the early stages of visual processing one may obtain significant performance gains by exploiting spatial parallelism in our calculations. Previously we indicated that it would take approximately 42 minutes to perform one iteration of the Canny edge finder over a 512×512 image, using a conventional serial computer. If we were to use a computer with, say, an array of 32×32 processing elements (1024 elements in total), and if we assume the same $1MIP$ rate as for the serial computer, then the same algorithm would take us

$$\frac{2.5 \times 10^3 \text{ seconds}}{10^3}$$

seconds of our processing array. If we assume that each processor in our array has the same basic processing rate as the original serial computer, then the total time taken to perform the edge finder would be only 2·5 seconds. This is a huge improvement over the serial computer although is still some way from achieving real time video rate processing of visual images. Machines already exist which contain such processor arrays, and we will look at these in the next chapter.

2.4 State of the Art Vision

We have looked briefly in the last two sections at vision algorithms which are used in almost all of the current industrial applications for machine vision systems today. We proceed in this section to introduce the framework on which most state of the art vision systems have been developed. These are algorithms which perform a far wider range of functions than the current industrial type, but which are still largely confined to laboratory use.

2.4.1 The Geometry Viewpoint

As we discussed in section 2, typical low level vision algorithms such as the Sobel operator, which find widespread use in industry today, were developed largely by looking at an image as a physical entity, and performing operations over the area of this image. However, algorithms which have been developed recently start from a different viewpoint. They regard the image as a two-dimensional projection of a three-dimensional scene, and analyse the problem, whether it is one of shape, lightness, stereo etc, from *global* three-dimensional constraints, such as the geometry and noise statistics associated with the problem.

This allows much more relevant information to influence the result of the algorithm, and is the main reason why the latest generation of algorithms produce much more detailed and accurate results than their predecessors did. However, this advance is not without extra cost; a scene in space is three-dimensional; adding motion takes us to six dimensions. However, images are only two-dimensional, and motion in images three-dimensional, so information is lost. This often means that the problem to be solved is underconstrained. Hence no solution can be found, unless further constraints are imposed upon the problem. The realisation that the lost dimensionial information must be recovered, led to the development of a whole new class of algorithms, which we discuss next.

2.4.2 The Need For Constraints

We see that by taking a global viewpoint of the three-dimensional scene that projects onto an image, we lose dimensional information from the problem. Hence extra constraints are now necessary if the problem is to have a unique solution. One of the first types of algorithm to solve an underconstrained problem was concerned with the analysis of shape.

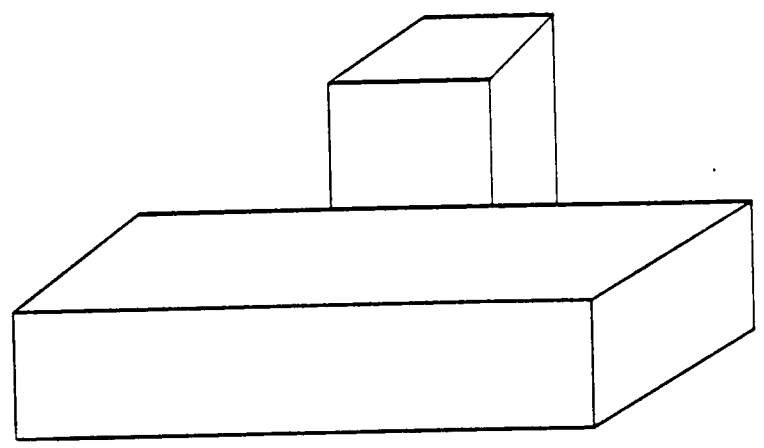
Huffman [1971] and Clowes [1971] worked on interpreting the objects in a picture. Their work followed on from that of Guzman [1968], who had developed an

heuristic program based entirely on image structure that proved very successful at labelling objects in an image. Huffman and Clowes explained the heuristic behaviour of Guzman's program in terms of the geometry of the scene, and showed how the program could be expanded. Looking at *Figure 2.5 (a)* the lines in the image were interpreted as edges. These were labelled as either convex, concave or occluding types. Similarly, junctions in the image are interpreted as vertices in the world. These vertices are obviously made up from intersections of edges. One would thus expect $3^3 = 27$ labels for a 3-line junction, but in fact only three labels are physically possible. This was a constraint upon the three dimensional problem, which simplified it significantly. However, in spite of constraints such as these, Huffmans's algorithm still left a tree search with $O(2^n)$ operations. However, these local constraints did not take account of the planarity of a face (see *Figure 2.5 (b)(c)*), where the objects are labelable as planar.

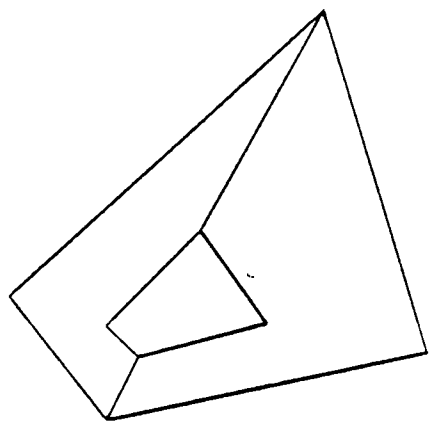
This technique was developed further by Waltz [1975], who was concerned with increasing the constraints on the possible set of edges. Waltz in fact complicated the problem (already $O(2^n)$) by adding more line types. However, he also introduced a *global* constraint to the problem by imposing the constraint of a single point light source upon the image. This dramatically improved the performance of the algorithm. Waltz classifies the primitives in an image in the following way:

- Boundary lines and shadows.
- Augment by including shadows as an extra constraint.
- Concave lines are classified to reflect the number of objects coming together and how these objects obscure each other.
- This line information is combined with illumination information.

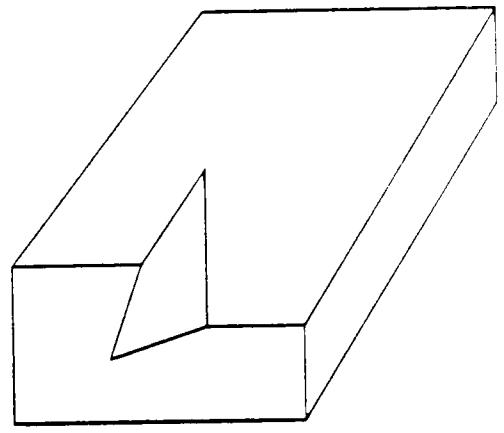
This gives a very large set (approximately 10^8) possibilities for line types. To solve for a particular scene, Waltz's procedure propagates symbolic constraints over the image. The process may be summarised as follows:



(a)



(b)



(c)

Figure 2.5: Huffman and Clowes' Shape algorithm. Part (a) shows a small part of a scene in which lines may be interpreted as edges. The scene in (b) may be interpreted as planar, whereas that in (c) is interpreted as a 3-dimensional figure.

1. Form a queue consisting of all junctions. Assign a pile of labels for each of the junctions.

2. Until the queue is empty:

   - Remove the first element from the queue. Call it the current junction.
   - If any junction label from the current junction's pile is incompatible with all of the junction labels from a neighbouring junction's pile, eliminate that incompatible label from the current junction's pile. Note that a pile change has occurred.
   - If a pile change has occurred, for each neighbouring junction with a pile that is not on the queue, add that junction to the front of the queue.

This was the first real use in vision of the idea of propagating constraints across a large set of data, to allow the solution of an underconstrained problem.

This process is similar to propagating numeric constraints, as is done nowadays in computer programs such as Lotus 1-2-3 and Visicalc, which use add-multiplier nets to propagate data. Thus if one alters a single data value in Visicalc, this change will propagate through the rest of the data, changing any of the data values that are linked to the first value.

One surprising interpretation of Waltz's work was that the solution employed was very similar to the mathematical technique of relaxation (see Rosenfeld, Zucker and Hummel [1976]). Relaxation is a mathematical technique used for solving large sets of linear equations, e.g.

$$\mathbf{Ax} = \mathbf{b}$$

where $\mathbf{x}$ is large and $\mathbf{x} \in \Re^n$. The general technique works as follows:

1. Guess $\mathbf{x} = \mathbf{x}_0$

2. The error in the guess is $\mathbf{Ax}_i - \mathbf{b} = \epsilon_i$.

3. Update with a new guess. $\mathbf{x}_{i+1} = f(\mathbf{x}_i + \epsilon_i)$. Thus one might say: $\mathbf{x}_{i+1} = \mathbf{x}_i + \epsilon_i$.

4. Loop to (2) until $|\epsilon_i| < |\epsilon_{\text{desired}}|$.

The relaxation formula relates the new value of the parameter to the old value and to the value of the parameter at neighbouring sites, usually with a confidence parameter.

2.4.3 Regularisation

We have discussed that the change from taking a local view of an image to a global view of the scene that is represented requires the addition of extra constraints to enable the problem to be solved. The mathematical technique which is used to solve most of these underconstrained problems is regularisation.

Regularisation works essentially as follows:

1. Set up an energy function for the particular problem to be solved.

2. Add the necessary constraints. A constraint such as one of smoothness of flow is the regulariser for the problem.

3. Establish and solve the corresponding Euler-Lagrange equations for the variational problem associated with the energy function.This typically yields a set of equations involving F_{min}, giving rise to an iterative equation.

4. Solve for F_{min} using local iterative methods.

As can be seen, the use of regularisation to solve the variational problem often leads to a set of equations which may be solved using iterative local methods, such as relaxation algorithms. This is a surprising conclusion; that the adoption of a global view of a three-dimensional scene, instead of a local image perspective, still leads to a local solution to the problem. A local solution was to be expected where the operations were linear, as in the examples which we have discussed. However, many intermediate level algorithms are highly non-linear, so the conversion to a linear solution is a very useful one in computational terms.

The first researchers to come to this conclusion were people like Waltz whom we have already mentioned. The theory is covered comprehensively by Horn [1986], and we will summarise the main points here:

- Calculate the global state of a particular image parameter from the local interaction of state parameters.
- The mathematical basis for this calculation is using variational methods on a vector space of functions.
- Derive an energy function for the parameter to be solved and minimise this function to solve the problem.
- The existence of such a minimal energy, F_{min}, can be proved (see Grimson, [1981]).

Although this might seem like a very computationally "neat" way of being able to perform quite complex calculations, it will not work for all vision problems, as we illustrate in the next section.

2.4.4 Relaxation

We have illustrated that a mathematical framework exists for solving quite complex problems by means of regularisation techniques. These lead to solutions which are often in iterative form. This conclusion in itself was unexpected; what is more surprising is that these iterative solutions can often be performed by means of convolutions, which we showed to be a generic operation for low level vision algorithms. This means that the change from viewing the image as an entity, and analysing small sections of it using local convolutions, to looking at the scene as a whole, has not lead, as expected, to very complex solutions, but instead will often lead to a solution which is once more in the same form as that used for low level algorithms. We see that the convolution can indeed be a generic operation for all vision algo-

rithms, playing a strong role in the calculation of complex global parameters. This is in opposition to the idea of it being an end unto itself for low level algorithms.

Of course the number of convolutions required to solve the often large sets of sparse equations produced by regularisation methods may be very large. This is less of a problem than it might seem, as the hardware which has been designed for performing low level vision algorithms may now be used to perform the majority of the work required in calculating these intermediate level algorithms.

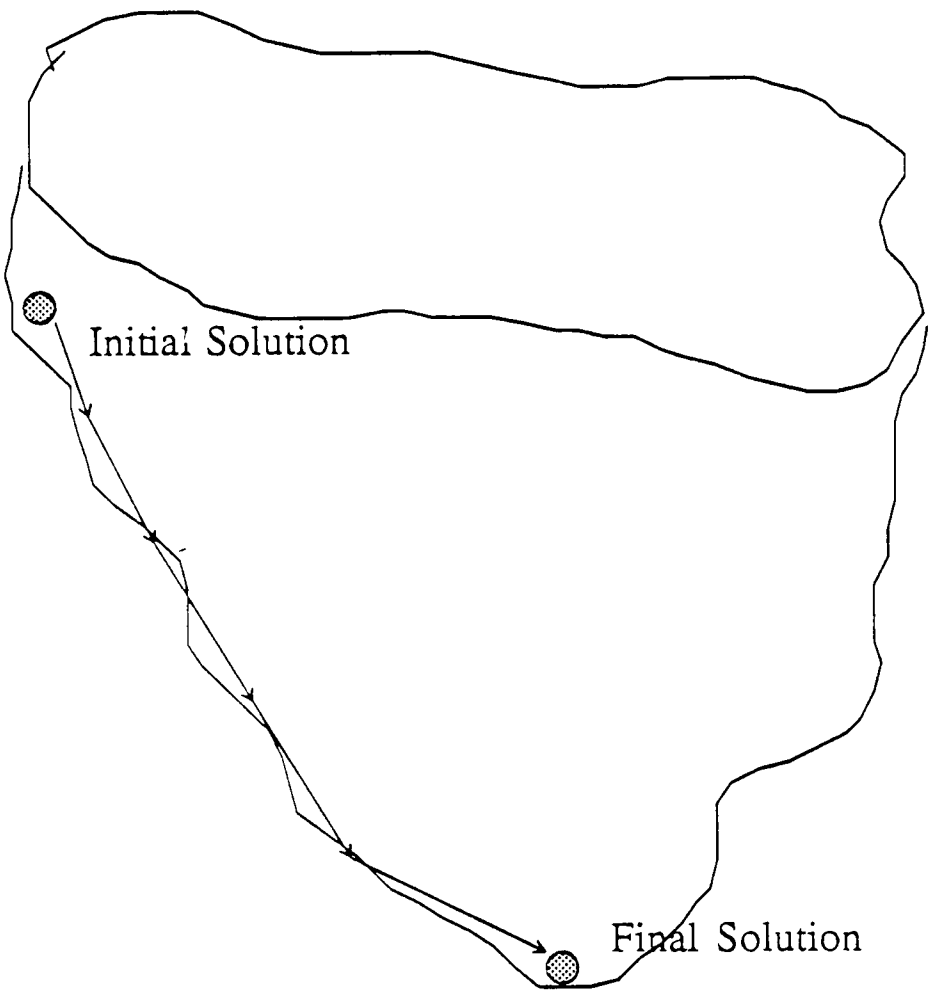
However, in order for the solution to converge, the calculations must be solved by iterating to a solution over a parameter space that is convex, see *Figure 2.6*, usually, as we have seen, by means of local convolutions. This means that the relaxation algorithm will always (eventually) converge to a solution. However, this is not the case with all of the problems in machine vision. Many problems give rise to parameter spaces that are non-convex, which may lead to algorithms converging to incorrect results, as shown by the local minimum in the figure.

Differing approaches have been made to solving such non-convex relaxation problems. Work has been done in this field by Blake and Zisserman [1985], and by Geman and Geman [1984]. We will look later at Geman and Geman's algorithm as an example of this type of relaxation, which may not be solved by local convolution.

2.4.5 Global State From Local Interaction

We summarise the last section by observing that vision algorithms being developed in research environments usually take a global view of the image. That is to say the image is regarded as a two-dimensional projection of a three dimensional scene. Three-dimensional properties are than analysed, using global constraints such as the scene illumination, position of the viewer etc. Thus a *global* image characteristic is calculated, with global constraints imposed on the solution.

This can be done by regularisation techniques, which often lead to an iterative solution, which may be in the form of a relaxation algorithm. This in turn may be



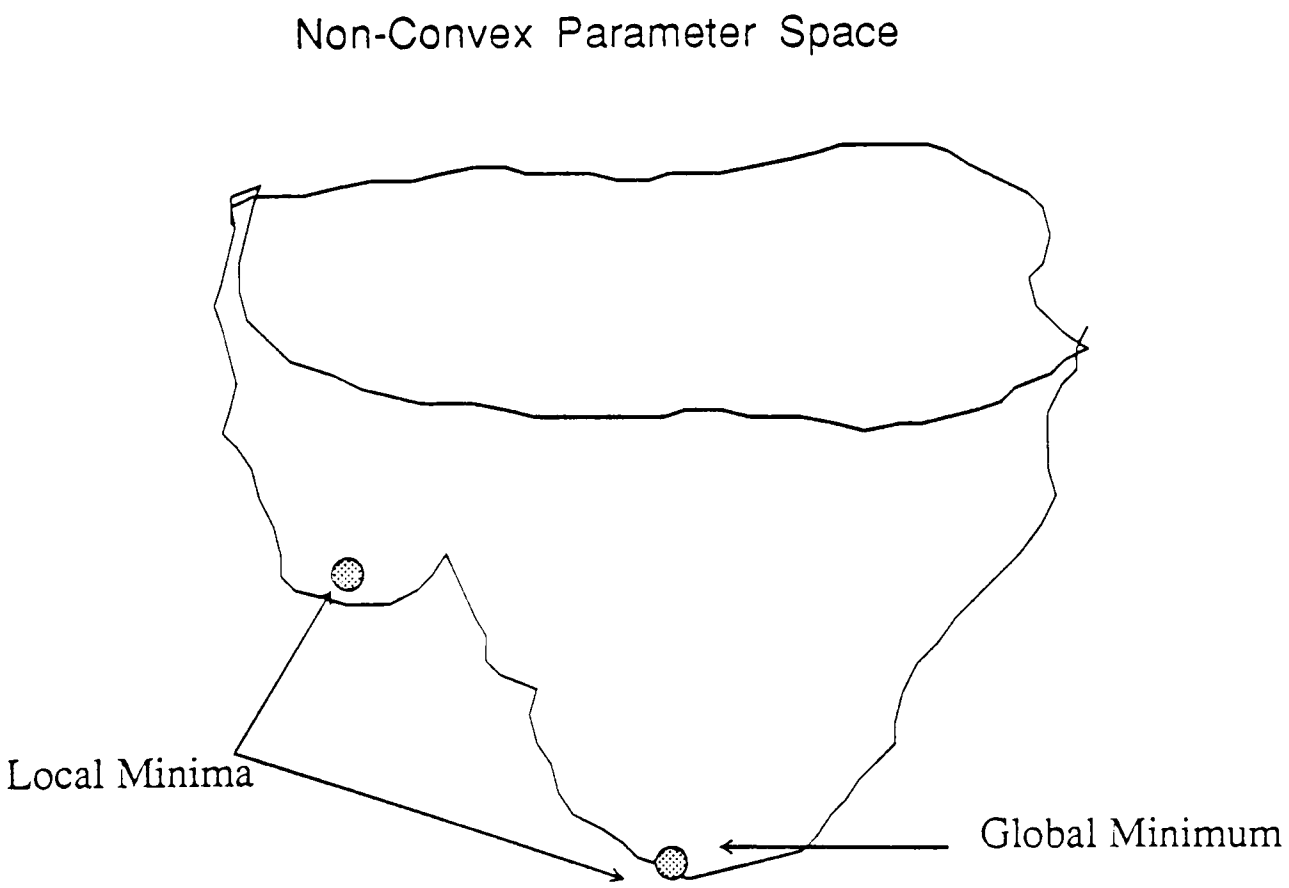


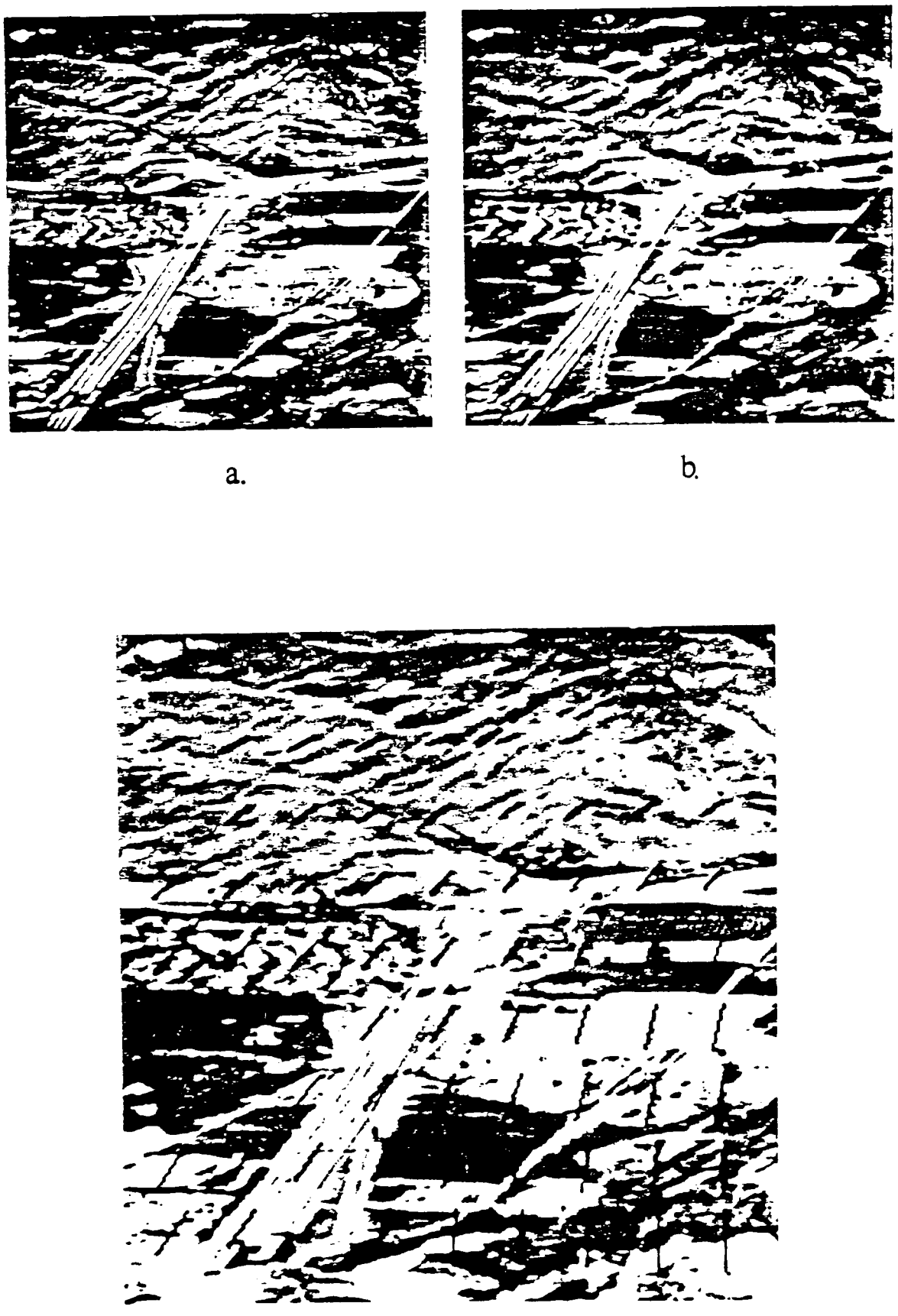
Figure 2.6: Figure (a) shows a convex parameter space in which a relaxation algorithm will iterate to a solution. In (b), depending upon the initial solution, the algorithm may not converge, because of the non-convex parameter space.

solved by an iterative method, using local calculations, often convolutions. Thus the actual mathematical solution to the problem is often in the form of iterated *local interactions* between pixels, such as convolutions, or local thresholds etc.

We thus develop the idea of *Global State from Local Interaction.* Algorithms which have been developed in this framework form the basis of current university research into vision computing. Many examples of this have been published in the literature. Work here has been done in areas which had previously been thought unattainable for local computation. Examples include shape from shading (Ikeuchi and Horn [1981]), (Brooks and Horn [1981]), optic flow (Horn & Schunck [1981]), structure from motion (Ullman [1979]), shape from contour (Witkin, [1981]), (Brady and Horn [1983]), stereo (Drumheller and Poggio [1986]), surface interpolation (Grimson [1981]) and (Terzopoulos [1984]) We discuss these algorithms in detail later in this chapter. For now, I show in *figures 2.7* to *2.12* some examples of the results that the above global state from local interaction programs can produce.

As we will see later, the underlying issue in most of these problems is that they are mathematically underconstrained. The solutions adopted by Hildreth, Horn and Schunck, Brady and Yuille, and Ikeuchi and Horn all add extra constraints, often in the form of a smoothness rule, which is a regulariser for the problem. The use of regularisation leads to an iterative solution of the problem, solved via local interactions in a parallel process. We will see this theme repeated through all of the algorithms discussed, with the main differences being in the accuracy of the result, and the speed of convergence.

We start our discussion with the shape from shading algorithm of Ikeuchi and Horn [1981], which provides a good example of a global image characteristic which may be calculated with repeated local convolutions.



a. b.

Figure 2.7: Results from Hildreth's optic flow algorithm. Pictures (a) and (b) represent two images taken in sequence from an airplane. Picture (c) shows the computed velocity field, with the average velocity vectors displayed in black.

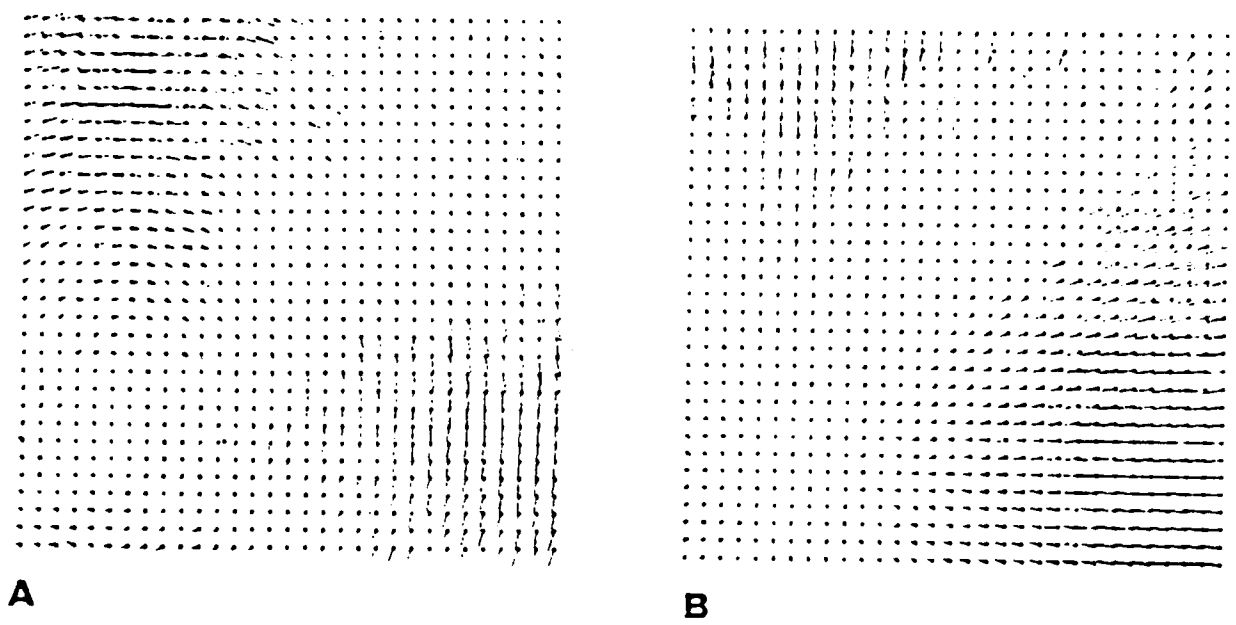


Figure 2.8: Results from Horn and Schunck's optical flow algorithm. (a) shows flow patterns computed for a simple rotation of 2.8 degrees per time step. (b) shows a simple contraction of a brightness pattern of 5% per time step.

Figure 2.9: Results from Canny's edge finder. This figure shows the original image of Westminster abbey. The next figure shows the results from the algorithm.

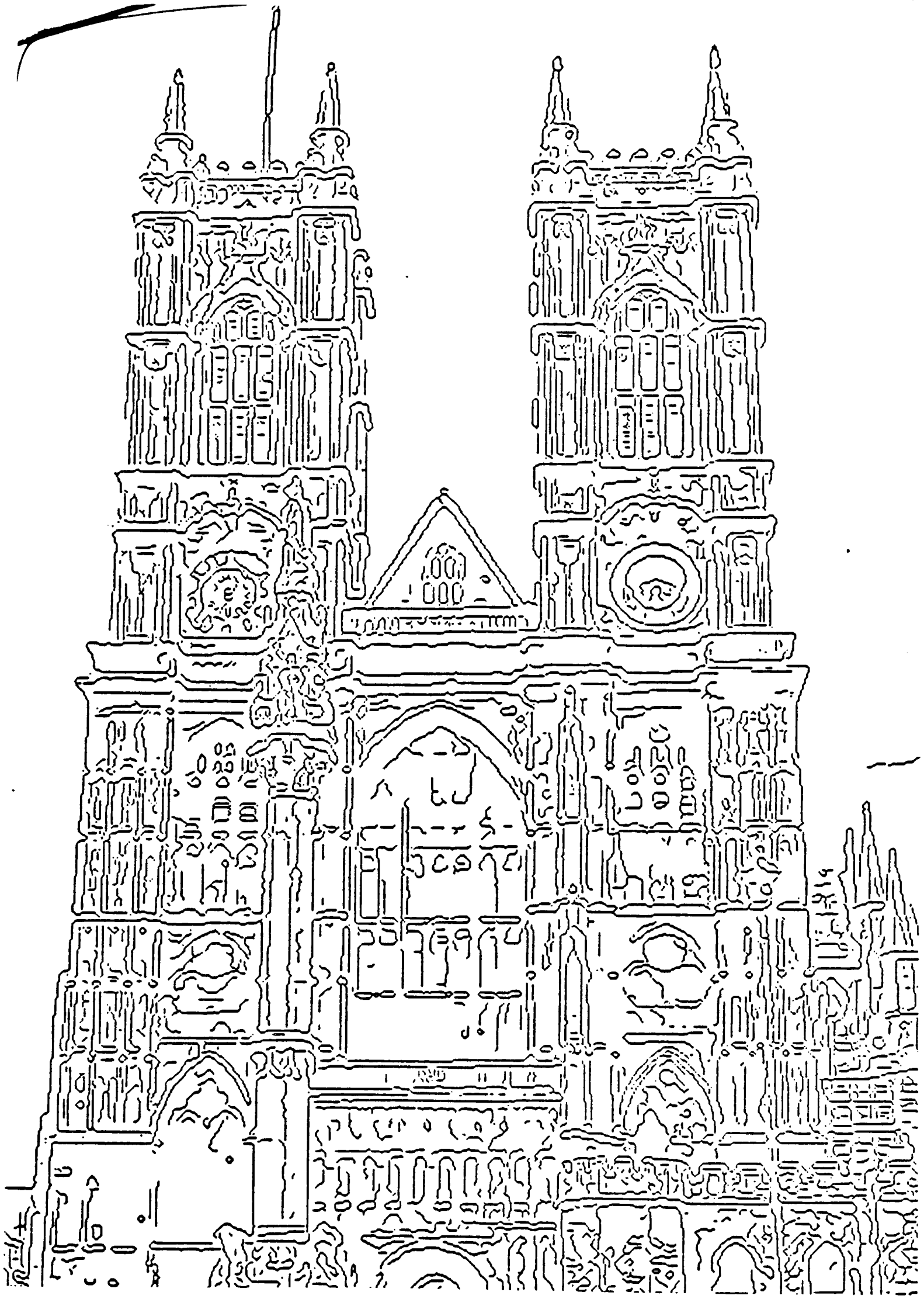


Figure 2.10: Showing the results of the application of Canny's edge finder to the Westminster Abbey image. Note the good performance of the algorithm over areas of varying contrast.

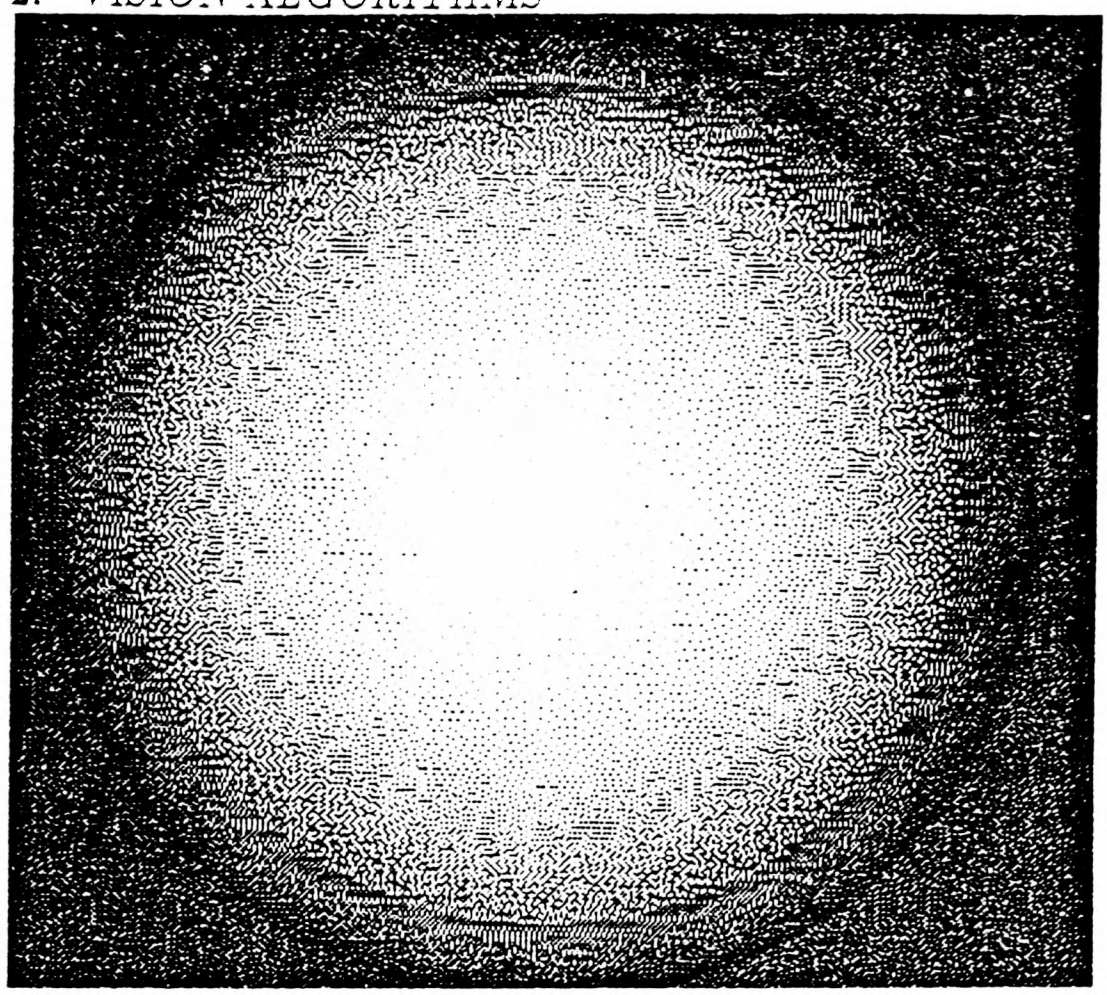


Figure 2.11: Results from Ikeuchi and Horn's shape from shading algorithm. The top Figure is a pseudo grey level image of a Lambertain sphere illuminated by a light source near the viewer. Figures (a),(b),(c) show the initial orientation array, and the results after 5 and 30 iterations.

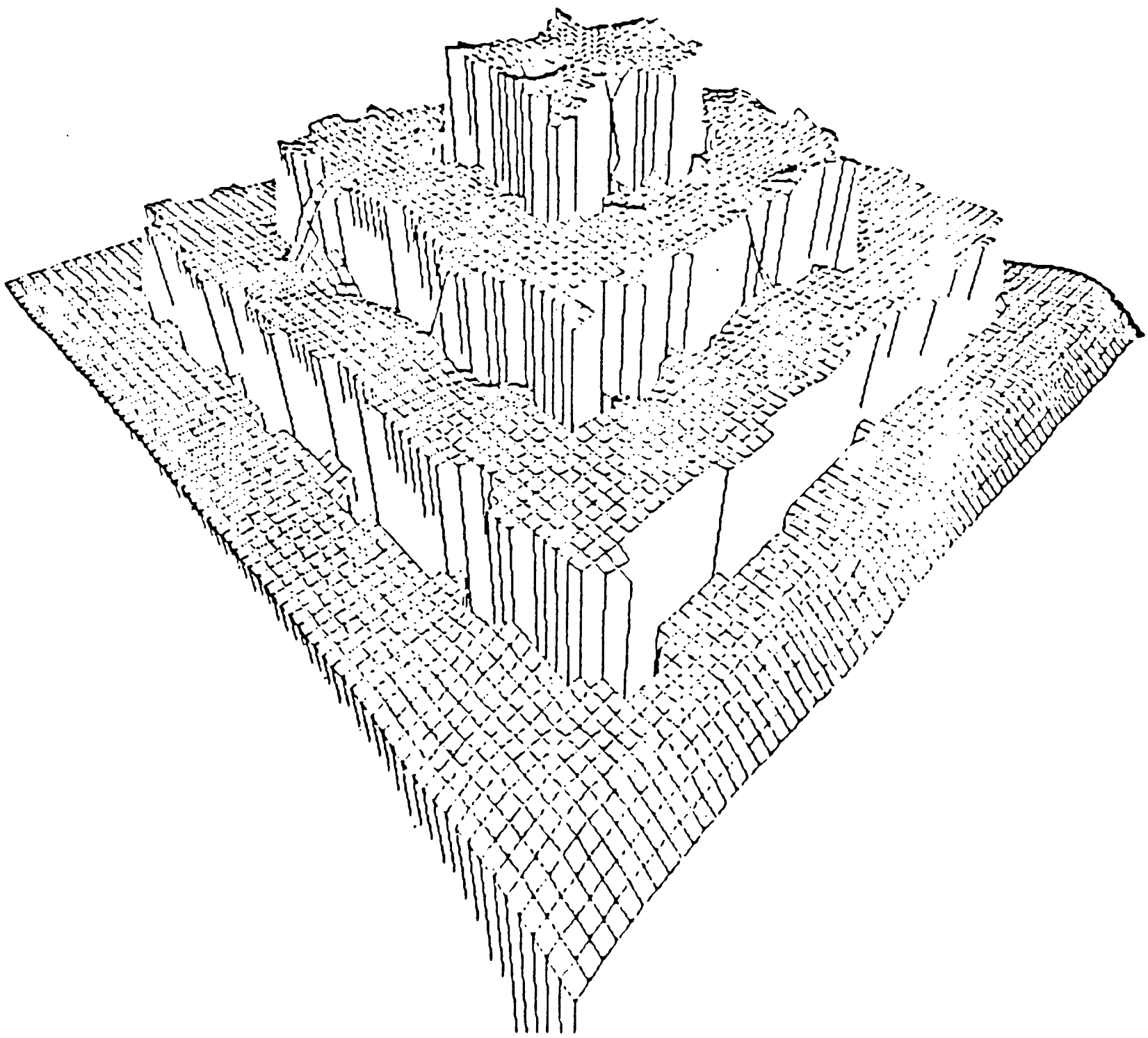


Figure 2.12: Results from Terzopoulos' surface reconstruction algorithm. This shows a reconstructed surface, including surface discontinuities. Note how the algorithm has coped well with such discontinuities.

2.5 Intermediate Level Algorithms

2.5.1 Shape From Shading

Ikeuchi and Horn's [1981] shape from shading algorithm exemplifies the discussion in the last section. The problem is to recover the surface normals from the image of a scene, given the assumptions of a known position of the illumination. Ikeuchi and Horn also assume that the reflectance of the scene is known, and that the viewpoint of the observer is also given, thus reducing a number of the variables in the problem. However, even given these assumptions, an image brightness can still correspond to an infinity of possible surface orientations, as long as the relative angles between observer and illumination are preserved.

However, Ikeuchi and Horn observe that the surface normal is known exactly along the bounding contour. Considering a pixel just inside the bounding contour, we have that for a brightness b there are an infinity of possible surface orientations. However, the normal $\underline{n}$ is known on the border, so the desired normal $\underline{n_{des}}$ must be close to $\underline{n}$ and must satisfy the image irradience equation:

$$E(x,y) = R(p,q)$$

where E is the observed brightness, $p, q = z_x, z_y$ are the viewed surface gradients, and R, the reflectance map involves angles that are dot-products. The Gradient space is one way to represent surface gradients. However, Ikeuchi and Horn use stereographic space (see *Figure 2.13*), with coordinates f, g.

$$E(x,y) = R_S(f,g) \tag{2.3}$$

They do this because in the gradient space projection, points on the equator of the Gaussian sphere, which correspond to surface patches on the occluding boundary are mapped to infinity; in the stereographic projection, the equator is mapped onto a circle of radius two. As their algorithm relies on starting from points on the boundary, the Gradient space projection would obviously be inappropriate. In this

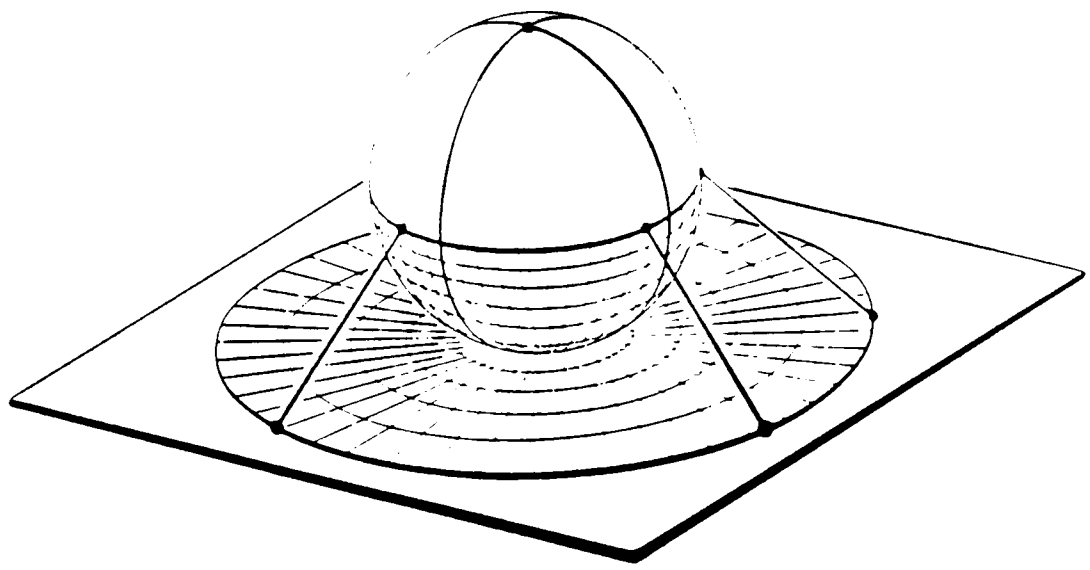


Figure 2.13: Ikeuchi and Horn use the Stereographic Plane rather than the gradient space representation. The stereographic projection maps the equator onto a circle of radius two. The equator of the Gaussian sphere corresponds to points on the occluding boundary of the object

way, they can calculate $\underline{n}_{des}$ and from there calculate points further away from the bounding contour. Using stereographic space, they compute depth from the brightness at each point in the plane, and from the constraint that the surface orientation is known along the bounding contour.

Their algorithm may be summarised as follows:

Define:

$$\begin{aligned} E &= \text{Observed brightness} \\ R_S &= \text{Reflectance map} \\ f, g &= \text{Surface orientation components} \end{aligned}$$

Ikeuchi and Horn then set up an energy function for the problem to be solved. They wish to solve equation 2.3, subject to the constraint on surface orientation along the bounding contour. The variational problem that they formulate is then:

$$\begin{aligned} e &= \int\int \{(f_x^2 + f_y^2) + (g_x^2 + g_y^2) \\ &+ \lambda[E(x,y) - R_S(f,g)]^2\} dxdy \end{aligned} \tag{2.4}$$

The above expression contains a term for equation 2.3 which is to be solved, and a first term where f_x, f_y, g_x, g_y are the first partial derivatives of functions $f(x,y)$ and

$g(x,y)$ which are to be found such that they make errors in the image irradience equation small, whilst also being as smooth as possible. The smoothness function is the regulariser for the problem, whilst the term in the square brackets is the constraint applied to values on the bounding contour. To solve this using regularisation methods, they calculate:

$$\begin{aligned} S_{i,j} &= [(f_{i+1,j} - f_{i,j})^2 + (f_{i,j+1} - f_{i,j})^2 \\ &+ (g_{i+1,j} - g_{i,j})^2 + (g_{i,j+1} - g_{i,j})^2]/4 \end{aligned} \tag{2.5}$$

and

$$R_{i,j} = [E_{i,j} - R_S(f_{i,j}, g_{i,j})]^2 \tag{2.6}$$

Ikeuchi and Horn then minimise the energy term:

$$e = \sum_i \sum_j (S_{i,j} + \lambda R_{i,j}) \tag{2.7}$$

This is done using an iterative method, so the algorithm is:

1. Set up arbitrary arrays f(i,j), g(i,j) to represent the desired surface normal: get pixel array E(i,j)

2. calculate the convolution of f,g with

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

to give average values $f^*(i,j)$ and $g^*(i,j)$. This is a simple local convolution.

3. Compute

$$\begin{matrix} R_S(f^*_{i,j}, g^*_{i,j}) \\ \frac{\delta R_S}{\delta f}, \frac{\delta R_S}{\delta g} \end{matrix} \tag{2.8}$$

from a look up table.

4. Then calculate improved estimates for the surface normals:

$$f_{i,j}^{n+1} = f_{i,j}^{*} + \lambda[E_{i,j} - R_S(f_{i,j}^{*,n}, g_{i,j}^{*,n})] \left(\frac{\delta R_S}{\delta f}\right) \qquad (2.9)$$

and similarly for g^{n+1}

$$g_{i,j}^{n+1} = g_{i,j}^{*} + \lambda[E_{i,j} - R_S(f_{i,j}^{*,n}, g_{i,j}^{*,n})] \left(\frac{\delta R_S}{\delta g}\right)$$

Again, this step can be performed using a local convolution, as long as the facility exists to add a number to the convolved data.

5. loop to 2) until values are "steady".

This algorithm produces reasonable results on a number of real images–as can be seen from Horn and Ikeuchi's paper (see *Figure 2.11*). One problem with the above algorithm is the number of assumptions that must be made initially, such as the position of the light source. The question then arises, could such a global parameter as this be calculated through local interactions on an image of a scene ? Brooks and Horn [1985] have developed a shape from shading algorithm which does determine the position of the light source as well as the surface normals.

This method differs from that of Ikeuchi and Horn in using the Gaussian sphere instead of a projection of it. Hence they use the surface normal in the parameterisation of the reflectance map, rather than gradient or stereographic techniques, in which the surface normal $\underline{n}$ is represented by p, q or f, g, in order to simplify the analysis.

Horn and Brooks define an

$$\begin{aligned} \text{Image} &= E \\ \text{Region} &= \Omega \\ \text{Surface normal} &= \underline{n}(x, y) \\ \text{Direction of point light source} &= \underline{s} \end{aligned}$$

Their basic equations are (for a Lambertian surface):

$$R_S(\underline{n}(x,y)) = \underline{n}(x,y).\underline{s} \tag{2.10}$$

The goal is to solve

$$E(x,y) = \underline{n}(x,y).\underline{s} \quad \forall (x,y) \in \Omega$$

The combination of the last two equations is the image irradiance equation that Ikeuchi and Horn began with. As "usual", this is approximated by a general least squares formulation, to minimise the integral:

$$\int\int_{\Omega} (E(x,y) - n(x,y).\underline{s})^2 dxdy \tag{2.11}$$

Once again, the solution of an underconstrained problem leads to the adoption of a smoothness constraint in the form of a regularisation term. These are well defined and convergent if the regularisation term is quadratic,i.e.

$$\int\int (n_x^2(x,y) + n_y^2(x,y))dxdy \tag{2.12}$$

Brooks notes that in order to make his algorithm converge, he must insist on the "integrability" of the above equation. Chellappa [1987] has recently studied this in a thorough discussion of this algorithm, and reinforces Brooks' condition on integrability. Finally Brooks and Horn insist that the normals have unit length, so that

$$n^2(x,y) = 1 \quad \forall (x,y) \in \Omega$$

These are all combined using the method of Lagrange multipliers, to give the composite functional

$$I(n,s) = \int\int_{\Omega} (E - n.\underline{s})^2 + \lambda(n_x^2 + n_y^2) + \mu(x,y)(n^2 - 1)dxdy \tag{2.13}$$

They minimise this with respect to n,and s using a Lagrange multiplier μ.

It can be shown (Brooks [1985]) that the Euler equation for this variational problem is

$$(E - n.s)\underline{s} + \lambda\nabla^2 n - \mu n = 0 \tag{2.14}$$

$$\text{where } s = ||\underline{s}||$$

Taking the following discrete approximation to the Laplacian as

$$\{\nabla^2 n\}_{i,j} \simeq \frac{4}{\epsilon^2}(\overline{n_{i,j}} - n_{i,j})$$

where

$$\overline{n_{i,j}} = 1/4(n_{i,j+1} + n_{i,j-1} + n_{i+1,j} + n_{i-1,j})$$

This gives a convolution kernel (as in Ikeuchi and Horn) of

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

So the Euler equation translates to the discrete form:

$$(E_{i,j} - n_{i,j}.s)\underline{s} + \frac{4\lambda}{\epsilon^2}(\overline{n_{i,j}} - n_{i,j}) - \mu_{i,j} n_{i,j} = 0 \tag{2.15}$$

This gives an iterative solution of the form

$$m_{i,j}^{k+1} = \overline{n_{i,j}^k} + \frac{\epsilon^2}{4\lambda}(E_{i,j} - n_{i,j}^k.s^k)\underline{s}^k \tag{2.16}$$

$$n_{i,j}^{k+1} = m_{i,j}^{k+1}/|m_{i,j}^{k+1}| \tag{2.17}$$

$$s^{k+1} = \left[\sum_{i,j\in\Omega} n_{i,j}^{k+1} n_{i,j}^{k+1,\mathrm{T}}\right]^{-1} \sum_{i,j\in\Omega} E_{i,j} n_{i,j}^{k+1} \tag{2.18}$$

This solves for $\underline{s}$, the direction of the illumination as well as for the surface orientation components. The first stage in the calculation, that of calculating $\overline{n_{i,j}^k}$ is performed by simple local convolution, as may the rest of equation 2.17, given the same proviso that additions may be performed to convolution results as in the last algorithm. The second step in the algorithm is obviously unsuited to convolution, as is most of the third step, which calculates the orientation of the illumination source.

2.5.2 Edge Detection

We have already discussed the detection of edges in an image, using the industry standard operators such as the Sobel. In this section we look at some more advanced ways of detecting edges, some of which are used in later algorithms.

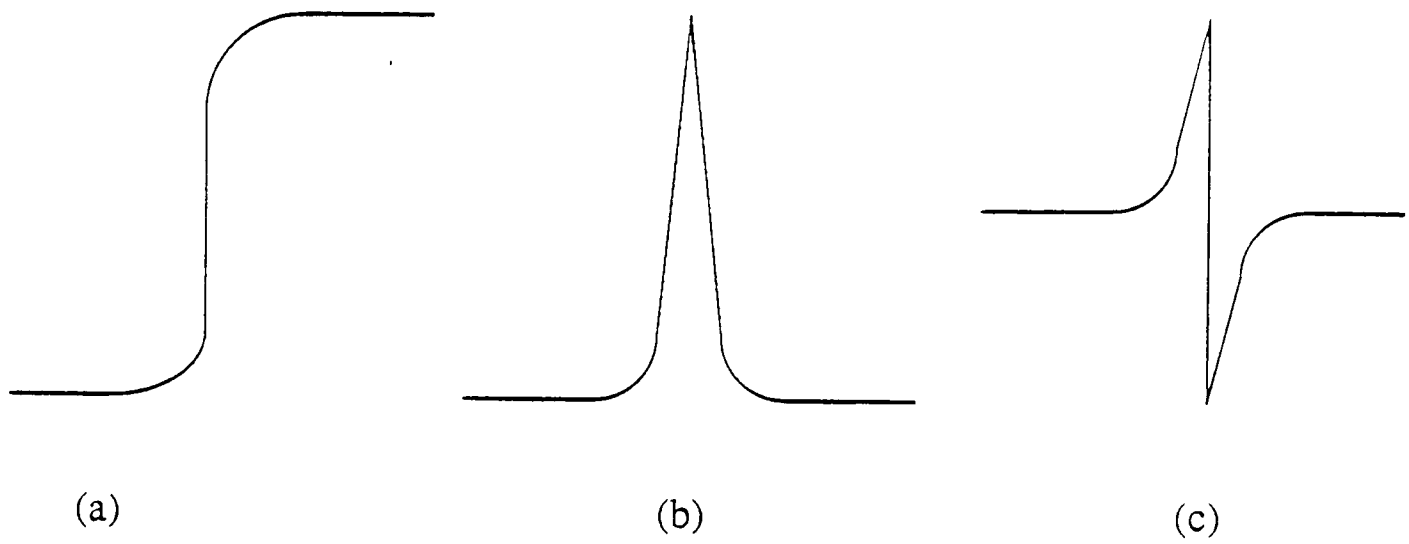


Figure 2.14: A zero crossing edge. The intensity change in (a) gives rise to the peak in the first derivative (b), and to a steep zero crossing in the second derivative (c).

Marr [1982] approached the detection of intensity changes from a psychophysical point of view, proposing two ideas essential to their detection:

- Intensity changes occur at different scales in an image, so their optimal detection requires the use of operators of differing sizes.

- A sudden intensity change will give rise to a peak or a trough in the first derivative, or equivalently, to a zero-crossing in the second derivative (see *Figure 2.14*).

Marr suggested that in order to detect intensity changes efficiently, the desired filter should take either a first (e.g. Sobel), or second derivative of the image. He also suggests that it should be capable of being tuned to act at any desired scale, so that large filters could be used to detect blurry shadow edges, and small filters to detect sharply focused fine detail in the image. This idea that different size filters might distinguish different intensity changes arose from analysing them algebraically as a function of the width of the filters.

Marr and Hildreth [1979] argued that the optimal operator to use is the Laplacian of Gaussian filter $\nabla^2 G$, where ∇^2 is the Laplacian operator $(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2})$, and G

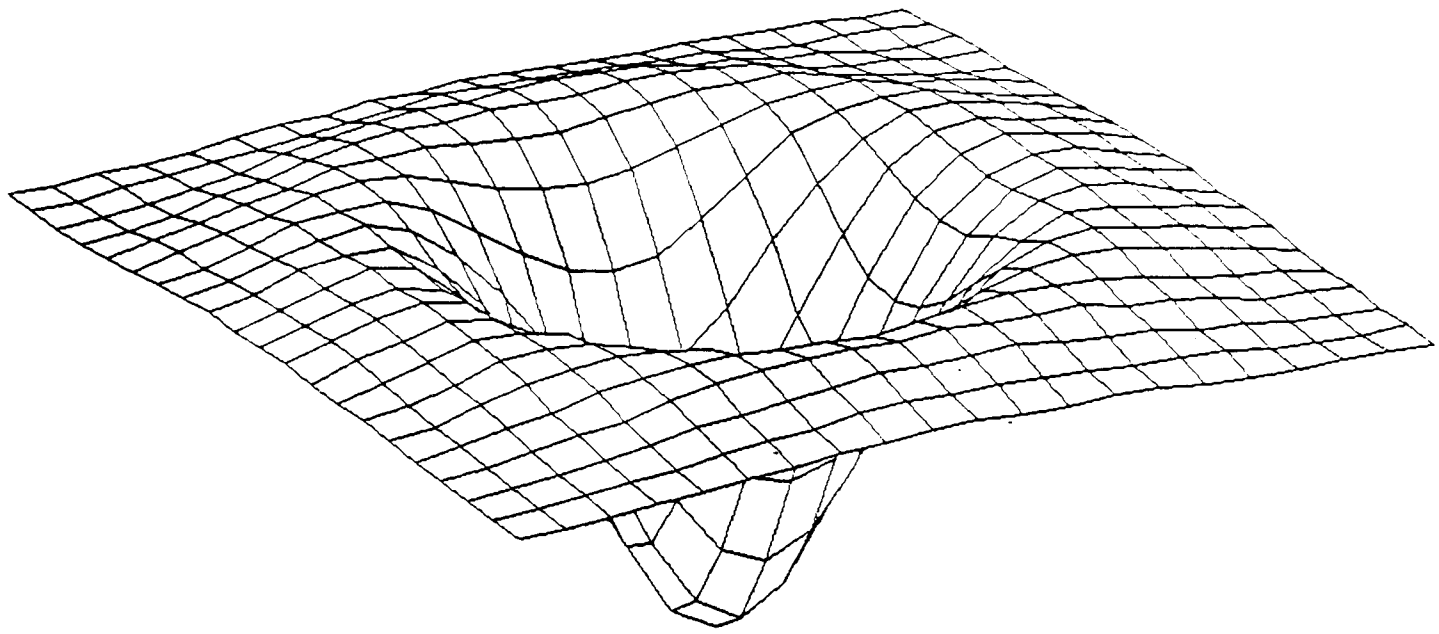


Figure 2.15: The "Mexican Hat" profile of the two dimensional Laplacian of gaussian filter

stands for the two dimensional Gaussian distribution :

$$G(x,y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2+y^2}{2\pi\sigma^2}}$$

with the standard deviation σ. This distribution has the characteristic "Mexican Hat" profile, shown in *Figure 2.15.* Earlier, Wilson and Bergen had developed a model of early retinal image processing based on a difference of Gaussians. Marr suggested that

$$\nabla^2 G = (g_{\sigma 1} - g_{\sigma 2})$$

This claim has recently been challenged however, but still has much support.

Marr and Hildreth justify the adoption of this filter for two reasons. The first is that the Gaussian part of the filter blurs the image, wiping out all detail at scales much smaller than the space constant of the Gaussian σ. The second reason is purely for ease of computation. An isotropic second derivative operator is used to avoid having to combine results from edges detected in different directions. Results

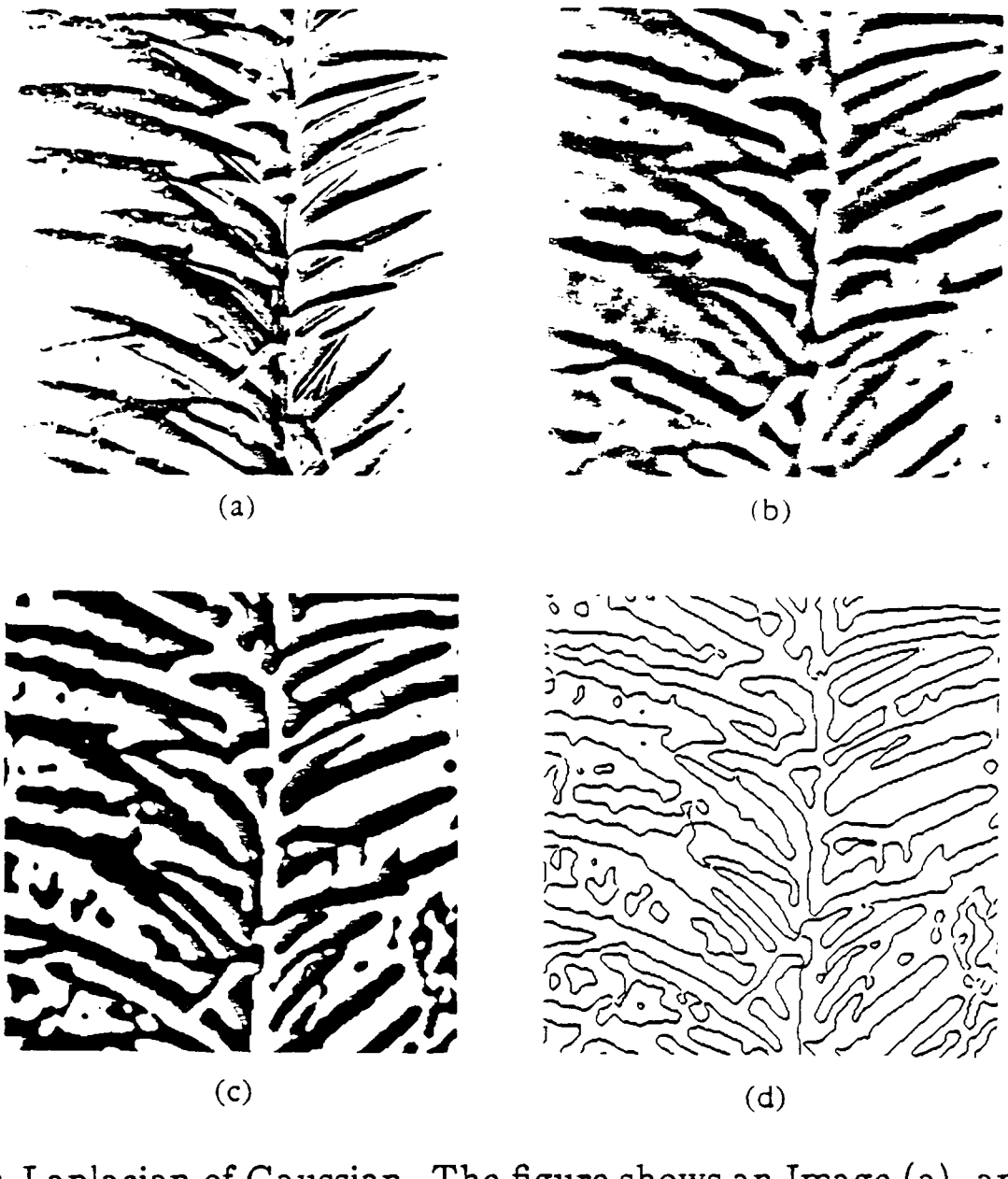


Figure 2.16: Laplacian of Gaussian. The figure shows an Image (a), and the image's convolution with $\nabla^2 G$(b) with $w_{2-D} = 8$. Part (c) shows the positive values in white, and the negative in black, and (d) shows the zero-crossings.

from the filter are shown in *Figure 2.16*.

The type of convolution with Gaussian or Laplacian of Gaussian functions that Marr and others advocated are in common use in many intermediate level vision algorithms. Not surprisingly, much effort has been devoted to performing these convolutions at high speed. Many of these ideas have been discussed by Knight [1983] in his paper which describes the design of a high speed digital gaussian convolver, which is described later in section 5.9. As we have seen, many low level vision algorithms rely upon the convolution of the input image with the two dimensional Laplacian of a gaussian, which is computationally expensive ($O(m^2M^2)$) for large window sizes.

There are numerous ways of saving computational effort. The first is to exploit

the separability of a Gaussian:

$$G_{xy} * I = G_x * (G_y * I)$$

This requires order $O(2mM^2)$ calculations. This method has been used with some success by Nishihara and Larsen [1981] in a real time implementation of the Marr and Poggio stereo matcher, which we describe in section 5.7.1. For this they have to perform a 32×32 convolution with a Laplacian of a Gaussian filter, which they achieve in real time by separating the convolution into two linear ones, and performing these separately with special hardware.

As mentioned before, Marr and Hildreth [1979] have shown that a good approximation to a two-dimensional Laplacian of Gaussian can be provided (within 4%) by the difference of two Gaussian shaped functions. With this approximation, the convolution process simplifies to the problem of convolving the image with a variety of gaussian shaped functions. This appears to be one approximation used by the human visual system to lighten the computational load.

Knight develops two classes of mechanisms for computing these gaussian convolutions. The first relies upon the central limit theorem, which, roughly stated, says that if you convolve together a set of functions with finite width, the resulting function will tend towards that of a gaussian (see Burt [1981]) with $\sigma \propto \sqrt{n}$, where n is the number of iterations. This means that by using small convolution kernels (e.g.3×3), and iterating the convolution, a gaussian of arbitrary width may be approximated. Knight uses a simply computed boxcar function, and applies this function repeatedly to the image. As convolution with this function requires no multiplications, the function is computationally cheap. The most computationally intensive aspect of this is in storing the intermediate convolution result, whilst the second convolution is performed. Knight's second method for convolving an input image with a series of distinct gaussian functions relies upon the behaviour of a distributed resistor/capacitor transmission line, and is described in Chapter 5.

The idea of using Gaussian filters in edge detection has widespread support, and

we look at one such intermediate level algorithm for edge detection below, which uses Gaussian filters in the early stages of the algorithm.

This edge detector was described by Canny [1985], who developed an edge finder which, although it produces less detail than the standard Sobel type, provides a much less noisy output, which is often easier to interpret. Some example results from this have already been shown in *Figure 2.9.* Canny's algorithm proceeds as follows:

1. Smooth the image with a gaussian filter G_σ.

$$I_\sigma = G_\sigma * I$$

   This is performed at four differing scales, again as advocated by Marr, and is the first example of convolution in the algorithm. It may be performed by any of the methods which we have just discussed.

2. Take the gradient of the blurred image.

$$\frac{\partial I_\sigma}{\partial x}, \frac{\partial I_\sigma}{\partial x} \tag{2.19}$$

$$||\nabla I|| = (I_x^2 + I_y^2)^{\frac{1}{2}} \tag{2.20}$$

   The gradient is usually taken by convolving the image with Prewitt filters, the two directional components then being combined. This step can be slow, as it is not amenable to convolution type processing, although it may be approximated by $I_x + I_y$, which is suitable for convolution.

3. Perform a Directional Non-Maximum Supression perpendicular to the edges. This is explained in *Figure 2.17.* In the figure, G_1 interpolates $||\nabla I(x+1, y)||$ and $||\nabla I(x+1, y+1)||$ and G_2 interpolates in the other direction. We mark G_0 if

   (a) It is above a threshold.

   (b) $G_0 > G_1 \wedge G_0 > G_2$

Although obviously a local operation, the interpolation required in the above part of the algorithm is probably most performed most efficiently via look up tables. The remaining phase involves thresholding three times, and anding the results together (which could be done by convolution). Alternatively, much of the processing could be done using a logical convolver.

4. Perform contour extension with hysteresis on the results from the above stage. This links broken edge segments. Most of the calculations required in this and the final stages of processing are not suitable for convolution.

5. Thin contours

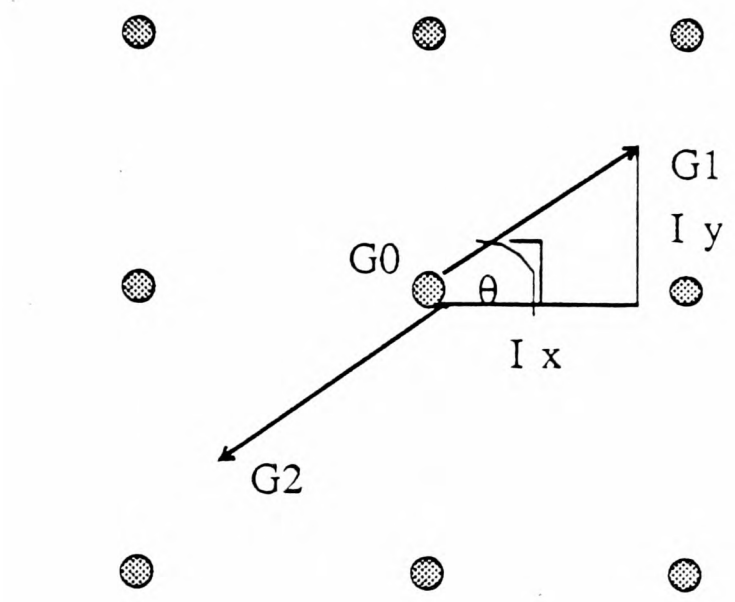
6. Combine the results from the differing scales.

The algorithm takes approximately 10^4 steps per pixel to perform, even using optimised techniques for the convolutions involved. As may be seen from the figure, the results achieved are impressive. Once again, the primary calculations in the algorithm may be performed by a combination of convolutions, thresholding and table look-up.

2.5.3 Lightness Computation

One of the early developments of *Global state from Local Interaction* was due to Horn [1974], in his work on the computation of lightness.

His work concentrated on "Mondrian" images, consisting of a number of patches each of uniform reflectance. His paper shows that lightness - a correlate of surface reflectance, may be computed, even under highly non-uniform illumination. Land and McCann [1971] supposed the colour image as composed from three separate channels, each of which is analysed as a monochrome image. Step changes in intensity are detected. By measuring the ratio of these changes, the lightness information can be extracted.



Directional Non-Maximum Supression

Figure 2.17: The figure illustrates Canny's non-maximum supression algorithm. G_1 interpolates $||\nabla I(x+1,y)||$ and $||\nabla I(x+1,y+1)||$ and G_2 interpolates in the other direction. We mark G_0 if it is above a threshold.

Horn works within the same framework, but provides a parallel algorithm for image reconstruction. Taking the image intensity p' as

$$\begin{aligned} p'(x,y) &= r'(x,y)s'(x,y) \\ r' &= \text{reflectance function} \\ s' &= \text{illumination function} \end{aligned}$$

Following on from Land's original idea, Horn sets $p = \log p'$, etc.

$$p(x,y) = r(x,y) + s(x,y) \tag{2.21}$$

Horn then applies a gradient operator L to the combined signal, to compute the lightness image

This works as follows:

- $\nabla^2 p$ finds the edges in the image, and produces a small response otherwise. As light in the image is assumed to vary slowly, and reflections to produce sharp changes in intensity, the detection of step edges produces a greater response for changes produced by reflection.

- Taking a threshold cuts out the component due to varying illumination, leaving only the edge responses.

- Taking the inverse Laplacian recovers the reflectance.

The function computed is thus:

$$l = L^{-1} \cdot T(Lp) \tag{2.22}$$

$$L = \nabla^2 \tag{2.23}$$

$$L^{-1} = (\nabla^2)^{-1} \tag{2.24}$$

$$T = \text{Threshold} \tag{2.25}$$

For a given function f and a fixed threshold λ:

$$(TF)(x,y) = f(x,y) \text{ if } |f(x,y)| > \lambda; 0 \text{ otherwise}$$

with the threshold removing all components except those due to reflectance.

$$T(\nabla^2 p) = \nabla^2 r$$

$$\Rightarrow (\nabla^2)^{-1} T \nabla^2 p = r \tag{2.26}$$

To summarise his method:

- Compute the Laplacian $F = \nabla^2 p$
- Threshold $E = T(F)$ to remove component due to varying illumination, giving only that due to reflectance.
- Recover reflectance by solving $\nabla^2 l = \nabla . E$ subject to $n . \nabla l = n . E$

L is applied as a local convolution over the image, where the convolution operator is an approximation to the Laplacian:

$$\begin{pmatrix} & \frac{-1}{6} & \\ \frac{-1}{6} & +1 & \frac{-1}{6} \\ & \frac{-1}{6} & \end{pmatrix}$$

Here we see an example of a Laplacian operator being applied via a simple local convolution to an image. The inverse operator is applied by an iterative process. Each iteration consists of applying the following convolution mask to the image, and then adding the input array to the output array.

$$\begin{pmatrix} & \frac{-1}{6} & \\ \frac{-1}{6} & +1 & \frac{-1}{6} \\ & \frac{-1}{6} & \end{pmatrix}$$

Thus, of the three steps in Horn's algorithm, the first and the third steps are both performed by way of convolutions, and the middle step is a simple thresholding operation.

Blake[1985] has refined Horn's work on lightness, taking Horn's equation 2.26:

$$(\nabla^2)^{-1} T(\nabla^2 p) = r$$

Blake then suggests replacing the use of the ∇^2 operator with the single differential

$$\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$$

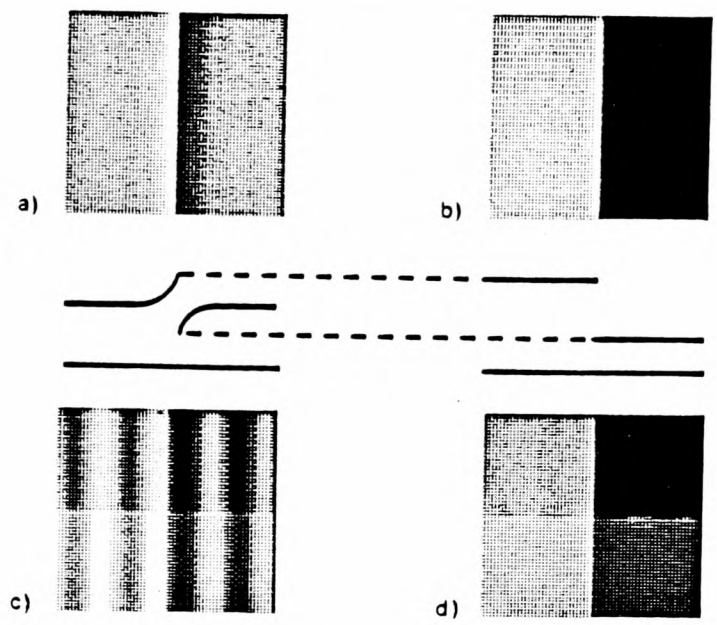


Figure 2.18: Results from Blake's Lightness Algorithm. The Craik-Cornsweet-O'Brien illusion (a) would be generated by the intensity whose logarithm is shown, with profile plotted below. The output of the computation(b) consists of two regions of uniform lightness, as expected. In (c) is a log-intensity profile generated by a piecewise constant reflectance with sinusoidally varying illumination; the resultant lightness (d) is piecewise constant-the sinusoidal variation has been supressed.

This means that you retain all of the information about p, up to an additive constant. Thus equation 2.23 is replaced by

$$\nabla l = E \tag{2.27}$$

$$E = T\nabla P \tag{2.28}$$

T is now applied to a vector field.

His algorithm also uses the convolution of a Laplacian mask with the image, and some results of the algorithm are shown in *Figure 2.18*. This proves to be yet another algorithm in which a significant part of the computational demand is in the form of local convolution support.

We will now leave lightness computation and go on to look at an algorithm for determining optical flow.

2.5.4 Optic Flow

Optical flow is the distribution of apparent velocities of movement of the brightness patterns in an image. It typically arises from the relative motion of objects in the

scene and the viewer (see *Figure 2.7*). Optic flow data can often be very useful for recovering motion and shape information from a series of images (see Scott's thesis [1986] for a recent use of such information). The particular method that we will discuss is the earliest by Horn and Schunck [1981].

They use the first order Taylor series approximation (see Nagel [1984])to the motion equation: (where E is the image intensity)

$$E(x+\delta x, t+\delta t) = E(x,t) + \underline{\nabla} E.\underline{\mu} + E_t + O(\delta^2)$$

Setting $E(x+\delta x, t+\delta t) = E(x,t)$ gives the motion constraint equation.

$$\underline{\nabla} E.\underline{\mu} = -E_t \tag{2.29}$$

here $\underline{\mu}$ is the distribution of velocities caused by smoothly changing brightness patterns, $\underline{u}$ is the unit normal in the direction of the gradient, and E is the image intensity. We thus have:

$$\underline{u}.\underline{\mu} = \frac{-E_t}{||\nabla E||} \tag{2.30}$$

Where $\underline{\mu}$ is the unit vector in the direction of the brightness gradient. We can see immediately that if $||\nabla E||$ is small, such as is the case at edges or textured surfaces in an image, then we can expect poor conditioning for the problem.

We also notice that only the component of $\underline{u}$ in the direction of $\underline{\mu}$ is determined. This gives the so-called "aperture" problem (see Figure 2.19). Hence $\underline{u}$ is undetermined. As usual $\underline{u}$ is found by adding a smoothness constraint, or regulariser, which says that flow components should be smooth and not contain discontinuities; this being based upon what we observe in the world (this ignores the edges of objects, where flow is not smooth). This is a reasonable assumption, except in cases where, for instance, an observer is watching a moving object within a moving frame of reference.

In order to impose this smoothness constraint, Horn and Schunck minimse the square of the magnitude of the gradient of the optical flow velocity, thus–

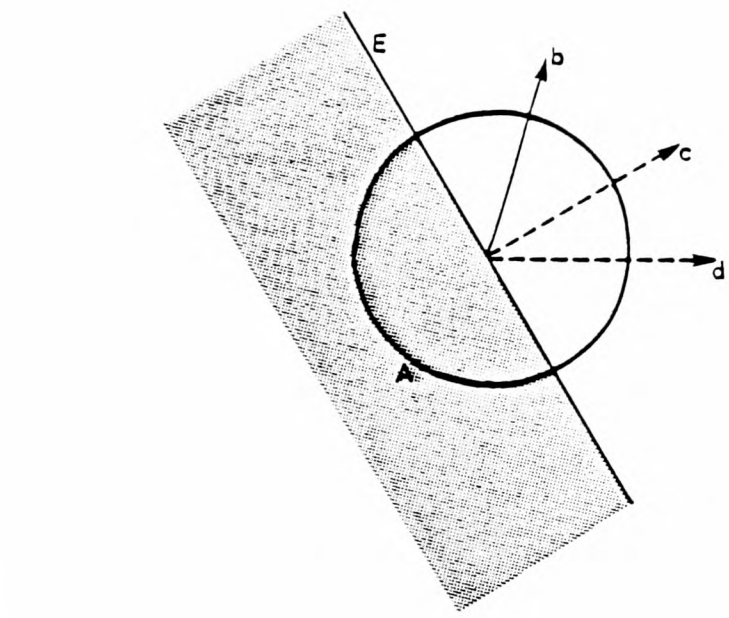


Figure 2.19: The Aperture Problem. An operation that views the moving edge E through the local aperture A can compute only the component of motion c in the direction perpendicular to the edge.

$$\left(\frac{\delta u}{\delta x}\right)^2 + \left(\frac{\delta u}{\delta y}\right)^2 \tag{2.31}$$

$$\text{and} \quad \left(\frac{\delta v}{\delta x}\right)^2 + \left(\frac{\delta v}{\delta y}\right)^2 \tag{2.32}$$

If we let $E_{i,j,k}$ be the pixel brightness at (i,j) in the k^{th} frame, then calculate, using simple convolutions:

1.
$$E_x = (1/4)\begin{pmatrix} -1 & 1 \\ -1 & 1 \end{pmatrix} * E_k + (1/4)\begin{pmatrix} -1 & 1 \\ -1 & 1 \end{pmatrix} * E_{k+1}$$

2.
$$E_y = (1/4)\begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix} * E_k + (1/4)\begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix} * E_{k+1}$$

3.
$$E_t = (1/4)\begin{pmatrix} -1 & -1 \\ -1 & -1 \end{pmatrix} * E_k + (1/4)\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} * E_{k+1}$$

4.
$$\Delta^2 u = 3\begin{pmatrix} 1/12 & 1/6 & 1/12 \\ 1/6 & -1 & 1/6 \\ 1/12 & 1/6 & 1/12 \end{pmatrix} * u \quad \text{etc}$$

The problem is to minimise the sum of errors in the equation for rate of change of image brightness

$$\epsilon_b = E_x u + E_y v + E_t \tag{2.33}$$

$$\epsilon_c^2 = \left(\frac{\delta u}{\delta x}\right)^2 + \left(\frac{\delta u}{\delta y}\right)^2 + \left(\frac{\delta v}{\delta x}\right)^2 + \left(\frac{\delta v}{\delta y}\right)^2 \tag{2.34}$$

and measure the departure from smoothness in velocity flow. To do this they let the total error to be minimised be:

$$\epsilon^2 = \int\int (\alpha^2 \epsilon_c^2 + \epsilon_b^2)\, dxdy.$$

Where α is a scaling constant. Now define $\overline{x}$ to be the local average of x.

$$\overline{x} = x * \begin{pmatrix} 1/12 & 1/6 & 1/12 \\ 1/6 & 0 & 1/6 \\ 1/12 & 1/6 & 1/12 \end{pmatrix}$$

they then proceed to compute $\underline{u}$ iteratively by:

$$u^{n+1} = \overline{u^n} - \frac{E_x[E_x\overline{u^n} + E_y\overline{v^n} + E_t]}{(\alpha^2 + E_x^2 + E_y^2)} \tag{2.35}$$

$$v^{n+1} = \overline{v^n} - \frac{E_y[E_x\overline{u^n} + E_y\overline{v^n} + E_t]}{(\alpha^2 + E_x^2 + E_y^2)} \tag{2.36}$$

In the above iterative scheme, the components E_x, E_y, E_t are all calculated by local convolutions as shown. The average values for u^n, v^n are also calculated with a 3×3 convolution mask, leaving just the squaring and dividing operations to be performed separately.

A similar approach is also used by Hildreth [1983] in her book on visual motion. We illustrate her work here, as it contains a vital component that cannot be calculated via a local operation.

She looks for an extra constraint to solve for the second component of velocity. She discusses a number of possible constraints, such as

- velocity constant over an area of the image

- velocity field consistent with rigid rotation and translation of objects in the image plane
- velocity field is smooth
- velocity field is constant over small time intervals.

She uses the motion constraint (see last algorithm) as her second constraint, saying that if $\|\nabla I\|$ is small, where I is the image then this implies that

$$\underline{\mu}.\underline{u} = -I_t/\|\nabla I\|$$

where $\mu(x, y)$ is the distribution of velocities caused by smoothly changing brightness patterns and $\underline{u}$ is the optic flow as previously defined. She traces the variation of μ along the contour. The problem here is to pool observations along a closed contour to compute a global value. Using values from two points along the contour at which $\underline{u}$ is different is reasonable because of Hildreth's initial motion constraint.

Hildreth bases her work on the following theorem: If $\underline{\mu}.\underline{u}$ is known along a contour, and there exist two points on the contour at which $\underline{u}$ is different, then there is a unique velocity field $\underline{\mu}$ that satisfies the known constraints on $\underline{\mu}.\underline{u}$ and minimises:

$$\oint \|\frac{\delta \underline{\mu}}{\delta s}\|^2 ds \tag{2.37}$$

Hildreth's scheme gives rise to the algorithm:

1. Compute zero crossing contours of $\nabla^2 G_\sigma * I = 0$ This may be performed via local convolution, as in previous algorithms.
2. Estimate equation 2.37 using $\sum_i \{(\mu_{x,i} - \mu_{x,i-1})^2 + (\mu_{y,i} - \mu_{y,i-1})^2\}$
3. Conjugate gradient descent algorithm
    - This is an iterative descent algorithm; a sequence of approximations $V^{(1)}, V^{(2)}, \cdots$ to the exact solution V is computed, given an initial approximation $V^{(0)}$; this sequence has the property that each new approx-

imation decreases the value of the objective function. The basic steps are

- (a) Start at an initial point $V^{(0)}$, which is usually $V^{(0)} = 0$.
  (b) According to a fixed rule, determine a direction of movement along the objective surface that will reduce the value of the objective function.
  (c) Move in this direction to a relative minimum of the objective function.
  (d) If the final solution has not yet been reached, return to step 2.
- One of the most common descent methods is that of *Steepest descent*, in which the direction of movement is given by the negative gradient of the objective function. This is slow to converge.
- She uses the following method where: Starting at any point $x_0 \in E^n$ define $D_0 = -g_0 = b - Qx_0$.

$$x_{k+1} = x_k + \alpha_k d_k \tag{2.38}$$

$$\alpha_k = -\frac{g_k^T d_k}{d_k^T Q d_k} \tag{2.39}$$

$$d_{k+1} = -g_{k+1} + \beta_k d_k \tag{2.40}$$

$$\beta_k = \frac{G_{k+1}^T Q d_k}{d_k^T Q d_k} \tag{2.41}$$

In Hildreth's algorithm, only step 1 may be calculated with local convolution support. The other two steps require global calculations. Thus although the algorithm attacks the same physical problem as Horn and Schunck's, the solution chosen is not suited to local computation.

2.5.5 Surface Reconstruction

After depth data is found from motion or shape from shading etc, the task then is to reconstruct the three dimensional surface from the depth data available. Unfortunately, the data available at this point in image reconstruction is often sparse,

and may typically give depth only at scattered points such as edges. This has been approached by Grimson [1981], and also more recently by Terzopoulos [1984]. Grimson's original algorithm worked well in reconstructing such surfaces, but was extremely slow to run, often taking as many as 200,000 iterations to converge!

Grimson used the calculus of variations to find a surface f which minimises the integral

$$\int\int (f_{xx}^2 + 2f_{xy}^2 + f_{yy}^2) dxdy \tag{2.42}$$

This expression is called "quadratic variation" (see Brady and Horn [1983] for a discussion of this type of expression).

Grimson's algorithm essentially operated as follows:

1. Determine a feasible initial surface s^0 passing through sparse data $c(i,j)$.

2. Once again, convolution is used to estimate $-\nabla s^n$ by convolving with

$$\begin{pmatrix} 0 & 0 & 2 & 0 & 0 \\ 0 & 4 & -16 & 4 & 0 \\ 2 & -16 & 40 & -16 & 2 \\ 0 & 4 & -16 & 4 & 0 \\ 0 & 0 & 2 & 0 & 0 \end{pmatrix} \tag{2.43}$$

The above is an approximation to equation 2.42. This is done by approximating to the second partial derivative in the x-direction as:

$$\frac{\partial^2 s(i,j)}{\partial^2 x} = \frac{1}{h^2}[s(i+1,j) - 2s(i,j) + s(i-1,j)] + O(h^2)$$

and similarly for the second partial derivatives in the y-direction and cross direction. The integral is then approximated by a discrete operation, leading to the mask above. It should be noted that this larger convolution mask may be developed by using a series of 3×3 computational molecules.

Then, in a similar way to Hildreth's conjugate gradient descent method, compute a scalar α^n locally

$$s^{n+1} = s^n - \alpha^n \nabla s^n$$

3. repeat step 2.

We see once again, that given the choice of initial surface s^0, the rest of the algorithm may be performed by local convolutions and additions.

Terzopoulos [1984] improved on this algorithm by using a multiresolution technique, which worked by stretching thin flexible plates to model the surface, these plates being "stretched over" the depth constraints and held in place with ideal springs. The problem is split into coarse, medium and fine levels, and surfaces reconstructed for all three. His method was designed for use on a pyramid architecture machine, ideally suited because of the differing scales used. Local convolution support plays the major role in his algorithm.

2.5.6 Stereo

In stereo, the basic problem is to recover depth data from a pair of stereo images taken of the same scene but from slightly different viewpoints. The method is as follows:

- Take two images separated by a baseline(see Figure 2.20)
- Identify corresponding points between the two images
- Use the inverse perspective transform (simple triangulation) to derive the two lines on which the world point lies.
- Intersect the lines to compute depth.

The hardest part of this process is the second step, finding matching points in the two images. One of the first stereo algorithms was provided by Marr and Poggio [1982]. They use two rules for matching points. These are:

1. Each point in an image may have only one depth value
2. A point is almost certain to have a depth value near to that of its neighbours

These "rules" seem reasonable based upon what we observe in the real world, although the second rule will obviously have significant exceptions.

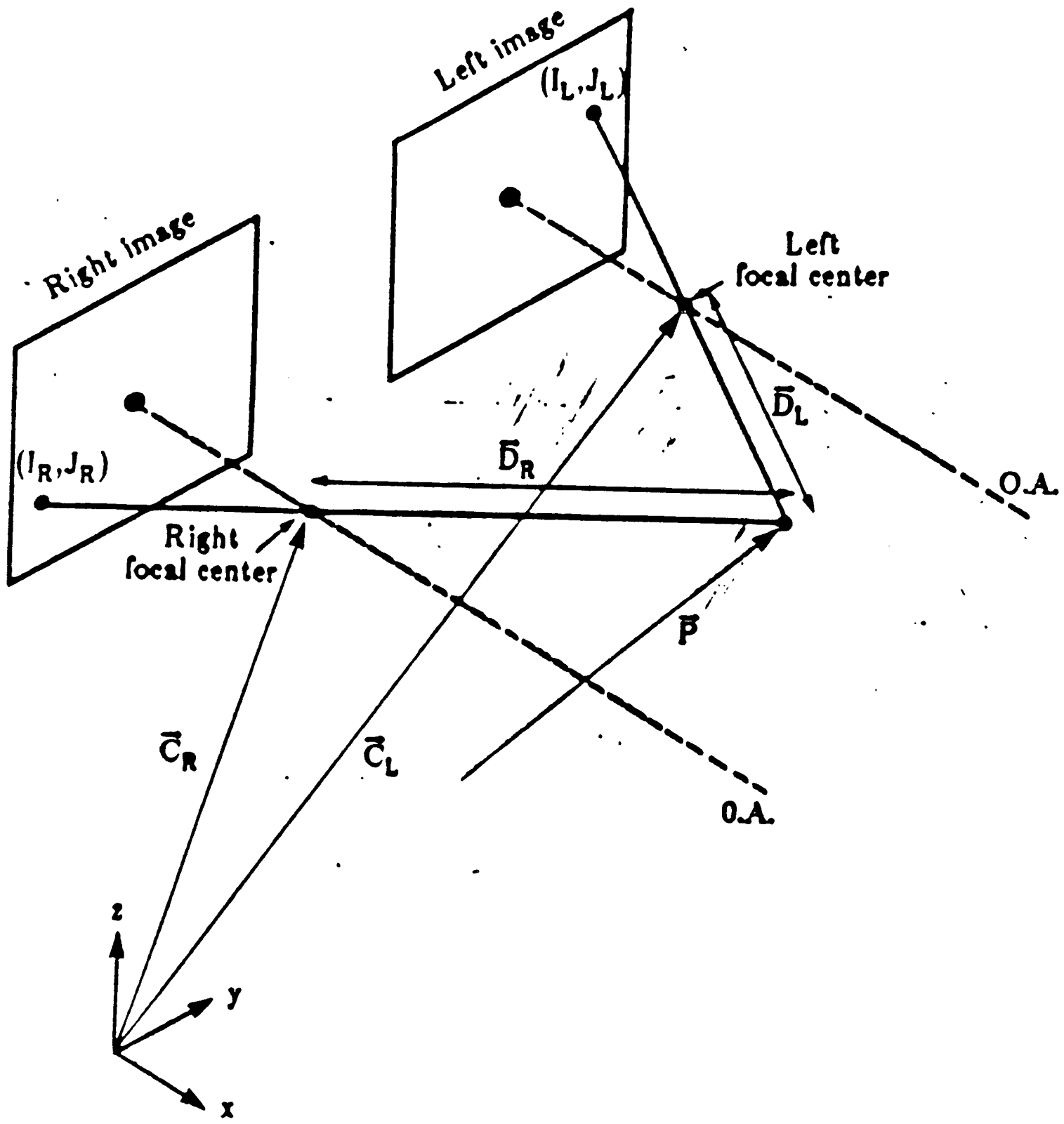


Figure 2.20: Two Images Separated by a Baseline

Defining a matrix $C(x, y, d)$ as a point x, y, d corresponding to a particular match between (x_1, y_1) in the right image and (x_2, y_2) in the left image, (see *Figure 2.21*) their algorithm proceeds in the following way:

Until C satisfies some convergence criterion, do

$$C_{n+1}(x,y,d) = \{ \sum_{x'y'd' \in S} (x'y'd') - \sum_{x'y'd' \in \theta} (x'y'd') + C_o(x,y,d) \} \tag{2.44}$$

Where the term in braces is handled as follows (2.45)

$$\{t\} = \begin{cases} 1, & \text{if } t > T = \text{threshold;} \\ 0, & \text{otherwise.} \end{cases}$$

$$S = \text{set of points } \{x'y'd'\} \text{such that } |x - x'| \leq \cdot 1 \text{ and } d = d' \tag{2.46}$$

$$\theta = \text{set of points } \{x'y'd'\} \text{such that } |x - x'| \leq 1 \text{ and } |d - d'| = 1 \tag{2.47}$$

The summations are performed over the immediate neighbours of each center pixel. Equal depth matches have reinforcing connections, which increase the value of the bracketed calculation. This algorithm has again produced impressive global results from the relatively simple local calculations, this time involving thresholding and summation.

Another stereo algorithm has been written by Drumheller and Poggio [1986]. Their algorithm is a development of Marr and Poggio's, which copes better with non-parallel surfaces, Marr and Poggio effectively preferring surfaces parallel to the image plane. This algorithm is similar to the one used by Pollard, Mayhew and Frisby [1985]. They use regularisation techniques in their algorithm and look for the disparity $d(x)$ between the two feature maps

$$L(x) \text{and } R(x + d(x))$$

They define an image dependent operator P_R by

$$P_R z(x) \rightarrow R(x + z(x))$$

The disparity function to be recovered is the solution to the inverse problem:

$$L(x) = P_R d(x)$$

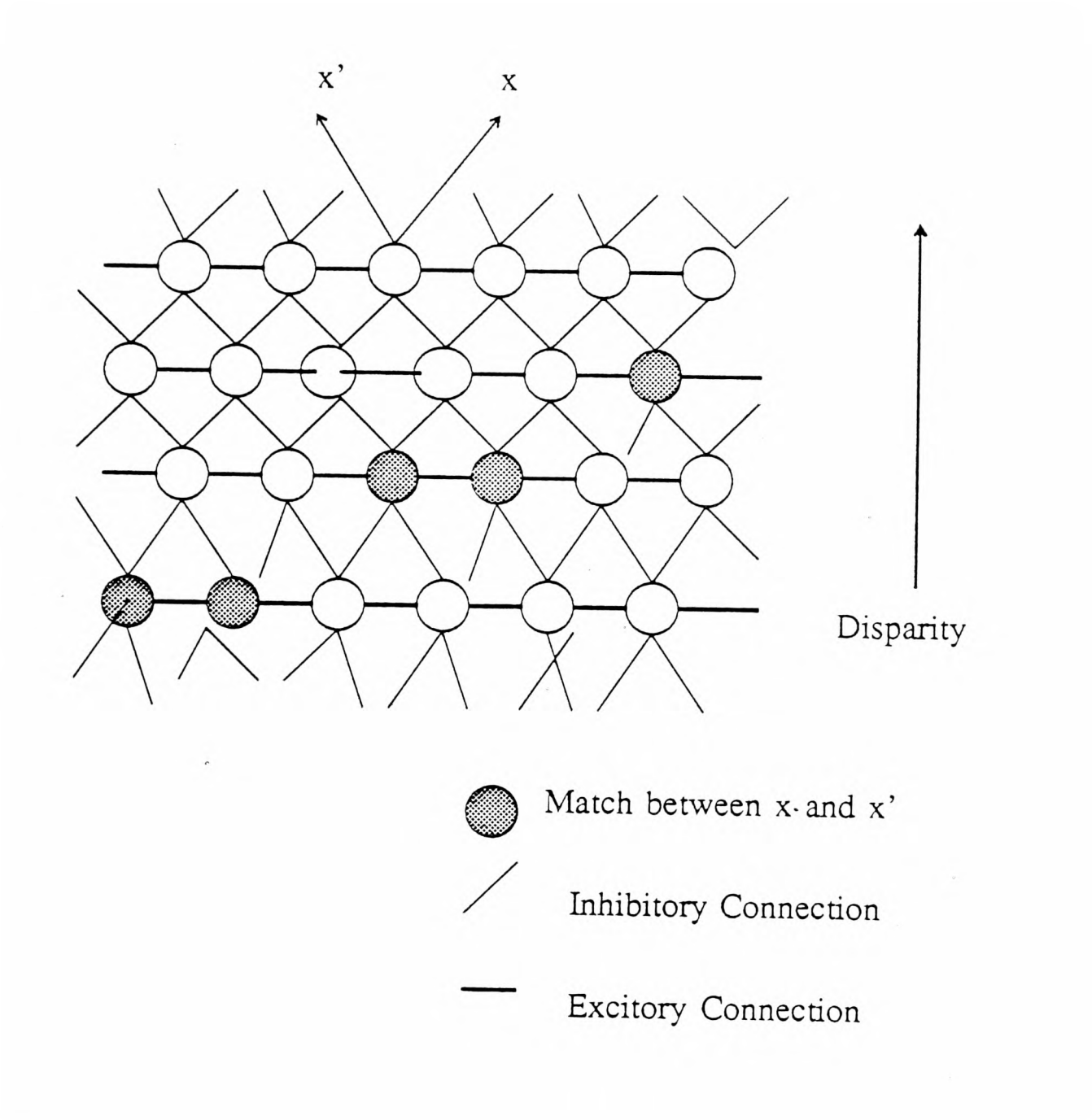


Figure 2.21: Disparity Matrix. Connections or alternative matches for a point inhibit each other, whilst matches at equal depth have connections that reinforce each other.

The natural way to determine $d(x)$ is to minimise

$$\|L(x) - R(x + d(x))\|^2 + \lambda\|\delta d(x)/\delta x\|^2$$

They have implemented their algorithm on the Connection Machine (see the next chapter) and have produced consistent results from local computation. See *Figure 2.22* for results. Their algorithm on the Connection machine runs as follows:

1. Compute features for matching
2. Compute potential matches
3. Determine the amount of local support for each potential match
4. Choose the correct matches on the basis of local support and constraints on uniqueness and ordering.

The features used are not fixed, but can be zero-crossings of a Laplacian filter, the sign of convolution output with a difference of Gaussians etc, such features all marking potential edges, and all suited for local convolutional computation, as we have seen. Potential matches are allowed to occur between two zero crossings of the same sign. The algorithm uses Marr and Poggio's compatability constraint, saying that neighbouring points are likely to have similar disparity values. Thus on the Connection Machine the final algorithm is as follows:

For D disparity values ranging between d_i and d_f:

1. Allocate field P = (paddr,D) to contain the set of potential matches. paddr is the address of the match. Initialise to zero.
2. Allocate two 2-bit fields L and R to contain zero crossings of the two images.
3. Shift R over L along the x-axis (this assumes that epipolars lie on scan lines) one pixel at a time from $x = d_i$ to $x = d_f$. after the I^{th} shift, write a 1 into the 1-bit field ((paddr $+i$),1) at each x, y where L and R contain identical zero-crossings

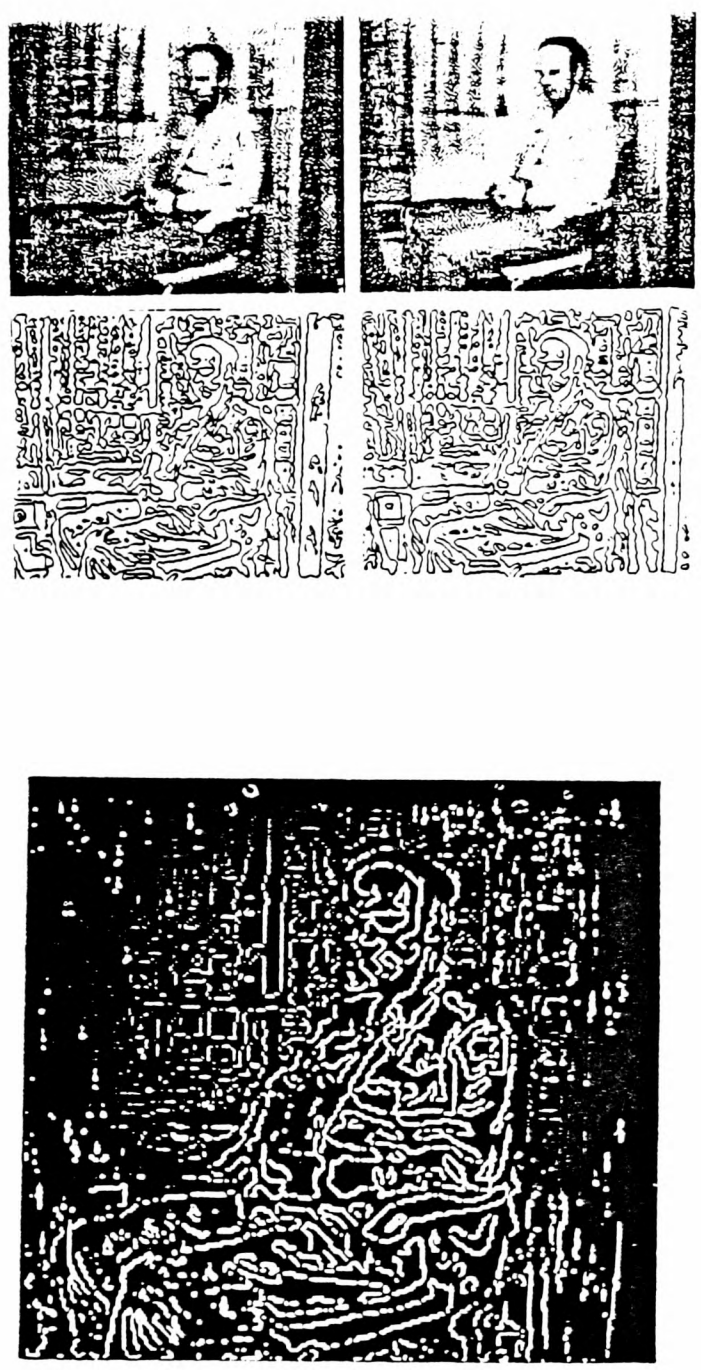


Figure 2.22: Results from Poggio and Drumheller's Stereo Algorithm. The figure shows a natural stereo pair of images, subsampled at 256×256 pixels. The images are smoothed with a gaussian (central width 1.5 pixels). The final image is the disparity map calculated for the pair.

Having calculated matching features with local convolutions, the next three stages use threshold and summation operations as in Marr and Poggio's algorithm. After the above three stages they gather local support for the results. This is done by using a three dimensional convolution in x-y-d space using a support function (after Marr and Poggio). As is becoming familiar, a complex calculation of global state is performed largely using local convolutions. Finally they perform a non-maximum supression on the results of this convolution.

2.5.7 Stochastic Relaxation

As mentioned when we discussed non-convex vs. convex parameter spaces, some vision algorithms are not suitable for local convolutional support, because the solution space to the problem is non-convex. We illustrate one example of this which uses stochastic relaxation to solve a problem.

Geman and Geman [1984] have developed an algorithm which works on a non-convex parameter space. They attempt to remove noise from an image in the following way. They form a stochastic model for the image based on a Gibbs distribution. The stochastic model assumes that noise in an image is statistically distributed, e.g. in a Gibbs distribution. Using a Markov condition for the image implies that the values of individual pixels depend only upon those in a local neighbourhood. They then prove that this Markov assumption on the stochastic model implies that the noise is distributed with a Gibbs distribution. Their restoration algorithm then depends on simulated annealing, a form of stochastic relaxation. Stochastic relaxation is different from deterministic relaxation where a sequence $I_1, I_2, \cdots$ of images is such that there is a decrease of an objective function along the series. In stochastic relaxation, the objective function can increase, as the changes are done randomly. This is one way of converging to a solution in a non-convex parameter space.

Their stochastic algorithm proceeds as follows:

- A local change is made in the image based upon current values of pixels

and boundary cells in the pixel neighbourhood. The change is random; it is generated by sampling from a local conditional probability distribution function.

- The local conditional probabilities depend on a global control parameter T; at low T states tend to increase the objective; at high T the function is essentially uniform.

- Their process starts at high T where stochastic changes tend to increase the objective function. Gradually, T is lowered.

$$\begin{aligned} \text{Take an image} \qquad & F \text{ to be recovered} \\ \text{Degrade this image to give } G \quad &= \quad \phi(H(F)) + N \\ \text{where } H \quad &= \quad \text{Blur function} \\ \phi \quad &= \quad \text{Transform} \\ N \quad &= \quad \text{Noise} \end{aligned}$$

Geman and Geman give a Markov condition on the above function: i.e. $\underline{F}$ is modelled as a Markov Random Field. This MRF-Gibbs distribution may be shown as follows:

$$\begin{aligned} & P(X_s = x_s | X_r = x_r, r \neq s) \\ = \quad & P(X_s = x_s | X_r = x_r, r \in \mathcal{G}_o) \end{aligned} \tag{2.48}$$

The conditional probability distribution assigns values to pixels $I_j(x, y)$ computed in the local neighbourhood. They prove that this local conditional probability distribution can be modelled as a Gibbs function, saying that $\mathcal{X}$ is an MRF with respect to a local neighbourhood $\mathcal{G}$ if and only if

$$\pi(w) = P(\mathcal{X} = w)$$

They define

$$P(X_s = x_s | X_r = x_r, r \neq s) = Z_s^{-1} \exp\{-\frac{1}{T} \sum_{C:s \in C} V_C(w)\} \tag{2.49}$$

$$\text{Where } Z_s \doteq \sum_{x \in \Lambda} \exp\{-\frac{1}{T} \sum_{C:s \in C} V_C(w^x)\} \tag{2.50}$$

Here the V_C's are potentials defined as (in a 3×3 neighbourhood).

$$V_C(f) = \begin{cases} \frac{1}{3}, & f_s = f_r; \\ \frac{-1}{3} & f_s \neq f_r \end{cases} \tag{2.51}$$

This takes f_s, f_r as pairs in a 3×3 neighbourhood. We show an example of this in *Figure 2.23.*

Their stochastic relaxation algorithm proceeds as follows:

- Given the state of the system at time $t : x(t)$
- Randomly choose another configuration ψ
- Compute energy change $\nabla\epsilon = \epsilon(\zeta) - \epsilon(X(t))$
- Calculate

$$q = \frac{\Pi(\zeta)}{\Pi(X(t))} = e^{-\beta \nabla \epsilon}$$

Then choose $0 \leq \zeta \leq 1$ uniformly. Set $X(t+1) = \zeta$ if $\zeta \leq q$ and $X(t+1) = X(t)$ if $\zeta > q$.

In the above:

$$\beta = \frac{1}{KT}$$

and

$$T(k) = \frac{C}{\log(1+K)}$$

where

$$1 \leq \quad k \quad \leq K$$

$$k \quad = \quad k^{th} \text{ iteration}$$

$$C \quad = \quad 3.0 \text{ or } 4.0$$

1	1	1
1	1	1
1	1	1

Potential is $V_C = \frac{8}{3}$.

1	1	1
0	1	1
0	0	1

Potential is $V_C = \frac{2}{3}$.

Figure 2.23: In Geman and Geman's algorithm, the potentials V_C are illutrated by the two examples above, assigning $\frac{1}{3}$ to the potential if neighbours are equal and $-\frac{1}{3}$ otherwise.

In practice, they pick $0 \leq \varsigma \leq 1$.

$$X(t+1) = \begin{cases} \varsigma & \text{if } q \geq \varsigma \\ X(t) & \text{otherwise} \end{cases}$$

This process can be performed by something akin to local convolution. The following definitions are also used:

$$\prod(w) = \frac{1}{Z} e^{-U(w)/T} \tag{2.52}$$

$$U(w) = \sum_{C \in \mathcal{C}} V_C(w) \tag{2.53}$$

U in the above is an energy function, used in the Gibbs distribution.

In addition, they use a line following process L, where they take all eight pairs of pixels surrounding pic(i,j) and add $\frac{1}{3}$ to the score V_C if the intensities are equal and $\frac{-1}{3}$ if different. These V_C functions represent contributions to the total energy of the system from external fields, pair interactions etc, just as in statistical physics. In the line process;

$$U(\underline{f}, \underline{l}) = \underline{U}(\underline{l}) + U(\underline{f}|\underline{l})$$

$U(\underline{l})$ is given in *Figure 2.24*.

$U(\underline{f}|\underline{l})$ is defined based upon a line site d; see *Figure 2.25*. If $L_d = 1$ then $V_{r,s}(f_r, f_s) = 0$, else $V_{r,s}(f_r, f_s) = \pm 1$ dependent upon $f_r = f_s$. Some results of their algorithm are shown in *Figure 2.26*.

Blake has also discussed Graduated Non Convexity with Zisserman (Blake and Zisserman [1987]). This is one technique for finding the minimum value of the cost function F in algorithms such as Geman and Geman's, where the cost function is in a non-convex parameter space. The method works by approximating a new cost

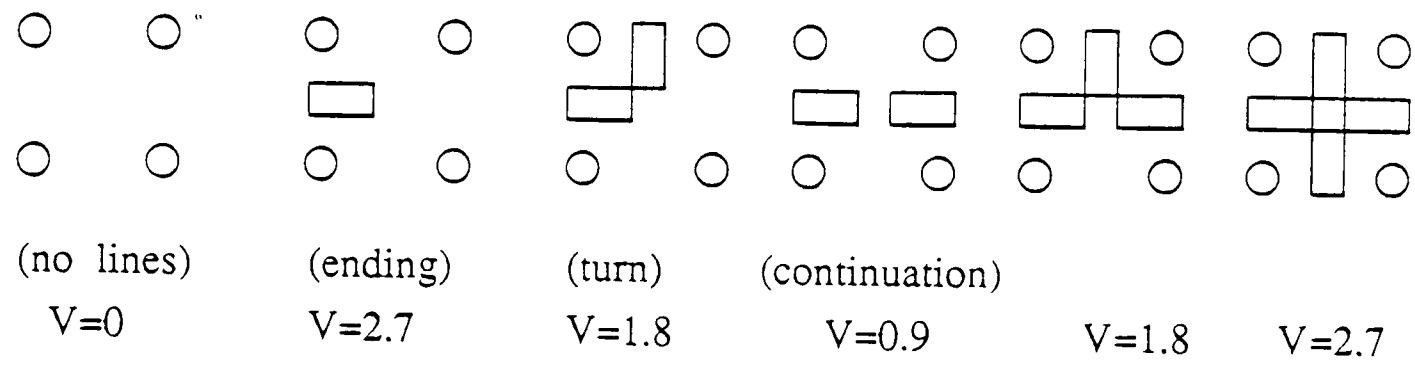


Figure 2.24: Definition of $U(\underline{l})$. For $U(l)$, only cliques of size four are non-zero, of which there are six distinct types up to rotations. These are shown above with their associated energy values.

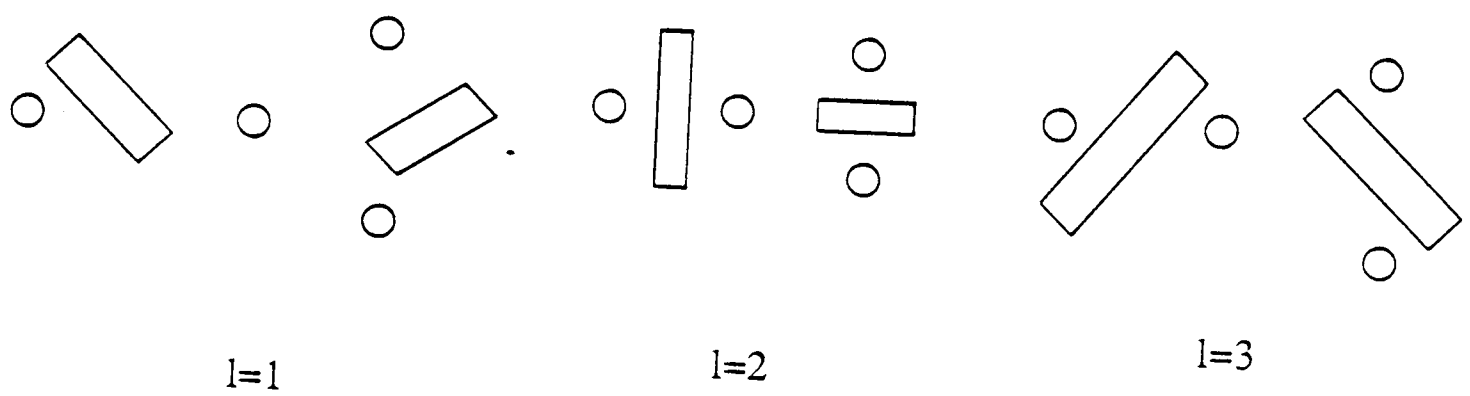
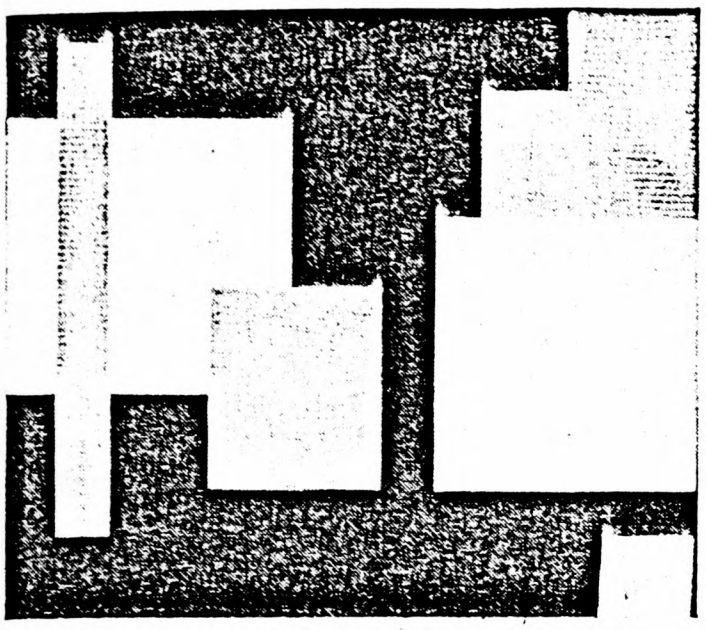


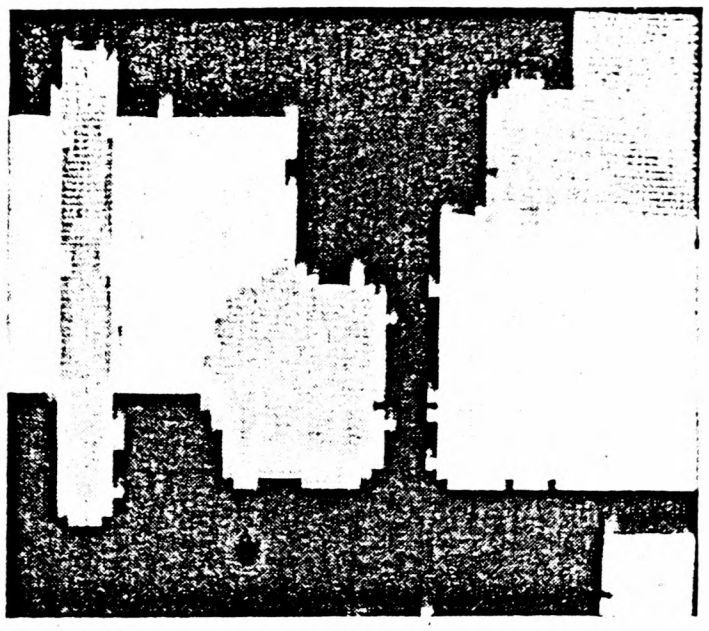
Figure 2.25: Line site. Line sites for $l = 1, 2, 3$ are shown. There are four possible states corresponding to "off" ($l = 0$), and three directions.

function F^* to the original function F, where the new cost function is convex, and hence can have only one local minimum, which must also be a global minimum. Descent on this cost function must end up at this minimum, which may also be a minimum of the original function F. A simple test is used to see if this has succeeded; if not, a slightly modified approximation F^* is used, and the method is repeated.

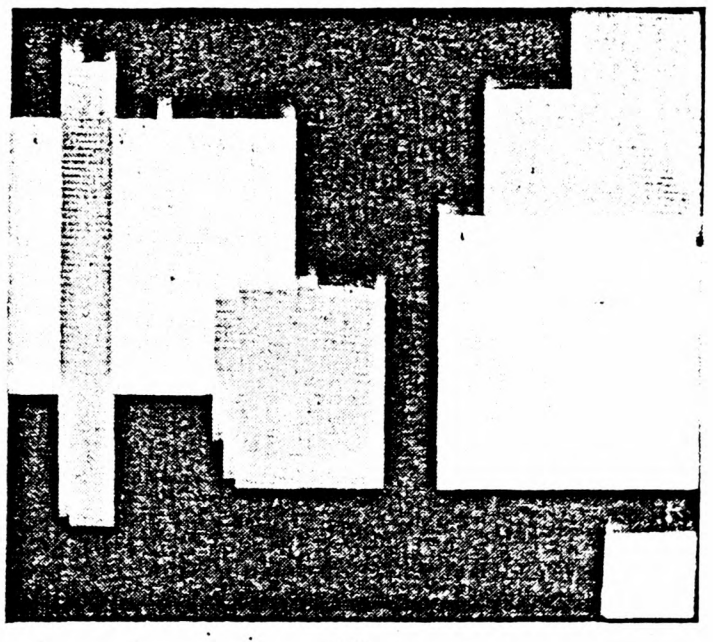


(a)

(b)



(c)



(d)

Figure 2.26: Results from Geman and Geman. (a) shows an original generated image. (b) shows the image degraded with additive noise. (c) shows the restored image after 1000 iterations, and (d) shows the restored image after 1000 iterations and the line process.

2.6 Chapter Summary

The discussion in this chapter showed that simple convolution type operations are not, as is often seen, an end in themselves, suitable only for low level algorithms such as Sobel edge filters, but surprisingly form a generic operation for intermediate level algorithms. This type of algorithm, which is at the moment largely limited to laboratory use, could be executed at much greater speed by the realisation of the role that simple functions such as convolution and thresholding play in their calculation.

We thus see an increasingly strong argument in favour of using highly parallel processing array structures for image processing, not only for the "early" stages of vision such as image enhancement and edge detection, but also for the later stages such as shape detection and stereo. We may summarise this section as follows.

- The aim is to reduce the amount of data in the picture.
- This may be done via local operations which, based on variational techniques, generate a global image state from local parameter interaction.
- Parallelism is a useful way to do this.

We have discussed intermediate level vision algorithms, and have looked briefly at some of the low level algorithms which provide the current industrial basis for machine vision. We will now proceed to discuss some of the computers which have been designed to make the real-time execution of these algorithms a possibility.

Chapter 3

CURRENT ARRAY PROCESSORS FOR VISION

3.1 Introduction

In the early days of machine vision, images were analysed on serial computers. Because of the very high computational demand made by vision algorithms the only way of providing a remotely reasonable performance was to restrict the computation to simple algorithms and binary images, or to completely abandon any hope of real time (video rate) processing. Even so, the performance of algorithms was very poor, both in the time taken to process an image and in the results produced.

Starting with an image that is $n \times n$ pixels spatially, it takes a serial computer $O(n^2)$ time to process such an image, i.e. the time taken is:

$$\lambda n^2 + \mu n + \gamma$$

for some constraints λ, μ, γ.

Restricting algorithms to operating on binary images, or simplifying them reduces λ, but not n, thus making little difference to the processing time. For this reason, early vision algorithms typically reduced n as well, in an attempt to decrease processing time, often decreasing the image size to 32×32 or even 16×16 pixels. For some applications, such as optical character recognition, this was acceptable, but in most situations much higher resolution is required.

The conclusion reached by vision researchers was that serial computers are inap-

propriate for vision processing. Research laboratories explored other architectures. For a number of years, vector processors had been used to speed the mathematical processing on serial computers. Similarly, it was proposed to develop array processors for vision. This idea embodied the following concepts:

- Use one processor per *small* number of pixels.
- Simple interconnections between processors, to allow local communication to support the type of convolution algorithm that we have seen in many of the algorithms discussed in the last chapter. Complex interconnectivity was not used because of hardware construction difficulties and because no need for them was seen.
- Make the machines **S**ingle **I**nstruction **M**ultiple **D**ata types (SIMD), where all the processors execute identical instructions in lock step. This restricts the machine to operating as above with local support operations, rather than allowing totally independent pixel processing. However, as we noted in the last chapter, a surprising range of vision processes can be reduced to local support operations suited to a SIMD computer.
- Have limited storage with each processor, to save memory access time.

Ideally, an array architecture for vision has one processor per pixel in the image. These processors are designed into I.C's,ideally with all of the processors on one chip, and with sufficient storage. However, the hardware cost of such a design is excessive. Crucially, even if the hardware cost is ignored, other problems arise with input/output to the chip(s). Essentially, this is because chips only communicate via their edges. Thus there may be $O(n^2)$ processors on a chip, but the chip perimeter, and hence I/O capability is only $O(n)$. Therefore, to fill n^2 processing elements with data takes $O(n)$ time, through $O(n)$ border points (assuming that the chip design allows such a pin-out). This complexity problem also affects networks of processor

chips, where the PE's on individual chips may have better communications than with those on other chips.

We will expand on this theme later in this thesis. The choice of a certain type of computer as being "optimal" for vision processing now presents the hardware designers with another problem. At the moment there appears to be no technological solution to the communications problem; two-dimensional I/O, or integrating processing with sensing are obvious approaches, but are difficult to achieve practically. However, in the late 1960's when this theory developed, the I/O problem was not crucial, as large performance gains over serial computers could be achieved, even with limited I/O.

The development of such array computers began with the design of the ILLIAC IV, whose architecture was based on a paper by Slotnick et al [1960] entitled the SOLOMON computer. This was meant to be a two dimensional array of 32×32 processing elements. The use of a small image size, coupled with parallel execution, was intended to decrease the processing time. Although the SOLOMON was never actually built, its architecture inspired not only the ILLIAC, but also the Burroughs PEPE computer, and many subsequent machines. The ILLIAC IV was not actually built until 1975 (see Falk, [1976]) and never achieved its performance goals, as its architecture was ahead of the technology available at the time. Unfortunately, by the time that the technology had increased to the stage when the design could actually be built, the architecture was out of date.

The original SOLOMON architecture also provided the inspiration for a number of other machines, which are similar in their overall design. The first of these was the ICL DAP (Distributed Array Processor) project that was started in 1972 and commissioned in 1976. The DAP together with the Goodyear MPP (Massively Parallel Processor) and the University College, London CLIP (Cellular Logic Image Processor) machines are perhaps the most widely known of the current array machines, and we discuss these architectures below, as all have been extensively

used for vision research. Both the CLIP and the MPP operate on a bit-serial/word-parallel basis. This architecture was suggested by Shooman [1960]. He argued that for two dimensional arrays of data (effectively three dimensional when the grey-level is counted as depth) it was more efficient to operate on a bit-slice, taking one bit from each word, rather than on the usual word-slice, taking all of the bits in parallel. This concept has been widely used in array machines, (see, for instance, the recent Disarry machine, Page [1983]), as it is easier to design a processing element to operate on a single bit rather than on a complete word. It also led to the idea of bit-slice processing, where a microprocessor is produced in, say, 4-bit widths, which may be chained together to produce the correct word length for any particular architecture. This type of design is used later in this thesis.

3.2 Three Array Computers

Although there now are many other array processors including the GAPP (Hannaway [1984]), and the GOP (Granlund [1983]), we will focus on just three designs. Currently, these designs are being overtaken in performance terms by simple architectures with more up to date technology, such as those on offer by companies such as Datacube, CRS, Imaging Technology, and Vicom, but the three we discuss chart the progress of vision processing. The three we consider in detail are the CLIP, DAP and MPP architectures, and will use these later in the thesis as benchmarks for performance calculations.

3.2.1 UCL CLIP

The CLIP (Cellular Logic Image Processor) series of machines have been developed at University College, London under the direction of Fountain (see Duff, [1976] and Fountain, [1981]). The CLIP machines have steadily increasing performance and array size. The CLIP 4 processor (recently released commercially) is an array of 96×96 bit serial processors (*Figure 3.1*). It is a SIMD architecture, the images being

processed in a bit-serial manner. Data is loaded by columns into the processors, which operate on a 96×96 pixel subwindow of the image. Programming the CLIP is done in microcode, with three basic instructions, instruction words being stored externally to the array, and applied at 1μS intervals. but a high level interface is available.

Each processing element in the 96×96 array has a two bit input boolean processor and adder. Input gating to the processor allows the selection of input data. Each processing element has a 1×32 bit memory, and there is programmable 4-, 6-, or 8-way interconnectivity between processors. The machine is constructed using LSI NMOS IC's, each chip containing a 4×2 array of processors. Edge connections are then carried over between the chips. The design has been used for vision applications at UCL, with basic logical operations and algorithms for thinning, median filtering, averaging and edge connection being implemented as tests, but is limited by the small processing element (PE) memory size of 32 bits per PE.

CLIP 6 was constructed as a 6 bit wide version of CLIP 4, to allow parallel grey level processing. It is approximately 5-10 times faster than CLIP 4. The PE memory size is increased to 6×64 bits, with PE's able to perform basic arithmetic and logical operations under microcode control. The device is constructed from MSI/LSI TTL components. Work on more advanced versions of CLIP offering increased processing power continues at UCL, and the CLIP 7 has recently been announced.

3.2.2 ICL/AMT DAP

The Distributed Array Processor was developed in the mid-1970's by ICL (see Reddaway, [1973]). It was not originally intended primarily for picture processing but rather as a general purpose computational machine. The architecture is shown in *Figure 3.2.* The DAP is installed within the memory of a host mainframe computer and functions as an intelligent memory with distributed logic. I/O from the array is memory mapped to the host computer via a set of registers, allowing data to be

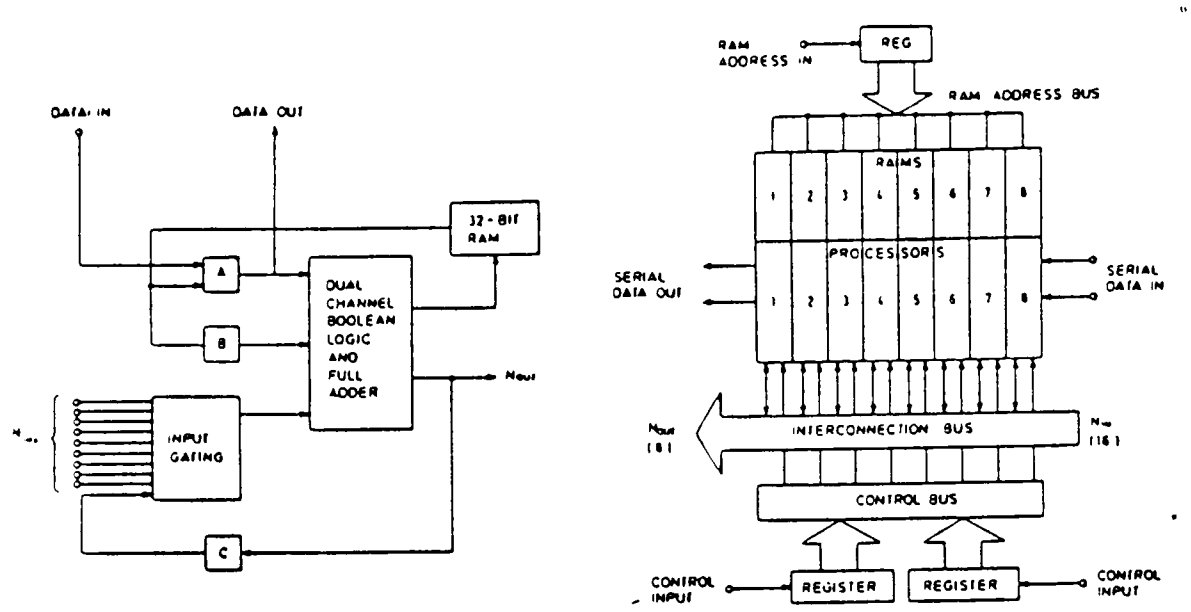


Figure 3.1: Processor Survey: The CLIP4 Processor and chip configuration. This was one of the earliest bit-slice processor designs, and has higher than usual 8-way local interconnectivity. It consists of a 96×96 array of processing elements.

accessed either word or bit-wise. Data may be transferred to the device in row-parallel fashion (this is discussed in the next chapter), much as in the CLIP, but some performance gains are made by doing this transfer through a DMA channel, as the machine is memory mapped to its host.

Other than the interface method to the host, the DAP is similar in principle to the CLIP. It is a SIMD array device, again operating in a bit-serial manner on input data. However, the array size is not as large as on the CLIP, with a 64×64 array of bit serial full-adder processing elements, with four-way interconnectivity and 4096 bits of memory per PE. This is more memory per processing element than on the CLIP, but the interconnectivity is restricted in comparison. In addition, each processing element consists simply of a full adder and output/input multiplexers, without the boolean processor that is found on the CLIP. This can limit the performance of the machine.

The DAP is programmed in DAP-FORTRAN, (see Parkinson [1983]), which is an array extension to FORTRAN, with limitations to make its implementation on the DAP efficient. This introduces vector and array data modes to the FOR-

TRAN language, and extends the scope of the available functions. The DAP has been used for vision work by Queen Mary College, London, on algorithms such as structure from motion (Buxton and Murray [1985]), and optic flow (Buxton and Buxton [1984]). However, it has the disadvantage that it must be used with an ICL mainframe as host. As the original design was also not specifically intended for vision work, there was no direct interface to camera or monitor, which can obviously prove a significant disadvantage for vision processing. However, the DAP is now being marketed by a new company AMT, which can supply a quite extensive library of vision algorithms, and fast fourier transform routines available for the machine, which increase its usefulness.

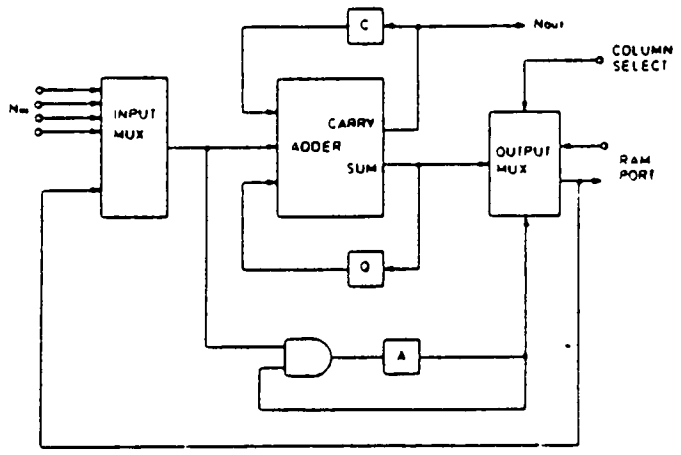


Figure 3.2: Processor Survey: The DAP processor. Originally intended as a general purpose computing machine to attach to an ICL mainframe, its applications in vision were soon realised. Based on a 64×64 bit slice array.

3.2.3 Goodyear MPP

The Massively Parallel Processor (Batcher, [1980], [1982]) was commissioned by NASA from Goodyear Aerospace. Like the CLIP and the DAP, the MPP is a bit-serial SIMD processor. As the name suggests, it was designed with a larger processing array size than the other machines, and has an array which contains 16384 processors arranged as a 128×128 array, with an extra 128×4 rectangle of PE's that is used to reconfigure the unit when a fault is detected. A diagram of the

computer is shown in *Figure 3.3* . The processing elements are bit-serial processors with a clock rate of 10MHz. The PE's are 4-way connected, with wrap-around facilities at the edges of the array. This wrap-around may be done in different ways to facilitate processing boundary pixels at the edges of an image. The MPP has been designed to provide the sort of local convolution support that we have discussed, with the available technology allowing a larger array size than was possible with the earlier CLIP and DAP designs.

Each processing element has six one-bit registers, a shift register with programmable length, a full adder , and has access to 1024 bits of memory (expandable to 65536 bits/PE). The PE's are packaged in 2×4 subarrays on a VLSI CMOS/SOS chip. RAM is not included on the chip but is packaged separately. Twenty four of these chips make up a board in a 16×12 array, with 96 boards making up an array unit. Thus the unit is a substantial size and has complex interwiring, even with only the simple 4-way local interconnections provided. However, there is a military version of the MPP which is small enough to be used as an airborne radar processor.

The MPP uses a VAX-11/780 as a host computer via a unibus. The chip uses column-parallel I/O which is overlapped with processing (see section 4.1.3) to increase speed. This means that the array can input data whilst it is processing previous data, the new data being stored temporarily in the PE's 's' input registers. This leads to an increase in speed over the other two machines, which have to stop processing before I/O can take place. However, there is no facility for grabbing frames from a camera, or for outputting the contents of the array to a monitor. In functions such as local convolutions, where multiplication is required, the MPP can be approximately sixteen times faster than the DAP. This is due in part to the four times greater array size, but also to the faster clock cycle, and to the shift register in each PE which can speed operations. This extra speed has seen the MPP being used to process satellite data where it can offer significant performance gains over

a serial computer.

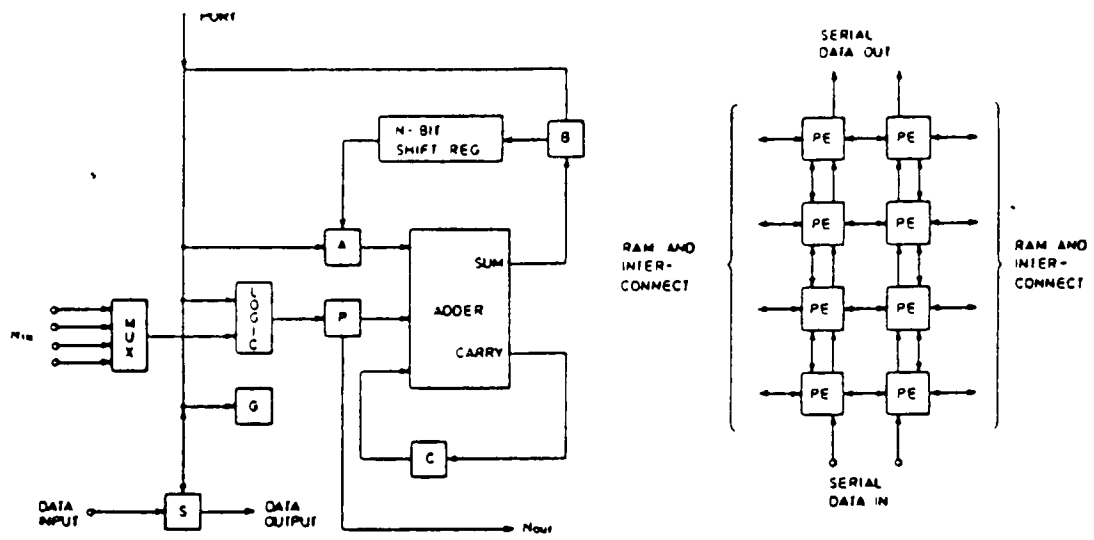


Figure 3.3: Processor Survey: The MPP processor and chip configuration. One of the largest bit-slice processors with a 128×128 PE array, its main problem for vision being data I/O.

3.3 Three Unconventional Vision Processors.

The CLIP, DAP and MPP represent established array architectures. We now review three other architectures, which although built with the same goals, have taken a different approach to provide local support operations.

3.3.1 The Cytocomputer.

The Cytocomputer, Sternberg [1978] was one of the first examples of a serial array computer or, as they are often called, a pipeline computer. It was designed, as the "cyto" in the name suggests, for the automatic inspection and screening of human cells. The computer contains two separate pipeline chains, one of which is eighty elements long and processes binary data, and a twenty five stage grey level pipeline. The architecture is illustrated in *Figure 3.4* . The binary stages can perform a binary function on the neighbours of one of the inputs and combine the function with the other 7 bits. Table look up memories are used to provide all functions.

Within each stage is a linked array of registers through which the incoming data is clocked. A 3×3 subarray of these registers is then connected to a neighbourhood logic unit, providing a convolution type operation. The device differs from the three array computers mentioned not only in being a pipeline rather than an array, but also in being a MIMD machine, where the different data in each pipeline stage at any one time may have different operations applied to it.

The Cytocomputer is based on algorithms that use morphological operations (see Serra [1982]). This mathematical technique treats images as sets, and image transformations as binary relationships between these sets. The basic operation is called *erosion*. In the case of binary images, erosion is a neighbourhood transformation in which each pixel is replaced by the logical and of preselected neighbours. Denoting a 3×3 neighbourhood by W and a subset of preselected window pixels by B, then the erosion of the binary image A by the structuring element B is given by:

$$A(-)B = \text{intersect}_{b \in B}(A + B)$$

Likewise, a *dilation* is defined as:

$$A(+)B = \text{union}_{b \in B}(A + B)$$

When used with greyscale images, we denote the grey levels of points j of the greyscale structuring element Bk as $Bk(j)$, the grey levels of the eroded image A are given by:

$$A(-)Bk = \text{minimum}[A(x - j) - Bk(j)]$$

Dilation is defined in similar fashion.

Haralick and Sternberg [1986] have recently discussed some uses of greyscale morphology, and have defined its relationship to binary morphology. They show that greyscale dilation can be accomplished by taking the maximum of a set of sums. Hence it has the same complexity as convolution. Likewise, greyscale erosion has the form of correlation, with the summation of the correlation replaced by the

minimum operation, and the product of the correlation replaced by a subtraction operation.

Using the two basic operations of erosion and dilation, algorithms are coded in suitable format and are then executed on the Cytocomputer. However, its restriction to morphological operations does make the Cytocomputer more difficult to program than the array machines mentioned, and limits severely its potential range of applications. An LSI version of the machine is currently being developed which will contain up to 550 stages, with each stage on a special LSI CMOS/SOS chip. Each stage will be able to process both binary and grey level data. The performance of this machine is compared with that of the three array machines mentioned in section 6.

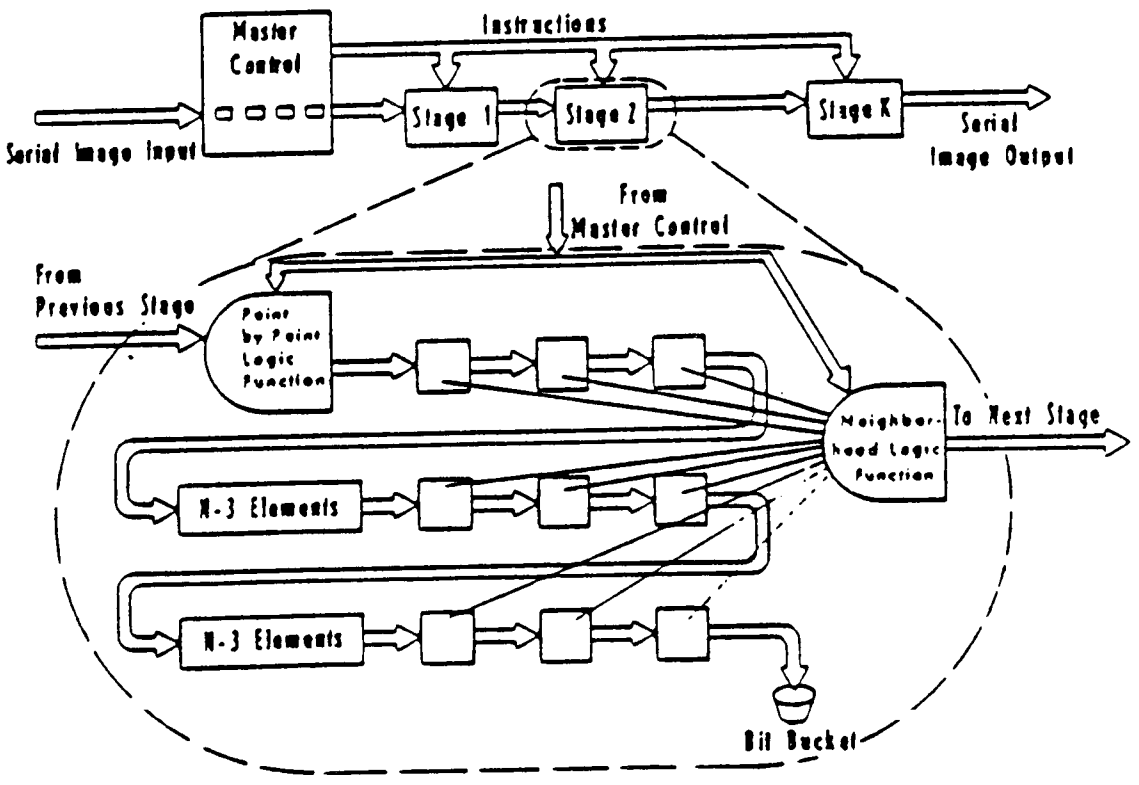


Figure 4. Cytocomputer block diagram

Figure 3.4: Processor Survey: The Cytocomputer. One of the first pipelined computers, with an 80-stage binary pipe and a 25-stage grey scale (8-bit) pipe. Designed specifically for vision processing, its main drawback is its reliance on morphological operations.

3.3.2 NBS PIPE Processor

The PIPE processor was designed by the American National Bureau of Standards as an image preprocessor (Kent [1984]). As with the other machines discussed in this chapter it is designed to act as a front end processor to provide low level local vision support. Like the Cytocomputer, the PIPE is pipelined. The number of stages in the pipeline is variable. A diagram of these stages is shown in *Figure 3.5.*

The pipeline stages are powerful in comparison with each Cytocomputer stage. Each stage contains two image buffers, and input multiplexers to accept data from any of the three image pathways, or an arithmetic or logical combination of these three inputs. A combination of arithmetic and boolean operations may then be performed on the image in either of the two image buffers, prior to its output. The PIPE may also function as a MIMD machine, with one of the two image buffers in a stage serving as a map for regions of interest in the image stored in the other buffer. PIPE may also function as a pyramid machine, with the images carried by the forward pathway being reduced in size by one half at each stage, while the image size in the retrograde pathway is doubled at each stage. All stages in the machine can operate independently, making it a MIMD design. However, each individual stage normally operates on a SIMD basis, with the exception of the case mentioned above.

Each stage currently supports 3×3 neighbourhood operations, but this will be expandable to 5×5 and possibly further in the future. Input and output buffers are provided to interface to camera and monitors, and these can communicate with any of the stages via one of four global buses. The device appears to be significantly more powerful than the conceptually similar Cytocomputer for two main reasons. The first is that each stage within the pipeline is more powerful in the PIPE, allowing the storage of two complete image frames, and operation in MIMD stand-alone mode if desired. The second factor is the provision of the recursive and retrograde image pathways, which allow, for instance, the simulation of pipelines of arbitrary length.

The PIPE is commercially available, and a library of functions is offered. This includes algorithms for edge detection and thinning, averaging and noise reduction, motion detection, region growing and shrinking, and some pyramidal operations. The user usually operates the PIPE through an IBM PC as controller, which provides a set of utility programs and generates the required look up table values. A subset of PIPE's instruction set is functionally equivalent to the hierarchical cellular logic (HCL) descibed by Tanimoto [1984], in which operations are applied to objects called bit-pyramids which themselves are functions on spaces called heirarchical domains.

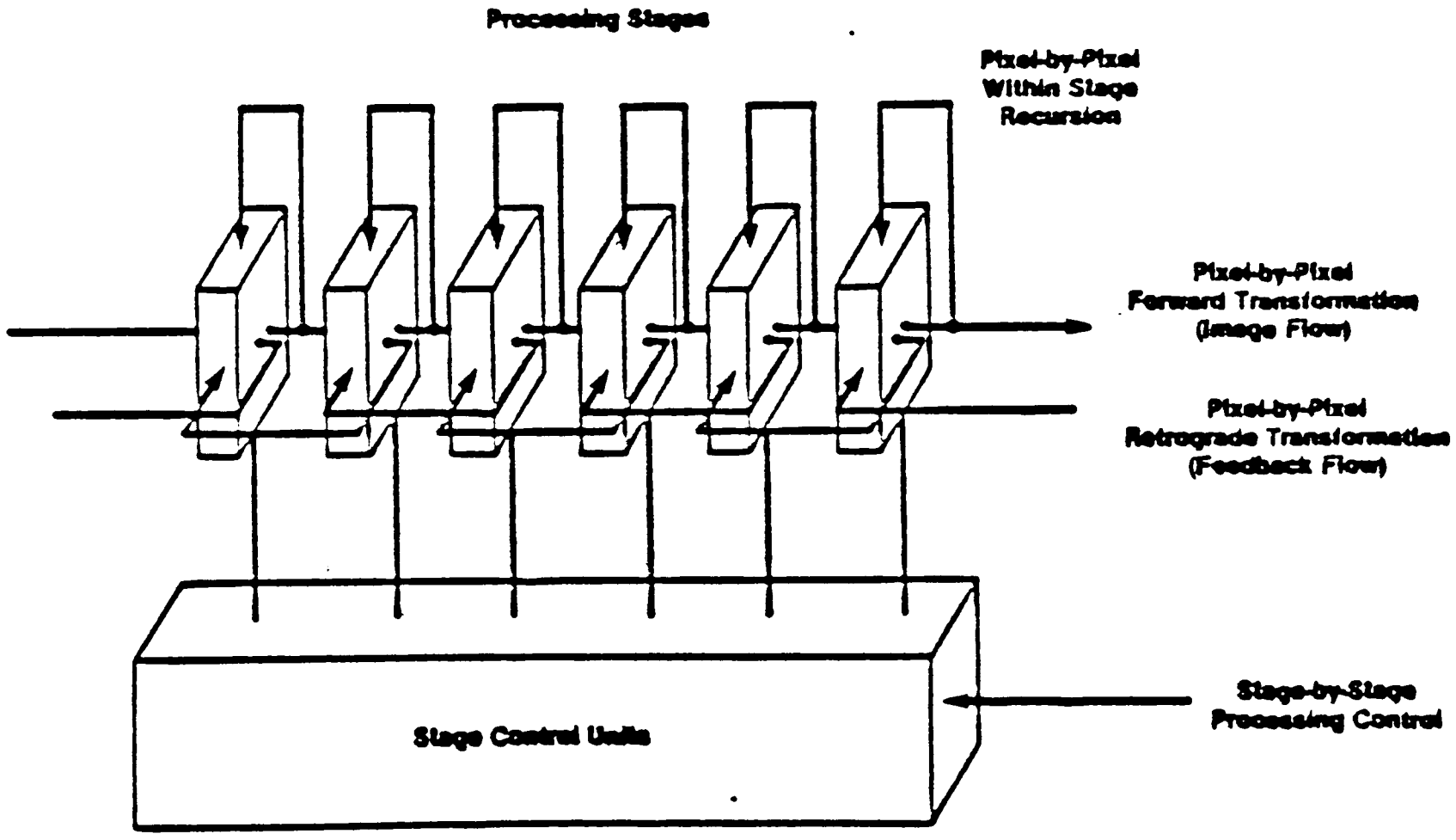


Figure 3.5: Processor Survey: The PIPE processor, showing pipelined stages. This differs from the Cytocomputer in having recursive and retrograde connections between stages, as well as feedforward.

3.3.3 The Connection Machine.

The Connection Machine (Hillis, [1985]) was developed at M.I.T in the early 1980's. One processing element is shown in *Figure 3.6.* It was designed as an array machine with 64K processing elements. However, the main feature of the architecture was, as the name suggests, in the interconnection between the processors. Whereas previous array machines had used simple 4-way connectivity, there are two communications networks in the Connection machine. The first is a "standard" local network with nearest 4-neighbour connections. There is also a global router, that allows any processor to communicate with any other processor in the array. This was designed so that the machine could be used for high level vision algorithms that cannot be performed by purely local interaction.

Each processing element consists of an ALU, flag register and 4k of attached memory (off chip). This basic operation reads two bits of memory and one flag register, combines them with a specified local operation, and writes out one bit to the memory and one bit to a flag. All of the PE's are broadcast the same information, but have independent operation controlled by the state of a flag. The main feature of the machine (other than the large number of processors) are the 4096 routers, each of which controls sixteen cells. These are connected in a boolean n-cube network, which allows processors to communicate with each other, and has a sustained random message bandwidth of about 20 GBits/sec. The PE's are controlled by a microcontroller which acts on macroinstructions from a host computer. PE's are packaged in a sixteen PE chip, which also contains a router.

Very high processing rates can be achieved by the machine. It has a peak instruction rate (32 bit additions) of about 1000 MIPS. Memory bandwidth is 200 GBits/sec and processor bandwidth is 330 GBits/sec. All processor actions are synchronised to an external clock. The router is responsible for routing messages between chips and delivering them to a specified address. The connection machine is programmed in CmLisp, which is a dialect of Common Lisp. This provides a high

level user interface. Example algorithms covering fields of document retrieval, fluid flow simulation and seismic data processing as well as for vision computing have been developed. A number of vision algorithms run on the machine, such as the Drumheller and Poggio stereo algorithm, and Canny's edge algorithm.

The Connection machine represents the current state of the art in available array processors, and shows a design trend away from concentrating on the processors themselves, to their interconnections. However, I/O operations are treated as normal, through the "edges" of the chips, still taking $O(n)$ time to input an image.

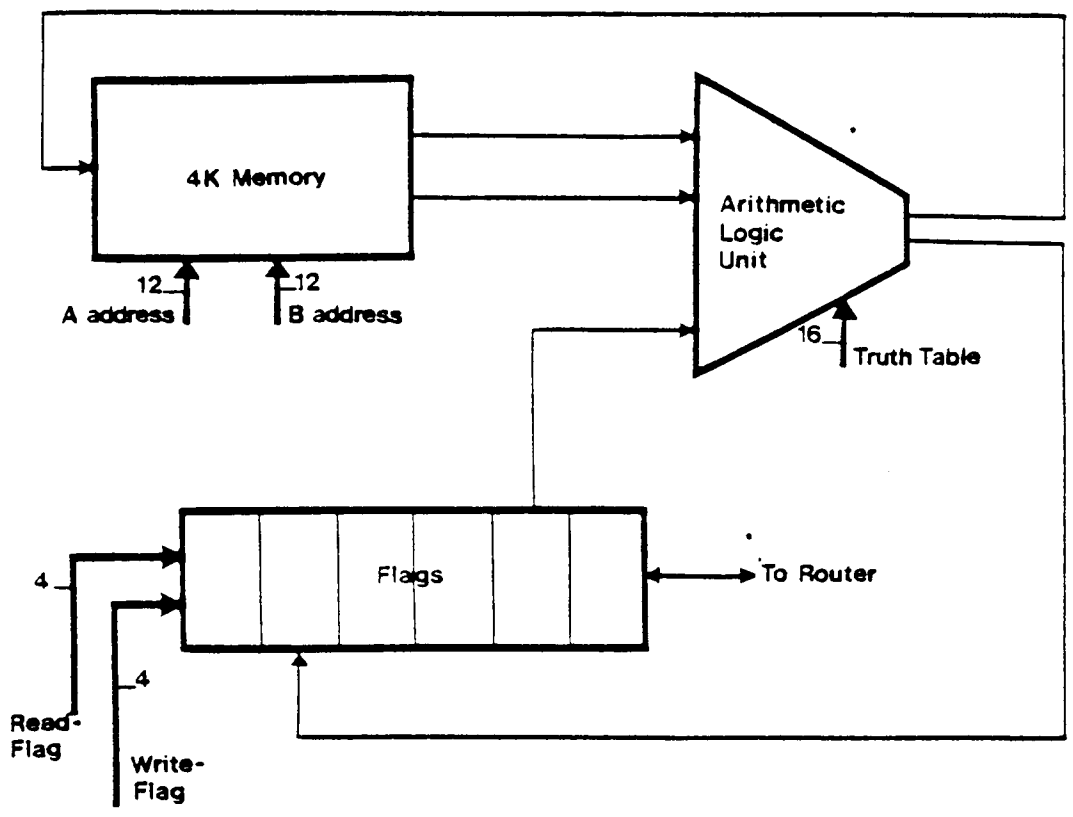


Figure 3.6: Processor Survey: One processing element of the Connection Machine, a large array processor with a global router communications network as well as the more usual local interconnectivity, making it perhaps the most powerful of current array processors

3.4 ARRAY PROCESSORS

To summarise the preceding two chapters on vision algorithms and the machines designed to execute them, we see that both spatial and temporal parallelism have been exploited in recent computer architectures to provide the very high computational rates necessary for real time vision processing. The main architectural thrust

is to use highly parallel computing structures, composed of hundreds or even thousands of individual computing elements. Such computing systems have structural properties that are suitable for VLSI implementation. By decomposing the system into a limited number of building blocks which can be used repetitively with simple interfaces, the cost-effectiveness of such a computer can be increased. However, the current trend indicates a diminishing growth rate for component speed, so major improvement in computational speed must come from the concurrent use of many processing elements and from advanced architectures. As we discussed with an example at the beginning of Chapter 2, we need an increase in processing speed over a serial computer of order $O(10^n); n > 6$ to provide real-time, video-rate processing for intermediate level vision algorithms. We are unlikely to be able to achieve more than $O(10^3)$ speedup by increasing device technology, so the interconnections between processors become increaingly important. However, this introduces heavy demands on communication and coordination between the individual processors, requiring algorithms which employ only simple, regular communication and control to allow efficient implementation.

This leads to perhaps the biggest developing problem of VLSI processing: communications bandwidth between processor and I/O device. In a compute bound algorithm, the number of computing operations is larger than the total number of input and output operations; otherwise the problem is I/O bound. As an example, matrix multiplication has $O(n^3)$ multiply add steps and $O(n^2)$ I/O elements and so is compute bound. On the other hand, the addition of two matrices is I/O bound, both operations taking $O(n^2)$ time, as there are n^2 adds and $3n^2$ I/O operations for the two input and one output matrices. Cutting down on the I/O time thus becomes crucial to high speed processing, in particular for I/O bound and non-intensive compute bound algorithms. The designer must pay attention to either reducing the I/O time itself, or to overlapping I/O operations with processing.

Chapter 4

AN ALGEBRAIC PERFORMANCE MODEL

4.1 Input/Output Methods

We have looked now at some current vision algorithms and at some parallel computers that were designed to execute such algorithms at high speed. There has been much discussion in the literature about the performance of such computers, and of their relative merits. However, a large amount of this analysis is subjective, being dependent upon the type of operation performed. This is unlike the analysis of software, where we may analyse average and worst case behaviour. Such performance analysis is crucial to the development of more efficient hardware architectures.

In this chapter we begin to develop an analysis of such computers. This analysis concentrates upon the relevance of the time taken to input and output images from the processing device compared with the time actually taken to process the image. This is one aspect of the performance of such computers that has been comparatively neglected in the literature, but which we felt could have important implications for the design of this type of hardware.

The analysis is performed within the framework of Hockney and Jesshope [1981] and of Hwang and Briggs [1984]. It is applied to array and pipeline computers, and we show that the time taken to input and output data can be a large fraction of the total processing time. In order to make the analysis tractable for complex ar-

chitectures, we obviously make a number of simplifying assumptions. We introduce them and discuss their reasonableness as we proceed.

We illustrate the relevance of I/O time to total processing time by means of some numerical examples. For an array processor, the total processing time is composed of two parts: that taken to load and unload the array with data, which we will denote as $T_{i/o}$, and that taken to actually process the data, which we will call the execution time T_e. Thus the total processing time for the array, ignoring the time taken to load the processing elements with instructions, which is a non-dynamic one-off operation, is:

$$\begin{aligned} T_a &= T_{i/o} + T_e \qquad (4.1) \\ \text{where } T_{i/o} &= \text{input/output time} \\ \text{and } T_e &= \text{execution time} \end{aligned}$$

We first consider the execution time T_e. This is proportional to the number of algorithm steps executed, and to the basic cycle time of the array T_c. We will assume that the processing elements can execute one instruction per cycle, which is typically the case with the processors that we have discussed. We also make the simplifying assumption that all of the array cycle times, T_c, are the same. If there is a single system clock (SIMD), then this is a reasonable approximation.

$$\begin{aligned} \text{Thus the execution time } T_e &= KT_c \qquad (4.2) \\ \text{where } T_c &= \text{time to execute one processor step.} \\ &\approx 100\text{nS} \\ \text{and } K &= \text{algorithm length} \end{aligned}$$

This assumes that all of the image can be loaded into the processing array at the same time. If this is not true (as is usually the case), then extra processing cycles must be used to process the image in sections. If the array size $= m \times n$ and the

image size= $M \times N$ then

$$T_e = \frac{M}{m} \times \frac{N}{n} \times KT_c \tag{4.3}$$

We have ignored boundary effects in the above equation. These would normally be found when processing pixels around the edges of the image. However, for the type of algorithms that we have discussed, this is simply a constant overhead, due to padding out the edge of the array, to allow local computational support.

Now we turn our attention to the input/output time, $T_{i/o}$. For an array computer aimed at vision, $T_{i/o}$ has two components (see *Figure 4.1*). If we assume that the image comes from a serial TV camera and outputs to a monitor, then it can only be read pixel-serially into the array if the array is as large as the image. We note here that the array has a boundary, over which image transfer usually takes place of $O(n)$, whereas the area of the array is $O(n^2)$. For smaller arrays (usually the case in practice- or the array is bit-serial) and for I/O methods other than pixel serial, the entire image field must be stored before and after processing. Thus there are two components to the I/O time - a time to store the entire image before and after processing, and the time taken to transfer the image from these stores into the processing array.

The time taken to store the image is a key bottleneck in the total processing time, being executed pixel serially. It is rare to find a camera system such as that shown in *Figure 4.2*, where the array may be loaded in a row-parallel fashion from the charged CCD line sensor. In the most usual case, when this part of the operation must be done serially, the first component of this time, that taken to store the entire image before and after processing, is:

$$T^1_{i/o} = 2MNT_s$$

where T_s = storage cycle time.

In order to derive reasonable formulae for the performance of array computers, we need to relate the parameters T_s and T_c. We will assume that the storage cycle

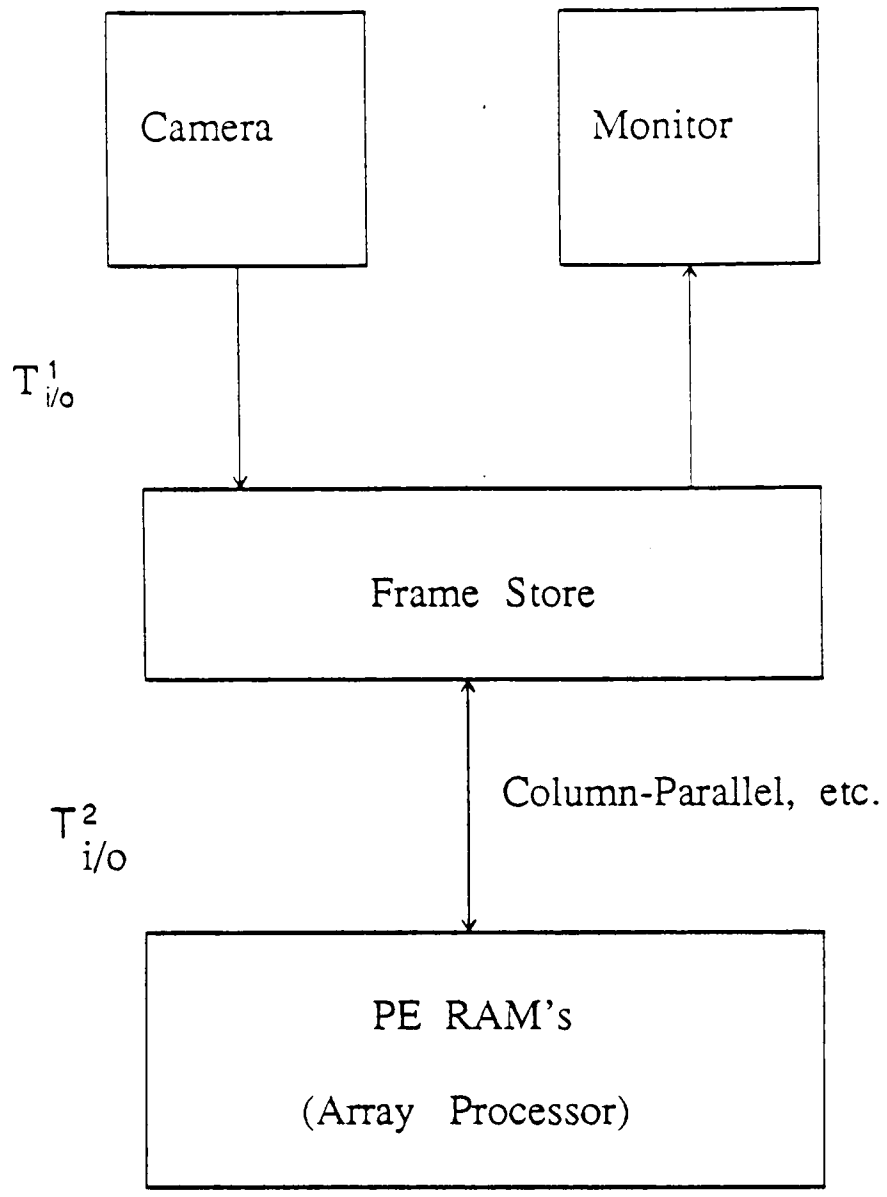


Figure 4.1: Two components to I/O array time. Shows the way in which data from a serial device may often have to be stored in its entirety before it can be read into the processing array.

time taken to store a pixel in memory is greater than the array cycle time. If an image is stored once every 1/25th of a second, then the average storage time per pixel varies from 152ns for a 512×512 image to 610ns for a 256×256 image. If the array cycle time is 100ns, then we can approximate:

$$T_c \leq T_s \leq 6T_c$$

We will choose

$$T_s = 3T_c$$

as a typical figure. This assumption does not change the essence of the results later in this chapter.

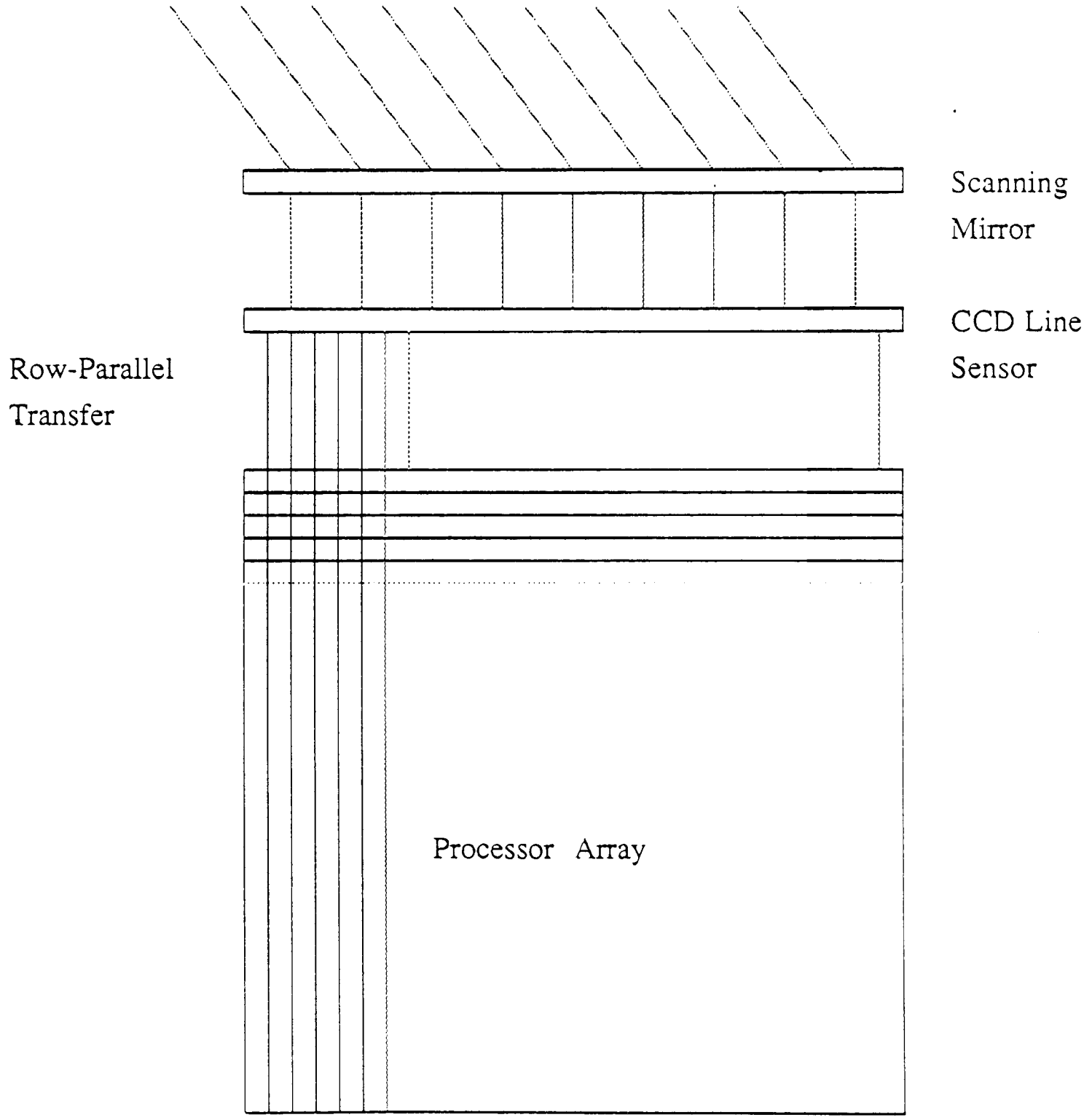


Figure 4.2: This illustrates the rare case in which the CCD sensor row is charged, and is then loaded in parallel into the processing array. This then takes $O(n)$ time to load, instead of the more usual $O(n^2)$.

The second component of the I/O time relates to transferring the image from the frame store into the array. We will compare four common ways of doing this (see Gerritsen, [1983]). The first of these is pixel-serial. This is the easiest and perhaps most common way. The second is to transfer the image in a column-parallel way. This involves hardware links between a column in the storage memory and one in the processing array, so that an entire image column may be transferred at once. We may also overlap this transfer with actual processing. If each processing element has only one data register, then processing must stop every time that data is to be tranferred in or out of the array. If, however, there is an independent I/O register in each processing element, this transfer may take place in the same clock cycle when the array is processing the last element of data. The final way that we consider is to transfer the image entirely in parallel from the storage memory. This is obviously the fastest method, but is also the most hardware intensive.

4.1.1 Pixel-serial

To transfer the image in pixel-serial fashion, requires a number of cycles equal to the number of pixels in the image, plus a factor for the number of bits per pixel, if the machine is a bit-serial one, as is the case with all of the machines mentioned in the last chapter. We must then transfer this data from the PE's I/O registers to the data registers. Both of these operations must be repeated to unload the array after processing. Define:

$$T_{i/o} = \text{array I/O cycle time } = 2T_c$$

($2T_c$ is the average figure for machines such as MPP,DAP,CLIP4, so we will choose this figure for our example. As we will show later, this second part of the I/O time is dominated by the first part, making this choice reasonable.)

$$T_i = \text{array instruction cycle time} = T_c$$

L = number of image bit-planes (bit-serial PE's)

For each bit-plane to be transferred from external memory to the PE-RAMs (or vice-versa) it is necessary to first transfer the bitplane to the PE's I/O registers. This takes $2MNT_{i/o}$ after which the bitplane in the PE's I/O registers needs to be copied to the PE-RAMs (taking one instruction cycle) $2LT_i$. Thus the total second phase I/O time is:

$$T^2_{i/o,s} = 2MNLT_{i/o} + 2LT_i \tag{4.4}$$

4.1.2 Column-Parallel

This type of I/O is used in the CLIP 4 and Connection Machine which we discussed in the last chapter. It is performed via a hardware link along one complete border of the PE array to the image memory. Again it is necessary to transfer each bitplane. However, the bit-planes can now be transfered by columns, which leads to a speed up by a factor of M over pixel-serial I/O, giving:

$$T^2_{i/o,c} = 2NLT_{i/o} + 2LT_i \tag{4.5}$$

4.1.3 Column-Parallel I/O overlapped with Processing

If column-parallel I/O is overlapped with processing (as in the MPP), then as I/O transfers only use the PE's S-registers (which are isolated during I/O), the processing needs only be interrupted for two clock cycles per bitplane transfer. So for the MPP the resulting I/O time for L bitplanes is

$$T^2_{i/o,co} = 2LT_c \quad K \geq 2LN \tag{4.6}$$

This relies on there being more algorithm steps than there are image transfers to be performed,ie $K \geq 2LN$. If this is not the case, then the algorithm will have been completed before image transfer is complete, so there will not be complete overlap. If this is the case, then the I/O time is:

$$T^2_{i/o,co} = \{2(N+1)L - K\}T_c \quad K < 2LN \tag{4.7}$$

Row-parallel I/O is essentially the same. However, in the DAP, the PE-RAMs are part of the host memory, so the host may fill the PE-RAMs row-parallel on a DMA cycle-stealing basis giving

$$T^2_{i/o,rp} = 2MLT_h \qquad T_h = \text{host cycle time } = 3T_c \tag{4.8}$$

4.1.4 Image-parallel

For image parallel I/O the transfer takes just one I/O cycle per bitplane so the I/O time is

$$T^2_{i/o,ip} = 2LT_{i/o} \tag{4.9}$$

4.1.5 General Case

For the general case where the PE array is not as large as the image array, we summarise our findings from the last four sections as follows:

$$\begin{aligned} T^2_{i/o} &= \tfrac{M}{m} \times \tfrac{N}{n} RT_c \\ \text{where } R &= 2L(2mn+1) && : \text{pixel} - \text{serial (A)} \\ &= 2L(2n+1) && : \text{col} - \text{parallel (B)} \\ &= 2L && K \geq 2LN \text{ overlapped (C)} \\ &= 2L(2n+1) - K && K < 2LN \text{ overlapped} \\ &= 4L && : \text{image} - \text{parallel (D)} \end{aligned}$$

We may summarise all of our findings so far as:

$$T_a = T_e + T_{i/o} \tag{4.10}$$

$$= KT_c + \{T^1_{i/o} + T^2_{i/o}\} \tag{4.11}$$

$$= KT_c + 2MNT_s + T^2_{i/o}$$

$$= (K + 6MN)T_c + T^2_{i/o} \tag{4.12}$$

4.1.6 Evaluation

Using the results from the last section, we can evaluate the importance of I/O time to total processing time. We do this by calculating the ratio of the execution time to the I/O time, which, from the above analysis is equal to:

$$\frac{T_e}{T_{i/o}} = \frac{KT_c}{6MNT_c + T^2_{i/o}} \tag{4.13}$$

We also calculate the ratio of the two parts of the total I/O time:

$$\frac{T^1_{i/o}}{T^2_{i/o}} = \frac{6MNT_c}{T^2_{i/o}} \tag{4.14}$$

where $T^2_{i/o}$ is evaluated using the formulae in the last section.

So to evaluate an example with $M = N = 512$ and $m = n = 128$ (quite a large size by current array standards) and with 8 bitplanes, we may calculate the ratio of execution to total I/O time and the ratio of the two I/O times for the varying I/O methods. In order to perform these calculations, we need to make an assumption about the algorithm length, K. For a simple, low-level vision algorithm, such as a Sobel edge finder, $K = 10$. For something more complex, such as a Canny edge finder, $K \approx 10000$. We take as our example the application of a Gaussian filter to the image, as used in many of the algorithms which we have discussed. We assume that the Gaussian to be applied has a window size of 20-30 pixels, as is the case with Marr and Hildreth's algorithm. We apply this filter by repeated iterations of a 3×3 mask (see Burt [1981]). To approximate to such a filter size will take of the order of 40-50 iterations, at 10 steps per iteration. Thus we assume that $K = 500$ for our example.

The results that follow from this are shown in table 4.1.

I/O method	(A)	(B)	(C)	(D)
$\frac{T_e}{T_{i/o}}$	.00005	.000305	.000318	.000318
$\frac{T^1_{i/o}}{T^2_{i/o}}$	.188	23.9	6144	3072

Table 4.1: Comparison of I/O methods, for pixel serial, column-parallel, overlapped column-parallel and image parallel.

There are two main conclusions that may be drawn from the results in table 4.1:

- I/O time plays the major part in PE array times (if the algorithm length is sufficiently small).

- The storage time to interface between a pixel serial camera and a bit serial PE array of smaller array size is also highly significant unless pixel-serial I/O

is being used.

In essence, the figures show that if it is necessary to store the entire image before and after processing, then with a non-pipelined system the actual I/O method used to interface memory to processor array has a comparatively small effect. There is also comparatively little gain to be had from using full image-parallel transfer as opposed to overlapping the I/O with processing (given that the average algorithm length is long enough to support this method).

Thus we can summarise our findings so far:

- I/O time is dominant for a large number of vision processing tasks unless the algorithm is very long.
- This leaves the designer with a serious data bottleneck between processor and memory or I/O device.

The designer of an array vision computer finds himself in the same situation that designers of serial computers found themselves in twenty years ago; that of having more processing power than the data pathways in the computer can manage. This bottleneck was alleviated by the development of *pipelining* techniques. By using pipelining in the architecture of serial machines, their performance in crucial areas such as arithmetic processing and maximum instruction rate was much improved. In the next Chapter, we discuss pipelining as a general technique and then go on to examine pipelining as a means of improving the performance of array computers in vision applications.

Chapter 5

PIPELINING

5.1 Pipelining - A Definition

We concluded the last chapter by noting that the I/O time for vision processors can often be the dominant fraction of the total processing time. We also saw that storing the video image before and after processing can take more time than transferring the image into the processing array. In this chapter, we investigate pipelining as a way of decreasing the proportion of I/O time compared to total processing time.

What do we mean by pipelining? I propose two main descriptions:

- *ASPIRATION* To ensure that the resources of a system operate at full capacity

- *STRUCTURE* Achieves this aspiration by means of overlapped computations to exploit temporal parallelism

In the following chapter I shall deal with these two main subjects, starting with structures employing pipelining in use today.

Partially pipelined architectures have existed for many years, their first main usage being in the instruction fetch of large mainframe computers. This is because there is typically a four stage operation such as :

Instruction Fetch

↓

Instruction Decode

↓

Operand Fetch

↓

Execute

(5.1)

The execution of the four stages may be overlapped in time. The flow of data from stage to stage is triggered by the pipeline clock (*Figure 5.1*). Pipelining may be used both on the instruction stream and on the data stream. Due to the temporal overlap of the stages, pipelining is suited for performing the same operations repeatedly, and is thus efficient for vector operations. Changes in operation (and thus contributory factors such as data dependency/ branching/ interrupts) all lower the effectiveness of the pipeline. A basic linear pipeline is shown in *Figure 5.2.* The pipeline stages are pure combinational circuits which perform arithmetic and logical operations on the data flowing through the pipe. High speed latches separate the stages and hold intermediate results between the stages. In a uniform-delay pipeline, all of the stages take an equal time to perform their functions. However, this is not usually the case in a practical system. The logic circuitry in each stage S_i has a time delay denoted by τ_i. Let τ_l be the time delay of each interface latch. Then the *clockperiod* (See Hwang and Briggs [1984] pp146) of a linear pipeline is defined by

$$\tau = \max\{\tau_i\} + \tau_l = \tau_m + \tau_l \tag{5.2}$$

The reciprocal of the clock period is called the *frequency* $f = 1/\tau$ of a pipeline processor.

It is possible (see Hwang and Briggs [1984]) to draw a space - time diagram to illustrate the overlapped operations in a linear pipeline processor. The space - time diagram of a four-stage pipeline processor is illustrated in *Figure 5.3* . It shows the overlapping of pipeline operations, and is used in a later section to define the efficiency of a pipeline.

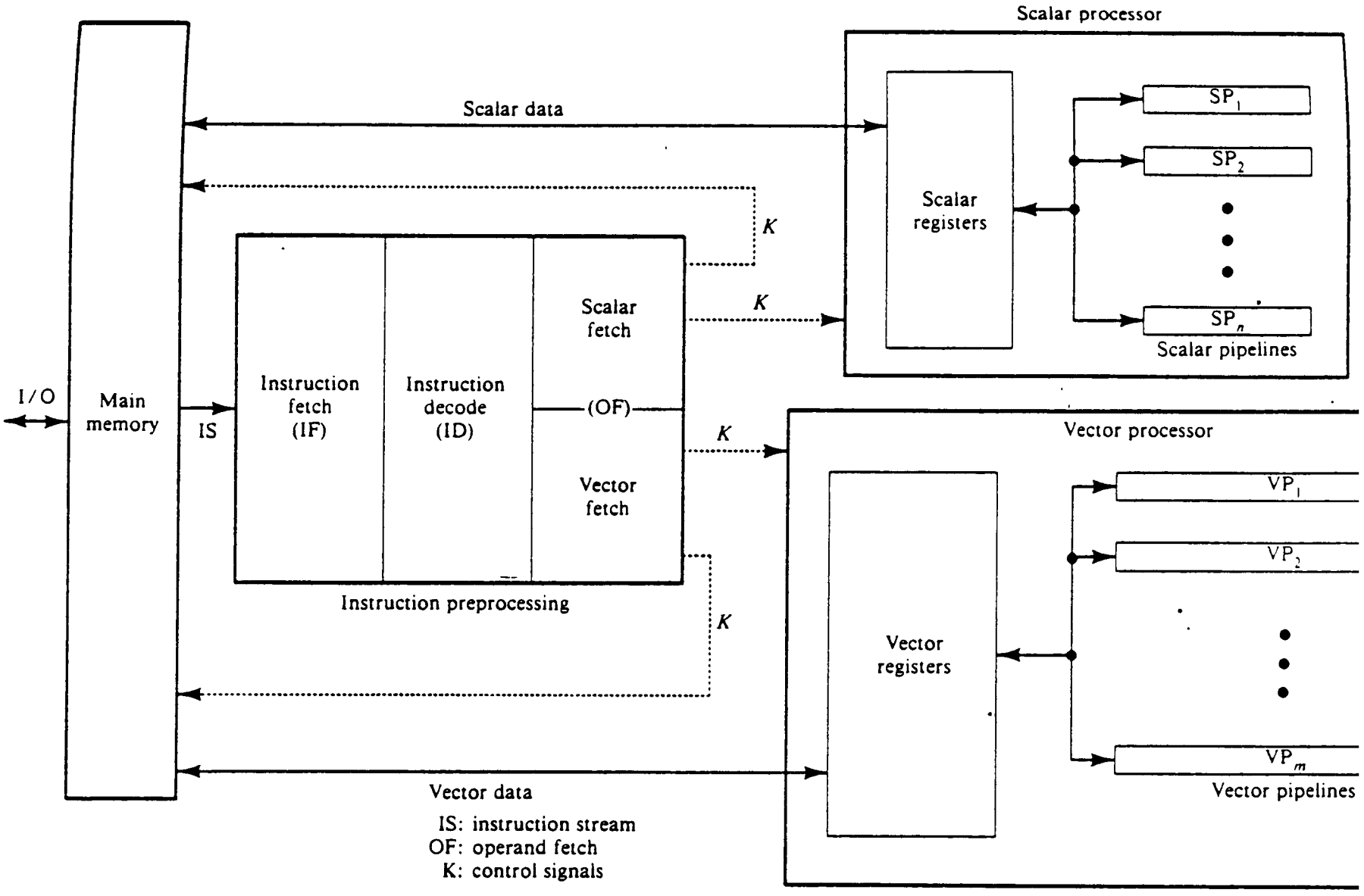


Figure 5.1: Functional structure of a modern computer with scalar and vector capabilities. The flow of data from stage to stage is triggered by the pipeline clock.

5.1.1 General Pipelines

Pipelines are not necessarily strictly linear. Various techniques have been developed over the years for increasing their performance. One way in common use is via feedforward and feedback techniques. In a feedforward pipeline, a stage S_i connects with "feedforward" to stage S_j where $j \geq i+2$ (see *Figure 5.4*). Likewise, feedback

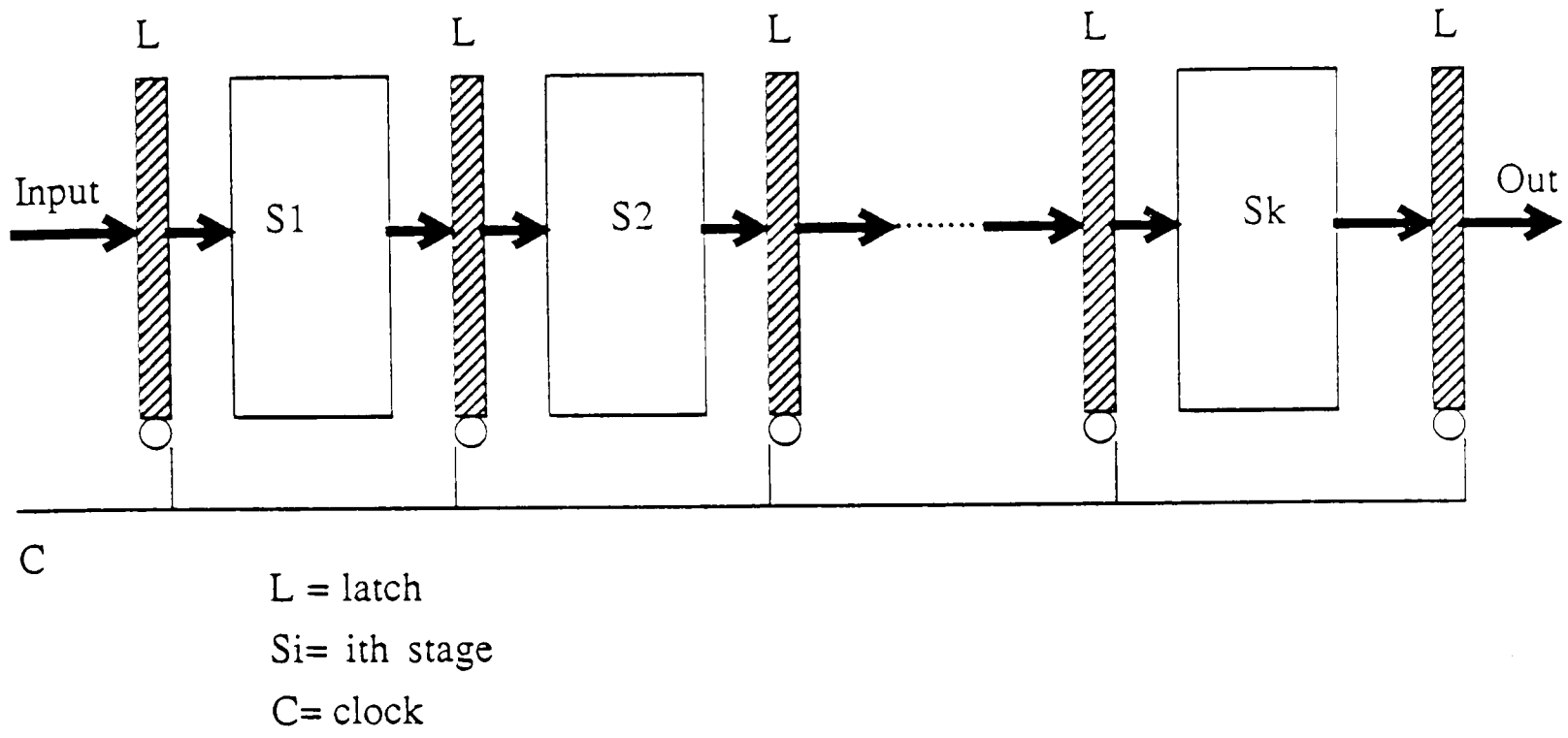


Figure 5.2: Basic structure of a linear pipeline processor. This shows the chaining together of (often identical) functional units.

connects S_i to S_j where $j \leq i$. We saw an example of this in the NBS PIPE computer discussed in chapter 3. Feedforward allows data to bypass certain computational stages within the pipeline, merging again with processed data further down the pipe. Feedback directly supports the implementation of recursive filtering. Feedback in pipelines is particularly common, as it is often the case that a pipeline has fewer stages than there are steps in the required algorithm, so the data must pass more than once through the pipeline. For vision, feedback within the pipeline can be very useful, as there are numerous algorithms which are recursive, as well as those which require repeated iteration of the same basic operation, such as application of a large Gaussian filter. Feedforward would be useful for higher level algorithms where data, such as a region in an image, can bypass stages in the pipeline depending upon an image parameter in the region.

Irrespective of whether feedforward or feedback is used, the pipeline can only be as fast as the slowest stage in that pipeline. This can cause a "bottleneck" within

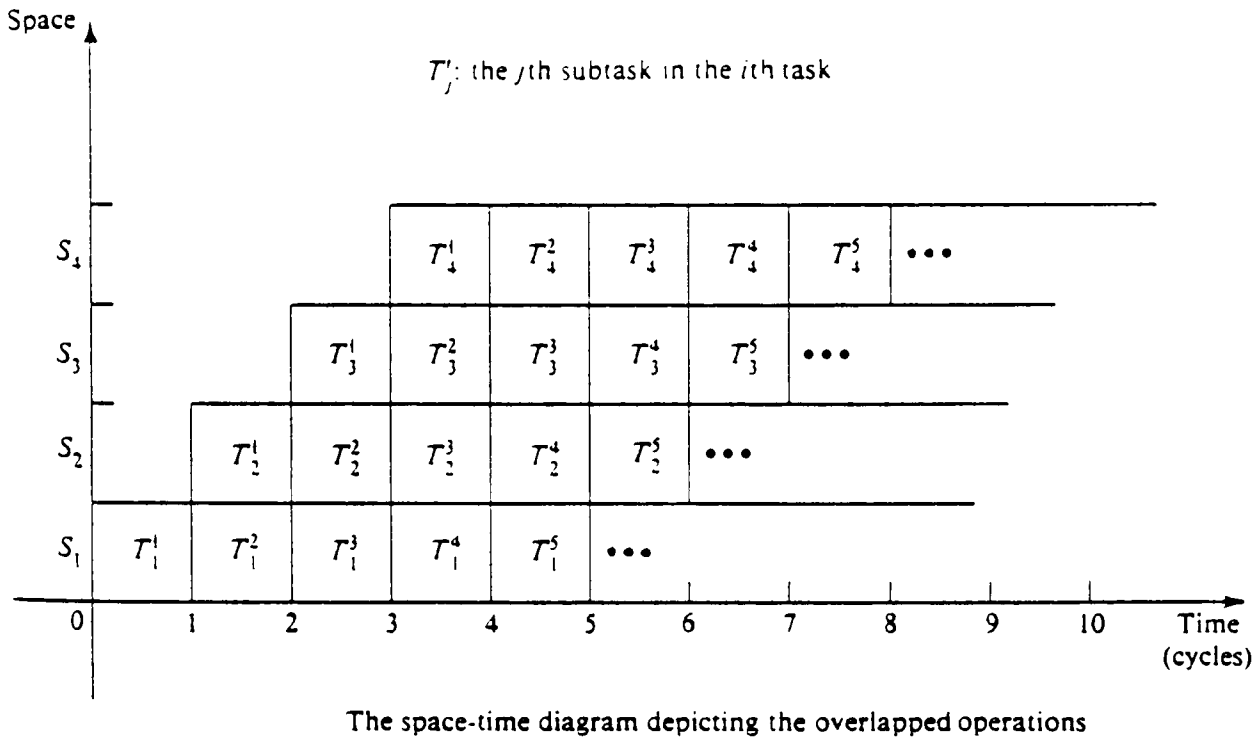


Figure 5.3: Linear pipeline processor for overlapped processing of multiple tasks. The space time diagram for a four stage pipeline illustrates the overlapped execution of program tasks.

the pipeline itself, and severely restrict its operating rate. Thus the subdivision of a task into sections which may be pipelined is crucial for efficient operation. Usually data and instruction buffers are used to even out the delays between pipeline stages and to provide uniformity of data flow. If one is still left with an unacceptably slow stage, then there are two main options. The first is to subdivide the stage further, if this is possible, or, if not, to replicate that stage, and to route data between one or more identical parallel stages (see *Figure 5.5*).

This problem of ensuring that no single pipeline stage takes longer to process its data than any other is similar to the problem facing the designers of parallel algorithms for array machines who have to ensure that the task is equally divided between all of the nodes in the computer. Careful design is necessary in both cases.

The choice of access to the data store is also of vital importance with vision computers. This is because of the large amounts of data being processed. As we have already seen, slow access to this data can easily dominate processing time with an array computer. Pipeline processors, like any others, can have memory-memory architectures (eg TI-ASC, CDC STAR100, CYBER 205) or register-register archi-

tectures (eg CRAY-1, VP-200). In a memory-memory architecture, the processing element or ALU takes its data directly from memory and when the result is produced, this is again output to memory. With a register-register architecture, the data for ALU's etc is taken from a register and the result returned to this or another register. The register then communicates this data with the main memory, as in a cache.

Memory to memory architectures tend to be slower, as access to memory takes longer than to registers. However register-register architectures involve extra hardware. For instance, the CRAY-1 has 64×4 instruction buffers and over 800 registers. Another notable feature of the CRAY-1 is its ability to chain its ALU pipelines together, so that results emerging from one pipeline can be fed directly into a second one without having to pass first through a register or through memory. This significantly eases the data flow through the machine. In the CRAY1 the 16 way interleaved memory has a data bandwidth of 320 Mwords/sec, although only 80 Mwords/sec of this are for data transfer (this means that data must be kept within the registers to avoid memory bandwidth problems). The actual processing performance of the machine varies with the type of operation being performed, the degree of chaining etc, but, typically, varies from 3-160 Mflops/sec.

The CRAY is not designed primarily for vision, but illustrates a number of points about pipelining which are equally important for vision computing, such as the ability to chain pipelines together. This is also illustrated in the PIPE computer, whose feedforward and feedback paths give the machine much greater flexibility than, say, the Cytocomputer. The interleaving of memory in the CRAY is also a useful design technique for vision, and is used in a pipeline computer design which we discuss later in this thesis.

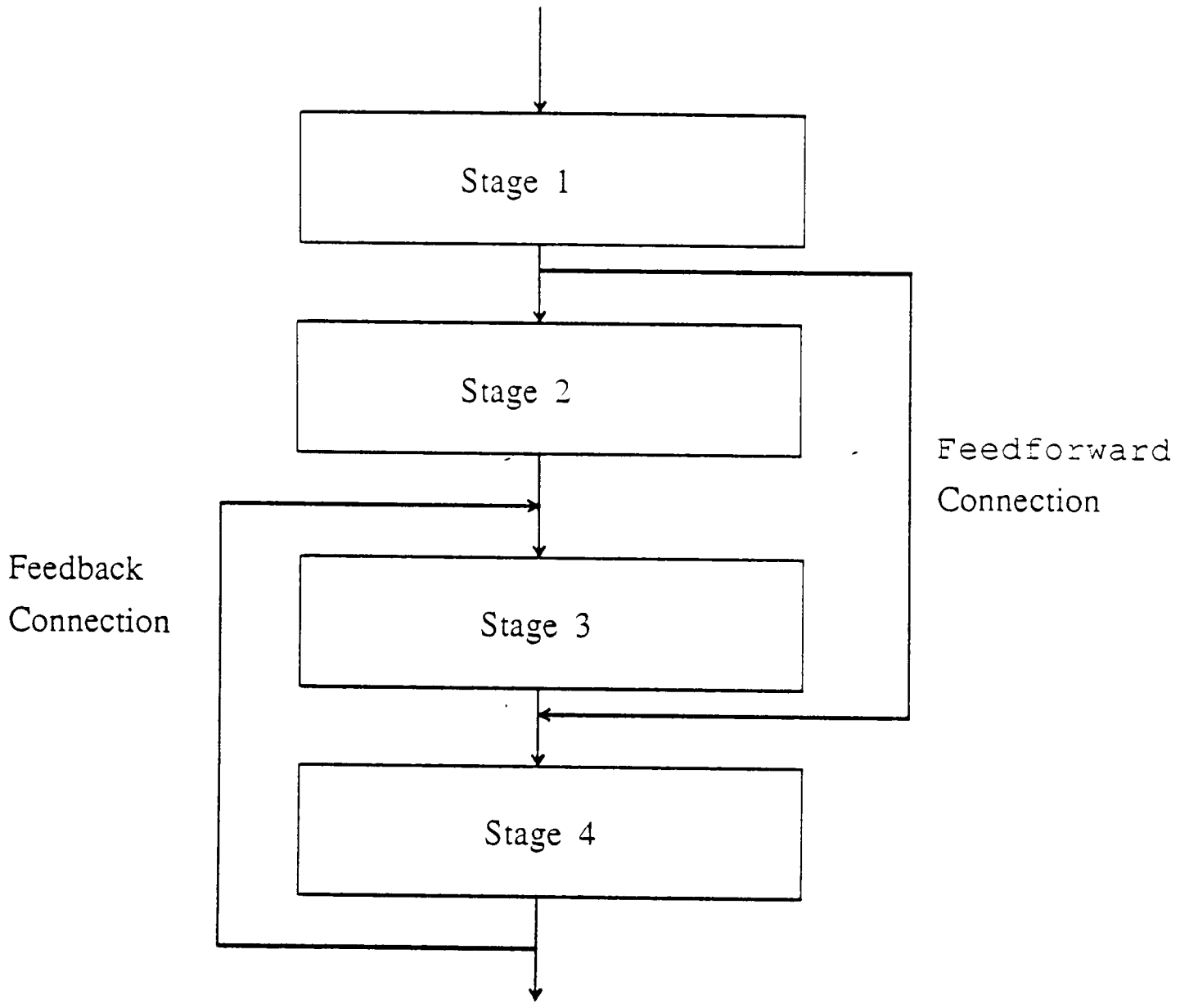
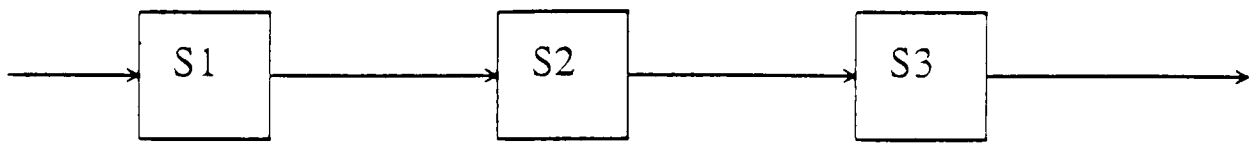


Figure 5.4: Part (a) shows feedforward in a pipeline chain, allowing data merging. Part (b) shows feedback within a pipeline, which can be used to accelerate the execution of recursive algorithms

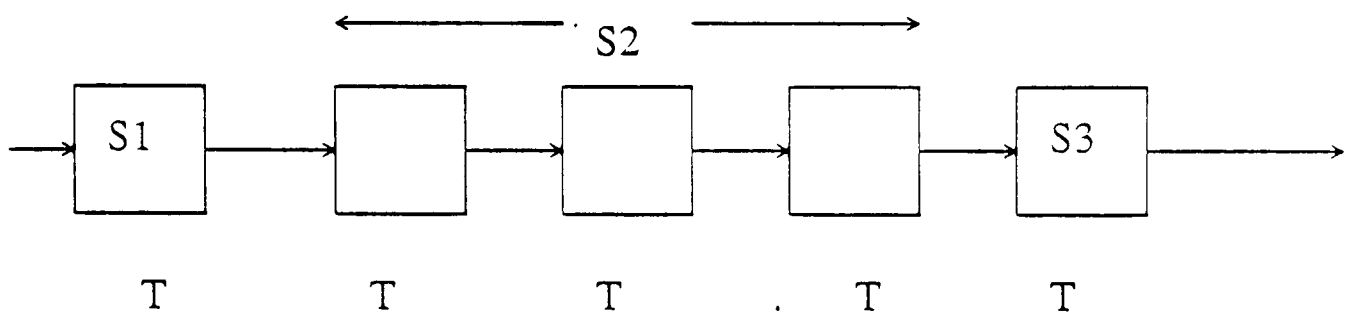
5.2 Speed and Efficiency

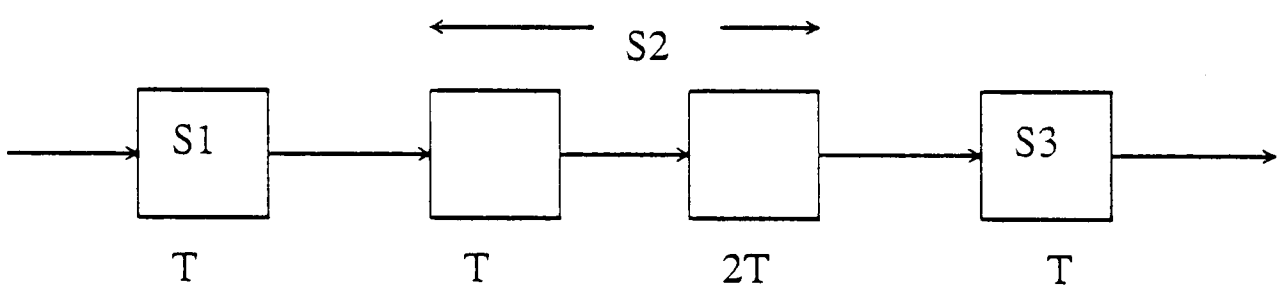
In the next two sections we analyse the speed and efficiency of a general pipeline. We show that as the number of tasks that the pipeline executes becomes orders of magnitude greater than the number of pipeline stages, its efficiency approaches one, and it can achieve a speedup over an equivalent non-pipelined processor almost equal to the number of pipeline stages. For this reason, it is important in vision machines that any pipelines used should be kept as full as possible to increase the efficiency of the architecture.



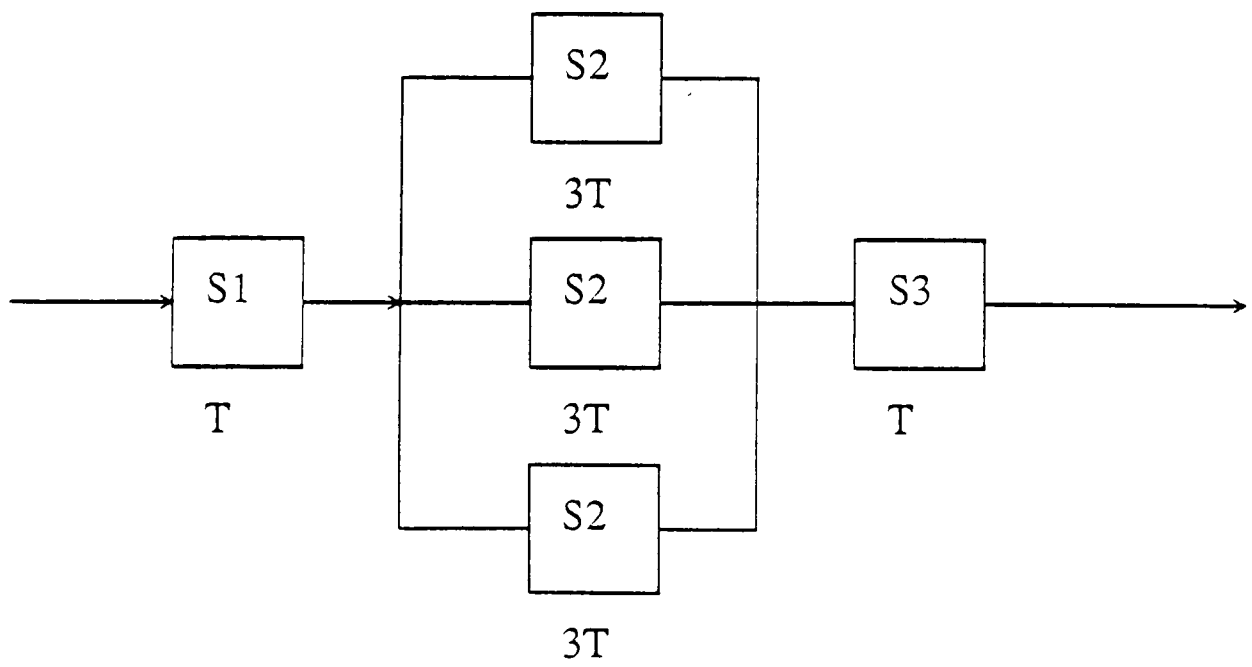
T1=T3=T, T2= 3T

(a) Segment two is the bottleneck





(b) Subdivision of segment two



(c) Replication of segment two.

Figure 5.5: Subdivision or replication to alleviate the bottleneck in a pipeline. With a slow pipeline stage causing a bottleneck in the pipe, the stage may be either subdivided or replicated to speed that stage.

In a linear pipeline, the pipeline cycle time can only be as fast as the slowest element in the pipeline, so the division of tasks into stages is crucial. For an instruction pipeline with K stages, N tasks (where each task has K parts) will take K machine cycles to fill up the pipeline completely. Results then come out of the pipeline at a rate of one per machine cycle, taking a further $N-1$ cycles to complete the task, so the total time taken is:

$$T_K = K + (N-1) \text{clock periods}$$

An equivalent serial processor (ie with the same clock rate) would take $N.K$ cycles to perform the same operations. We can define the speedup over an equivalent non-pipelined serial processor as (Hwang and Briggs [1984] pp148)

$$S_K = \frac{T_l}{T_K} = \frac{N.K}{K + (N-1)} \tag{5.3}$$

Thus when there is only one stage $K = 1$, the speedup is one, ie we have a serial computer. The design presented later in this thesis has $K = 2$, and so the speedup is

$$\frac{2N}{1+N}$$

which is approximately equal to 2, when N is large. Similarly when $K = 3$, the speedup is

$$\frac{3N}{2+N} = 3 - \frac{6}{2+N}$$

which is approximately equal to 3 when N is large. We see that, as expected, $S_K \Rightarrow K$ as $N \gg K$.

As the number of tasks gets very large, the speedup over an equivalent non-pipelined processor approaches its theoretical maximum of K, the number of pipeline stages. Similarly, we can define the *pipeline efficiency* by introducing two measures of performance in a linear pipeline processor. We say that the product of a time-interval and a stage-space in the space-time diagram (*Figure 5.3*) is called a time space span. This can be either in a *busy* state or an *idle* state. We measure the

efficiency of the pipeline by the percentage of busy time space spans, over the total time space span, which equals the sum of all busy and idle time-space spans.

$$\mu = \frac{\%\text{ busy time space span}}{\text{total time space span}}$$

where time space span = (time interval) $\times$ (stage space) (see *Figure 5.3*) Let

N = number of tasks

K = number of stages

T = pipeline clock period

Then

$$\mu = \frac{N.K.T}{K[KT + (N-1)T]} = \frac{N}{K + (N-1)} \tag{5.4}$$

Note that the limiting efficiency as N becomes very large is 1, and that

$$\mu = S_K/K$$

The *Pipeline throughput* is the number of tasks completed by the pipeline per unit time.

$$W = \frac{N}{KT + (N-1)T} = \frac{\mu}{T} \tag{5.5}$$

Note that $W = 1/T = f$ when $\mu = 1$.

5.3 Performance Analysis

5.3.1 Pipeline Speed

We can analyse the performance of our pipelined machine more formally by generalising the analysis to include pipelined vector machines. This is a pipelined computer with the capability of working on vector data, rather than scalar data, such as a byte-serial vision machine (eg the Parallel-Pipelined Processor later in this thesis), or a hypothetical machine which could process, for instance, one image row at a time. The analysis follows Hwang and Briggs [1984], and proceeds as follows. Let

there be

$$\begin{aligned} K &= \text{number of pipeline stages} \\ T &= \text{total pipeline delay in one instruction execution} \\ n &= \text{number of instructions in the program} \\ N_i &= \text{length of vector operands used in the } i^{th} \text{ instruction} \\ ie\ N(i) &= 1 \Rightarrow \text{scalar instruction} \\ T_i &= \text{time required to finish } i^{th} \text{ instruction in the pipelined computer} \\ T_p &= \text{total task time for } n \text{ instructions} \\ S_K &= \text{speedup of pipeline computer over serial one} \\ \mu &= \text{pipeline computer efficiency} \end{aligned} \tag{5.6}$$

The average time delay in a stage

$$\tau = T/K$$

$$T_i = K.\tau + (N_i - 1)\tau = (N_i + K - 1)T/K$$

where $K.\tau$ = time to fill up pipeline

When N_i gets very large so $T_i \Rightarrow N_i.\tau$ and throughput $W \Rightarrow K/T$.

For a sequence of n vector instructions, the degree of parallelism is represented by the vector length N_i. If we assume that the same vector length implies that the execution times are equivalent, then we may say

$$T_p = \sum_{i=1}^{n} T_i \tag{5.7}$$

$$= (T/K)[(K-1)n + \sum_{i=1}^{n} N_i] \tag{5.8}$$

The same code on a serial non-pipelined processor takes

$$T_S = T.\sum_{i=1}^{n} N_i \tag{5.9}$$

$$\text{So speedup } S_K = \frac{T_S}{T_p}$$

$$= K.\sum_{i=1}^{n} \frac{N_i}{(K-1)n + \sum_{i=1}^{n} N_i} \tag{5.10}$$

$$\text{Efficiency} = S_K/K$$

The pipeline efficiency may be interpreted as the ratio of the actual speedup to the maximum possible speedup K. We can use a numerical example as an illustration. Consider a vector job with a vector length distribution $N_i = 7, 3, 10, 1, 4, 6, 2, 5, 2, 4$ for $n = 10$ vector instructions. A plot of S_K and μ against K may be found in *Figure 5.6* for the above distribution. When K increases beyond the average vector length (4.4 in this example) the increase in speedup becomes rather flat while the efficiency continues to decline. This means that there is a diminishing return for fixed algorithm length in increasing the number of pipeline stages beyond a certain point.

In general, pipeline computers favour long vectors. As the vector length fed into a pipeline increases, so the overheads proportionally decrease. *Figure 5.7* shows the speedup against vector length for a pipeline of length $K = 8$. As the speedup approaches its maximum value of 8, so the vector length approaches infinity. The dashed line in the figure shows the effect of partitioning the vector into 16-element segments. The maximum speedup drops to $8 \times 16/(16 + 7) = 5 \cdot 5$. This occurs when the vector length is a multiple of 16, the number of component registers in the vector register. The analysis shows us that in our vision machine we need to operate on the longest vectors that we can, for instance on a 512 element row.

5.3.2 Pipeline Cost

The usefulness of a pipelined computer depends on its hardware cost and on the delay of each pipeline stage (amongst other things, such as its ease of debugging). In this section we introduce the cost of pipelined hardware in relation to the speedup gained. We once again show that the limiting efficiency for the pipeline is one. We also derive a formula to calculate the optimum cost-efficient length for a pipeline for a particular hardware structure. As most vision designers wish performance from

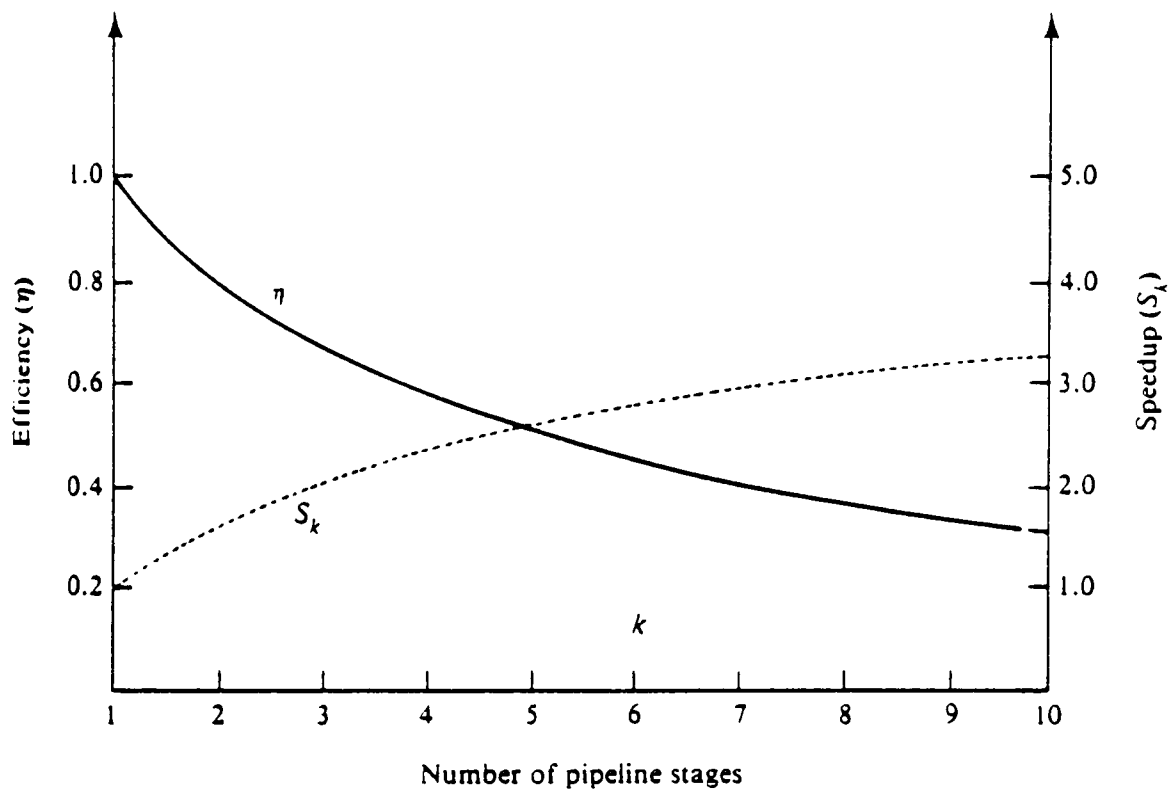


Figure 5.6: The speedup (s_k) and efficiency (η) of a pipelined processor with k stages. For a fixed length vector instruction, the speedup and efficiency increase as the pipeline length increases.

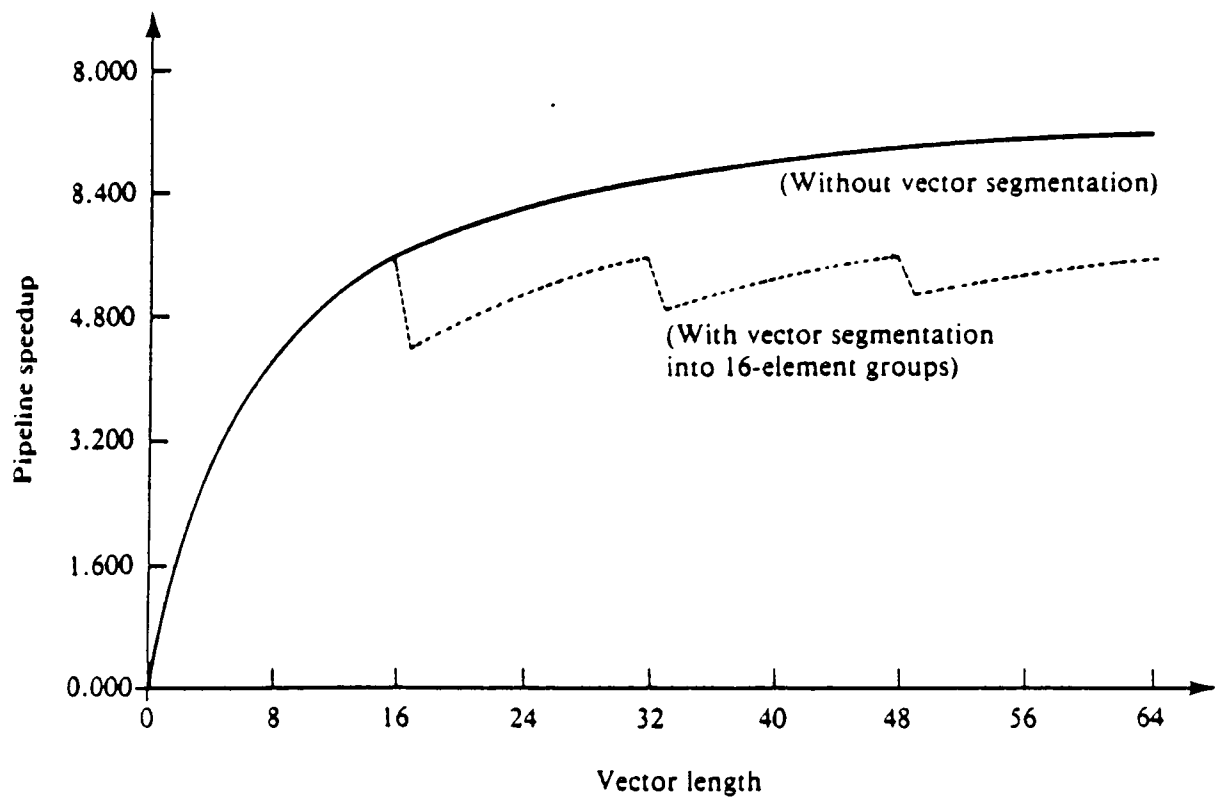


Figure 5.7: Pipeline speedup with and without vector looping. As the vector length increases, the pipeline overheads decrease. The plot shows speedup versus vector length for an 8–stage pipeline. As the vector length increases, the speedup tends to the theoretical maximum of 8 times.

their machines which can be attained in general use, rather than in a few restricted operations, this must be gained at a reasonable cost to the user, so it is important to analyse the type of hardware that we are proposing. We may analyse the cost of such a structure thus (Hwang and Briggs [1984], Eq 4.16). Let C_i = cost of ith stage

τ_i = delay of ith stage

k = number of stages

n = total number of jobs

$\tau = \tau_j$ = pipeline clock period equal to delay of slowest "bottleneck" stage.

$$\mu = \frac{n \sum_{l=1}^{k} C_l \tau_l}{\sum_{i=1}^{k} C_i [\sum_{j=1}^{k} \tau_j + (n-1)\tau]} \tag{5.11}$$

With uniform delay $\tau_i = \tau$ (a reasonable assumption in a pipeline for vision as most of the tasks are of similar nature) then this implies

$$\mu = \frac{n}{k + (n-1)} \tag{5.12}$$

When the pipeline approaches steady state, with sufficiently long vector input, then the limiting efficiency becomes

$$\lim_{n \to \infty} \mu = \frac{\sum_{i=1}^{k} C_i \tau_i}{\tau \sum_{i=1}^{k} C_i} \tag{5.13}$$

Again with uniform delay $\tau_i = \tau$ the limiting efficiency is one and the maximum speedup tends towards k.

The cost effectiveness of a pipeline computer is indicated by its potential throughput relative to the total processor cost. If we assume a job that takes a total time T_s in a serial processor and consider the same job in a k stage pipeline.

Let pipeline clock period be

$$\tau = T_s/k + \theta$$

where θ = the latch delay, ie the time taken to just transfer data from one pipeline stage to another.

Thus in $n.\tau = T_s + n\theta$ time units, n results can be produced. Hwang and Briggs [1984] (pp.319, eqn 4.19) show that:

$$W = \frac{n}{n(T_S/k + \theta)} = \frac{1}{T_S/K + \theta} \tag{5.14}$$

Now, let $C = \sum_{i=1}^{k}$; where C_i =total cost of all stages and d = average latch cost so that the Cost of entire pipe $= C + k.d$ Then if we define a *performance/cost ratio* (PCR) as

$$PCR = \frac{W}{C + k.d} = \frac{1}{(T_S/k + \theta)(C + k.d)} \tag{5.15}$$

Thus we can calculate the optimum number of stages in a pipeline for given cost parameters, after Hwang[1984], (eqn 4.2.1)

$$\frac{\delta(PCR)}{\delta k} = \frac{T_S C/k^2 - \theta.d}{(T_S/k + \theta)^2(C + k.d)^2} \tag{5.16}$$

$$\Rightarrow k_0 = \sqrt{\frac{T_S.C}{\theta.d}} \tag{5.17}$$

We now apply this analysis in our design of a pipelined vision computer to indicate the optimum pipeline length to use in, for instance, each ALU. We let the latch delay

$$\theta = 40\text{nS}$$

being an easily attainable register clock rate. The time taken to do the job on a serial computer we assign as being the time taken to cover a 512×512 image, with nine operations per pixel (i.e. one 3×3 convolution, and a clock rate of 1MHz). Thus

$$T_S = 512 \times 512 \times 9 \times 10^{-6}$$

We now need to define values for the value of the total cost of all stages and the average latch cost. We use a ratio here for the costs of $\frac{C}{d} = 100$, latches being significantly cheaper than the other circuits such as multipliers to be found in pipeline stages.

Using these figures, we get an optimal value of 150, 512-pixel rows. This would obviously make a very large pipelined computer, and it may be that global cost functions would then alter the calculations such as the memory access problems that would occur with such a large machine. The main point to be made from this analysis is that the to increase the efficiency and cost effectiveness of a pipelined computer, the number of stages must be made significantly large with respect to the total amount of data to be processed. ie the image size.

5.4 Pipeline Problems

We saw earlier that pipeline processors only work to full advantage when the pipeline is kept full, as stopping and reloading the pipeline causes delay and significantly reduces the processing speed, and the pipeline efficiency. This can occur as a problem in an ALU type of pipeline, where the operator might wish to change the function of the pipe from addition to multiplication, but it is perhaps a more significant problem in instruction pipelines (the most common use). Here branching and conditional instructions can cause serious delays within the pipe.

A simple analysis of the above may be made as follows (see also Hwang [1984] pp191). We use this framework to look at the parallel-pipeline computer design in Chapter 7. If we assume an n-segment instruction pipeline, with one instruction including n pipeline cycles, then if no branching occurs, the performance is one instruction per pipeline cycle. Let p be the probability of a conditional branch instruction occurring and let q be the probability that the branch is successful. Also assume that there are m instructions waiting to be executed.

The number then of instructions causing successful branches is mpq. Since $(n-1)/n$ extra time delay is needed for each successful branch, the total number of instruction cycles needed to process m instructions is

$$\text{number of cycles } = (n+m+1)/n + (mpq)(n-1)/n \tag{5.18}$$

As m becomes very large, the performance of the pipeline is measured by the

average number of instructions executed per pipeline cycle.

$$\lim_{m\to\infty} \frac{m}{(n+m+1)/n + mpq(n-1)/n} = \frac{n}{1+pq(n-1)} \tag{5.19}$$

Thus if there are no branch instructions ($p = 0$) then n instructions are executed per n pipeline clocks, which is ideal. If we take an example with a 5 stage pipeline, where $p = 20\%$ and $q = 60\%$ then you get only 3.38 instructions per pipeline cycle on average, meaning that 32.4% of cycles are wasted. Although we may still be gaining an improved performance compared with a non-pipelined processor, the efficiency of our pipelined processor has decreased. Thus when using a pipelined program control unit (as in one of the later design vignettes) to control image data flow through an ALU, careful attention must be paid to any branching that is necessary.

One way of getting around the problem is to use sequential and target prefetch buffers inbetween the memory and the pipeline. The sequential buffer holds instruction words fetched during the sequential part of a run. When a branch occurs the contents of this buffer are dumped and the pipeline is loaded from the target prefetch buffer. This buffer holds instruction words fetched from the target of a conditional jump. By using these two sets of buffers the branching delay can be reduced to zero in many cases, at the expense of extra register hardware. This may well not be cost effective for a processor that will execute a few conditional jumps, or an ALU type of processor which performs a largely non-intelligent function, but can be very worthwhile for a controlling PCU.

5.5 Pipeline vs. Array Computers

Having discussed the concept of pipelining we now compare the performance of four of the machines introduced in section three. We will examine the performance of three established array computers (CLIP4, DAP and MPP) with that of one of the few pipelined computers dedicated to vision available today, the Cytocomputer. We

have already shown (Eqn 4.2) that for an array computer the total time to process an image is given by

$$T_a = T_e + T_i/o \tag{5.20}$$

where the execution time is given by Equation 4.3

$$T_e = \frac{M}{m} \times \frac{N}{n} \times KT_c \tag{5.21}$$

and the I/O time is composed of two components, the second one being given by Equation 4.1.5.

$$T^2_{i/o} = \frac{M}{m} \times \frac{N}{n} RT_c \tag{5.22}$$

where R depends upon the I/O method being used, and is defined in Equation 4.1.5. We combine these to give:

$$T_a = BIT_h + [M/m][N/n](K + R)T_c \tag{5.23}$$

where R is defined as before to take into account the differing I/O schemes. Here we also include the time to load the array with instructions BIT_h, where

B = number of bytes per instruction

I = number of instructions = number of algorithm steps.

T_h = Host computer cycle time (all current vision computers interface to a high level machine, which will, in most cases, have a slower clock rate than the vision processor).

We have also not included the time to store the entire image before and after processing as in most cases this will affect both pipelined and array computers. Only when the pipelined machine has enough stages to fit the entire required algorithm into one pass through the pipe can this store be dispensed with.

We now calculate a performance formula for the Cytocomputer mentioned in section three. The Cytocomputer consists of 88 pipelined stages, each capable of performing a 3×3 neighbourhood operation on an image passing through the stage.

K	CLIP4	DAP	MPP	CYTO
0	3.93	5.99	2.97	2.62
50	4.14	6.32	2.98	2.88
100	4.32	6.66	2.98	5.36
150	4.49	6.99	2.99	6.81
200	4.66	7.32	2.99	11.0
250	4.84	7.65	3.0	11.8
300	5.01	7.99	3.0	16.7
350	5.18	8.32	3.01	23.5
400	5.36	8.65	3.01	23.5
450	5.53	8.99	3.02	24.8
500	5.7	9.32	3.02	31.2
550	5.88	9.65	3.03	32.8

Table 5.1: Comparison of Execution time vs. algorithm length for some vision computers

Thus each stage takes $N + 2$ cycles to load, where N is the number of columns in the image, and then MN cycles to process the image. The total execution time for the Cytocomputer is then

$$T_C = BIT_h + [(MN) + K(N + 2)]\lceil K/S \rceil T_C \tag{5.24}$$

Here K is the number of steps in the algorithm and the $\lceil K/S \rceil$ term is the number of times that the data must pass through the entire length of the pipeline if the algorithm length is greater than the number of stages. We can thus calculate the execution time, in multiples of T_c the array or PE cycle time for algorithms of varying lengths K. These figures are only correct up to a multiplicative constant, so only the slopes of the graphs are relevant for our analysis. These figures are summarised in table 5.1.

As can be seen from the results in the table, when the algorithm length gets longer than the number of stages then the performance of the Cytocomputer tails off rapidly. It starts with a performance to equal that of the MPP, which has a far higher hardware cost. It is also noticeable that the MPP has a distinct performance advantage over the other two array processors. This is due to the fact that it has a higher number of processing elements and also to its overlapped I/O scheme. The

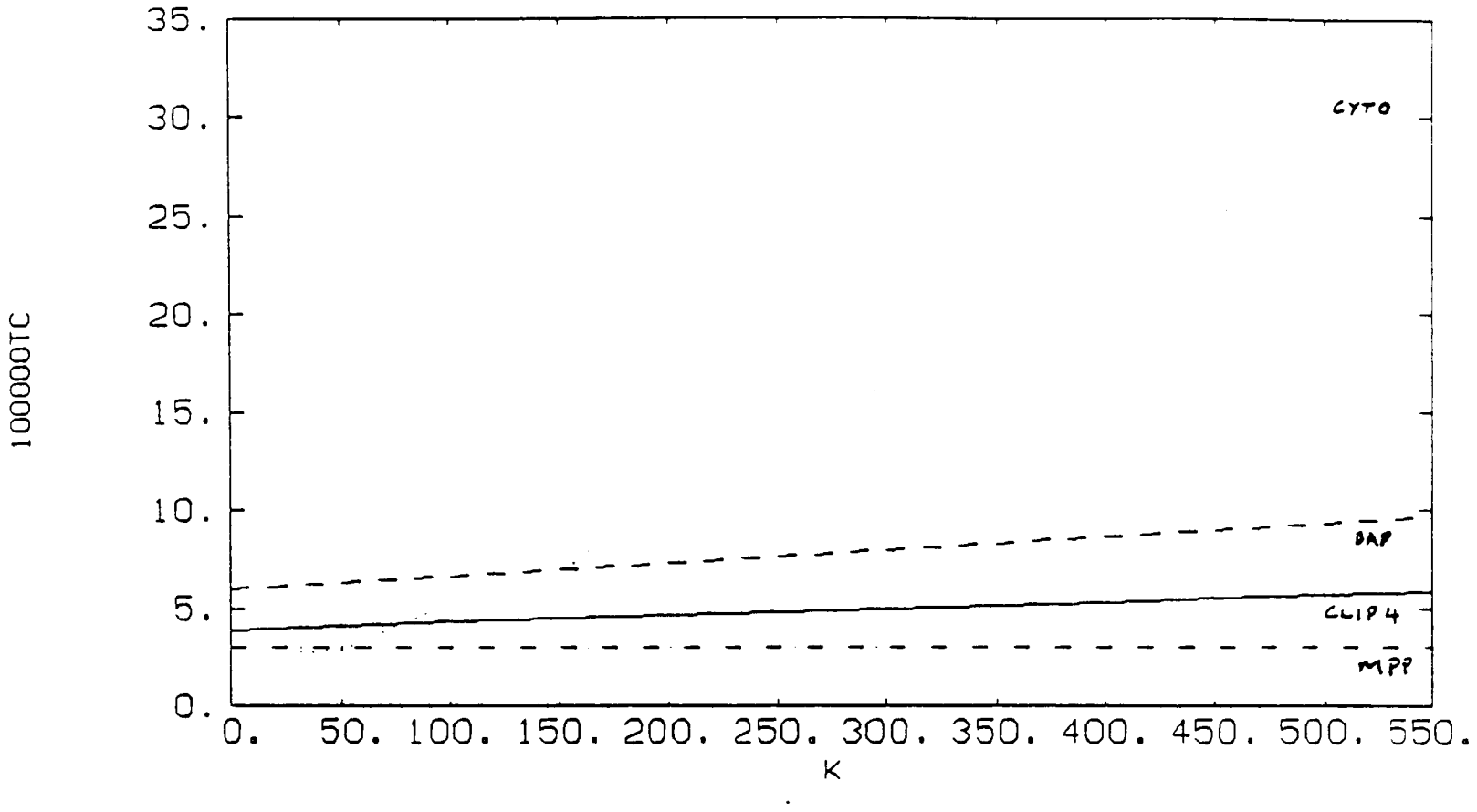


Figure 5.8: Computer performance; T_c vs k for 512×512 image. This shows the total execution time in multiples of T_c against algorithm length for the CLIP4, DAP, MPP and Cytocomputers. Notice how the Cytocomputer is competitive until the algorithm length becomes longer than the number of pipeline stages.

data is illustrated in *Figure 5.8*.

We see from the diagram the similar performance of the three array computers, and the very large tail-off in the performance of the Cytocomputer when the algorithm length gets longer than the number of pipeline stages. This empahsises the point that in pipeline computer design, the designer must be very careful to ensure both that the pipeline is kept full of data, and also that recursive use of the pipeline is not necessary. However, machines such as the PIPE, with a recursive pathway built into each stage, may be able to avoid this problem by careful program coding. In this way, if an algorithm required the same convolution to be repeated over the image a number of times, this could be done recursively in one stage. If this were the case, the pipeline in the PIPE computer would not have to be emptied between each pass, as would be the case with the Cytocomputer.

5.5.1 Classification of existing pipeline architectures.

We now examine three ways of classifying parallel and pipelined computers. These are used to analyse some current computers to see which of these computers may be best suited for vision applications, and are then used in later sections of the chapter to identify areas of a computer design which may benefit from pipeline techniques. We investigate three such areas.

Handler has proposed a classification scheme which considers parallel-pipeline processing at three subsystem levels (Hwang [1984] pp37)

Processor Control Unit (PCU) - this corresponds to one processor or CPU. A single CPU may control a number of ALU's within a computer. It thus executes the computer macro instructions.

Arithmetic Logic Unit (ALU) - This is equivalent to one PE in a SIMD array

Bit Level Circuit (BLC) - This corresponds to combinational logic circuitry needed to perform one-bit operations

He then defines a computer system C by the triple

$$T(C) =< K \times K', D \times D', W \times W' > \tag{5.25}$$

where :

$$\begin{aligned} K &= \text{number of PCU's} \\ K' &= \text{number of PCU's that can be pipelined} \\ D &= \text{number of ALU's under control of one PCU} \\ D' &= \text{number of ALU's that can be pipelined} \\ W &= \text{word length of one ALU or PE} \\ W' &= \text{number of pipeline stages in one PE/ALU} \end{aligned}$$

omitting K', D', and W' when they are equal to 1

For example,the ILLIAC IV computer has one controller operating sixty four arithmetic pipelines, each with 64 bit word lengths. Hence the triple is

$$T(ILLIACIV) =< 1, 64, 64 >$$

Using this classification, we would look for a vision system that had a high value of D, indicating a large number of processing elements that can be operated in parallel. If the value of D' is also high, the computer may have been optimised for data flow to and from the processors. A summary of triples for a number of computer systems is shown in table 5.2.

Computer model T(C)	System specification [1] $< K \times K', D \times D', W \times W' >$
T(PEPC)	$< 1 \times 3,288, 32 >$
T(IBM 360/91)	$< 1, 3, 64 \times (3 \sim 5) >$
T(Prime)	$< 5, 1, 16 >$
T(Cray-1)	$< 1, 12 \times 8^2, 64 \times (1 \sim 14) >$
T(CLIP4)	$< 1, 9216, 1 >$
T(DAP)	$< 1, 4096, 1 >$
T(MPP)	$< 1, 16384, 1 >$
T(Connection M)	$< 1, 65536, 1 >$
T(Cytocomp.)	$< 1, 1 \times 80, 1 >$
T(Illiac IV)	$< 1, 64, 64 >$
T(AP-120B)	$< 1, 2, 38 \times (2 \sim 3) >$

Table 5.2: Handler's Classification of Pipeline Architectures

This shows a brief definition of a number of current machines, showing that some utilise variable length pipeline stages for maximum efficiency. We show some serial computers as well as those that we discussed in Chapter 3 by way of comparison. We can see that the machine with the highest value of D is the Connection Machine, which was designed with vision processing as a prime task. However, there is no facility on this machine to pipeline the ALU's together. Note also that W for the CM is one, indicating that it is a bit-serial machine, and must thus process images in bit planes. This simplifies ALU hardware, but is not optimised to the usual 8 bits

[1] K', D', W' are omitted when equal to one

[2] For CRAY-1,the pipeline chaining degree is a variable with maximum value equal to 8

per pixel of vision. It is an approach adopted by the majority of array processors, as a means of simplifying the hardware involved.

A definition of the use to which a pipeline is put has been proposed by Handler (Handler, [1977]). He suggests three categories of usage:

- *Arithmetic*
- *Instruction*
- *Processor*

According to Handler's classification, an arithmetic pipeline is one that splits up ALU functions to provide a pipelined ALU. Thus for an ALU that is computing a floating point multiply, we may have individual stages for exponent equalisation, addition of the numbers, and normalisation. An instruction pipeline is used within the CPU of a system for instruction prefetch and overlapped execution. Thus, as already mentioned, we may overlap instruction fetch and decode, for instance. The processor type of pipeline links separate ALU's or other functional units together to provide pipeline chaining. This technique avoids unecessary memory and register interface to save processing time, and is used on machines such as the CRAY series. This classification shows the three main categories of use to which pipelines are currently put; by far the most common is arithmetic pipelining, with processor pipelining being comparatively rare.

In serial computers instruction pipelining is in common usage, as is arithmetic pipelining. With the type of array computer that seems likely to be most efficient for vision computing, it is the arithmetic pipeline that will be the crucial one. As we are likely to be using a SIMD machine, the instruction rate will be far lower than the actual data processing rate, with all of the PE's executing the same instructions, so the real problem will be to keep the processing array supplied with data at a sufficiently high rate. Arithmetic pipelining may be the way to achieve this.

Ramamoorthy and Li [1977] refine the classification of an individual pipeline further into the following groups.

- *Unifunction vs Multifunction* A multifunction pipeline can perform different functions at the same (or possibly different) times, whereas a unifunction pipeline can only perform a single function (for instance the pipeline in an ALU). We may regard the pipeline in a machine such as the Cytocomputer as a multifunction one, in the sense that its function is not strictly fixed; the coefficients and values of the look up tables etc. may be changed. This is different to a strictly pipelined ALU, where only the data may change. For vision computing our aim is to have a multifunction pipeline, so that the neighbourhood operation may be redefined in terms of function, neighbourhood size, etc.

- *Static vs Dynamic* A static pipeline is one that may assume only one functional configuration at a time, thus requiring that it executes a function that does not change frequently. A dynamic pipeline permits several functional configurations to exist simultaneously. Therefore a dynamic pipeline must be a multifunctional one; a unifunctional pipeline must, by definition, be static. So far dynamic pipelines are purely conceptual, in that no one has succeeded in making one. However, one might regard a data flow machine (Computer [1986]) as being a type of dynamic pipeline, in that the data packets are routed according to the identifiers that they carry. The pipeline itself does not allow several functions to exist simultaneously, but switches between functions for each individual data packet.

A Dynamic pipeline could definitely be of great use in vision processing, as such a pipeline could automatically reconfigure its state according to input parameters such as the input image. Thus the operations that are performed on an individual pixel neighbourhood could be a function of both the neighbourhood and of the pixels in surrounding neighbourhoods, with the type of

operation being constantly updated.

- *Scalar vs Vector* A scalar pipeline processes a sequence of scalar operands under the control of a do loop, whilst a vector pipeline handles vector instructions over vector operands (as in the CRAY), ie in a vectorised do-loop. At the moment, most vision computers are scalar ones, but we might imagine, technology constraints permitting, a vector machine that processed vector instructions over an entire row-vector simultaneously.

Our "ideal" vision machine should thus have a multifunction pipeline to allow easy reconfiguration; a dynamic pipeline so that the values of pixels around the region of interest may define the function applied; and a scalar pipeline if we wish to use it in the current SIMD format, which is employed in most parallel vision algorithms.

5.6 Pipelining as a Technique.

All the previous discussion relates to machines which use pipelines in order to improve instruction execution times, whether CPU or ALU. However, we can generalise the term "pipelining", so that it includes the following aims. For an instruction process, we wish to keep a smooth control flow, including cases where branching is performed. We must also enable instructions at a sufficient rate to keep the execution processors operating at their full capacity. For execution processors, such as ALU's, we wish to increase the operation rate to the maximum attainable.

One area that links all of the above is the memory-data bandwidth. Obviously it is of no use to have an ALU which can operate at 10MHz and then only be able to feed it data at 0.5MHz. It is to this area that pipelining is of most use, in particular to applications such as vision, where the amounts of data involved can be very large. This bandwidth thus needs to be high enough both to keep the processors supplied with data at the maximum rate that they can handle and also to keep the instruction pipelines full.

Whereas an instruction or PCU type of pipeline can improve the performance of a vision machine, the gains are likely to be small when compared with those obtainable when pipelining is used on the ALU's or processing elements of a vision machine. This is particularly true when the number of PE's in a machine is high, thus increasing the proportion of operations that are I/O rather than compute bound. For example, an array computer with 1000 processing elements is likely to be able to process data at a rate of a few hundred MFlops, whereas data buses that operate at such rates are rare and usually very complex. As the size of such processor arrays increases and the individual PE cycle time decreases with advancing technology, so the problem of keeping the data supplied to the processing array will increase. Current high speed data transfer methods, as used with high speed disc drives etc will not suffice for this problem, and new methods must be investigated.

5.7 Pipelining to Improve Memory Bandwidth

The opportunities for pipelining techniques in one or more dimensions to be used on such machines are great, and we will now look at some ways in which such techniques have already been used.

5.7.1 Serpentine Memory

Designers have attacked the bandwidth problem between large data memories (ie frame stores) and processing units with a number of different methods, such as simply increasing the transmission rate by careful interconnection of the components. But there are a number of pipelining techniques which exist to try and solve the problem of finding extra bandwidth, and one such technique is using a serpentine memory. This is configured as is shown in *Figure 5.9* .

Serial data is read into the memory from the left and "snakes" its way through the array until the array is full. The contents of the memory may then be downloaded into an array of PE's by whatever I/O method is the fastest. Data then

emerges from the bottom of the memory, which may then be refilled. This method is quite widely used to interface serial devices such as cameras to parallel arrays. Such a serpentine array has already been used in some machines, for example, as Nishihara's Gaussian image convolver (Nishihara, [1981]).

This design is illustrated in *Figure 5.10.* Output from the camera is fed into a serpentine memory, where it is temporarily stored and then supplied to the digital convolver. This approximates the convolution of the image with a Laplacian of Gaussian mask via a convolution with two Gaussian masks and then taking the difference of these. The two gaussian convolutions are then calculated by separating the x and y components, reducing the number of multiplications required from $O(m^2)$ to $O(m)$. After the image has been convolved with the DOG filter, further modules perform the zero crossing detection and stereo matching.

This is the most commonly used pipelining technique within vision, for it provides a convenient way of interfacing a two dimensional memory with a one dimensional camera or display device. The designer is still usually faced with the problem of interfacing this memory with the actual processing device, although this could perhaps be achieved by DMA. As a means of actually linking a processing array to a camera or similar device it is somewhat limited by the serpentine timing, which has to be controlled by the camera rather than the processor, thus limiting the amount of processing time available. As well as the Nishihara device, the technique has also been used in machines such as the Cytocomputer and in the NBS PIPE processor (see Kent, [1979]).

This technique may also be built into the processing array itself, as is shown in one of the design vignettes in the next chapter.

5.7.2 Memory Interleaving

Another method which is widely used is that of interleaving the memory. This technique is used in current vision computers to increase the memory bandwidth

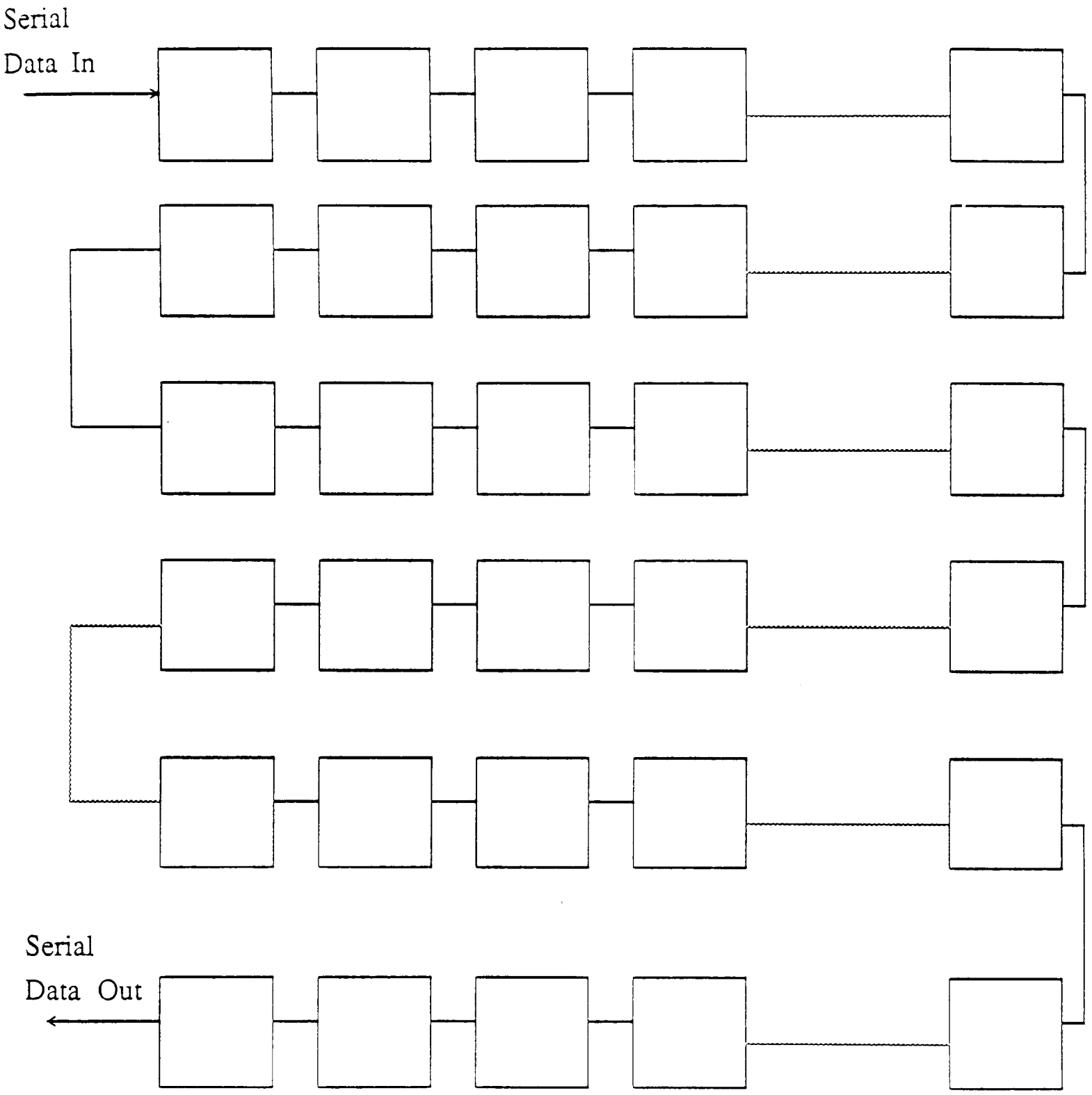


Figure 5.9: A Serpentine memory. Data snakes through the memory from a serial device, such as a camera, to allow processing "on the fly".

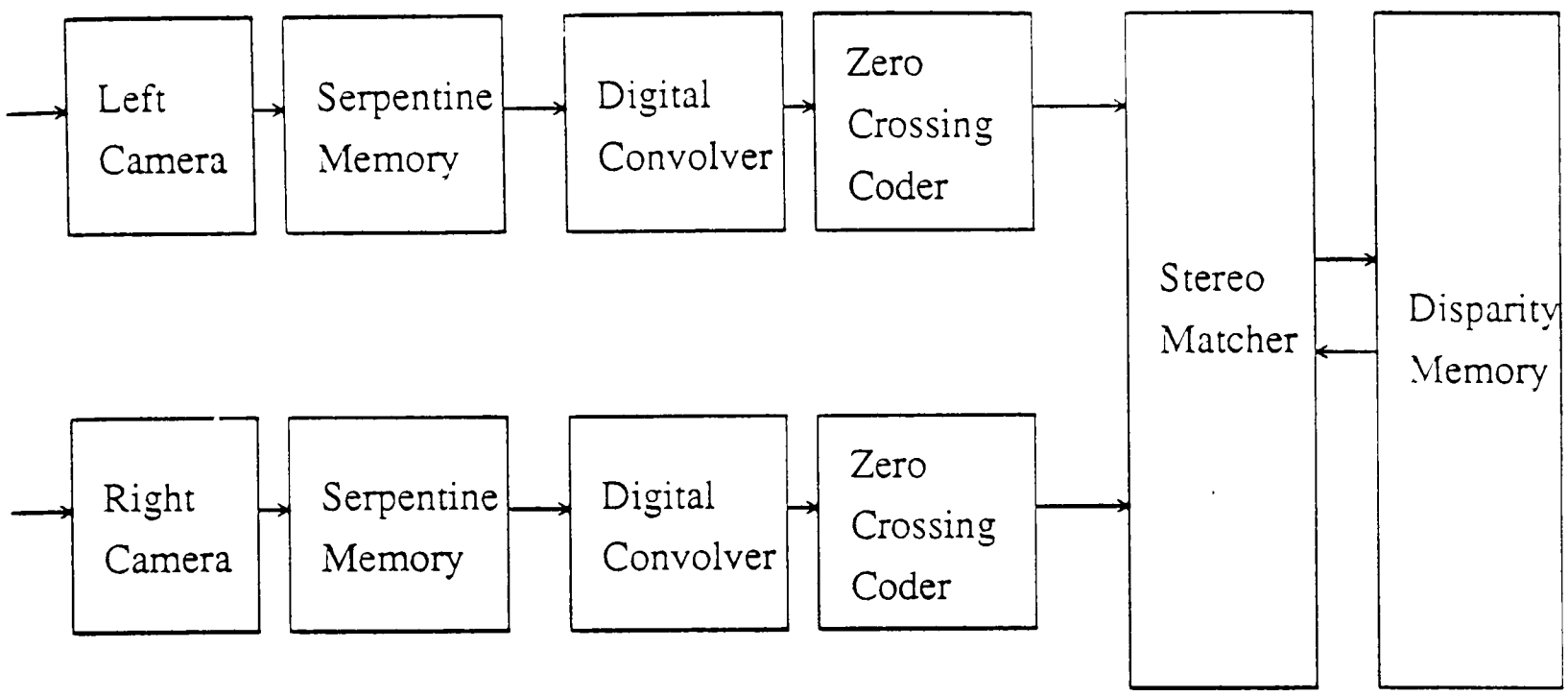


Figure 5.10: Nishihara and Larsen's Gaussian Convolver uses a serpentine memory to store the incoming video image before passing it into a digital convolver, which performs a difference of gaussian convolution by separating the convolution into two successive one-dimensional ones.

(see the Parallel-Pipelined convolver in Chapter 7). It is also of use to the vision computer as, by using such methods, the bandwidth between the image memory and the processors may be significantly increased, whilst not requiring faster memory devices or significant extra hardware. The design vignettes presented in the next chapter use interleaving to increase memory bandwidth, as do most of the current high performance general computers, such as the CRAY series. We will now look at a couple of ways of achieving this extra memory bandwidth. There are two main modes of interleaving, *S-access* and *C-access*. Both of these have been studied in detail, and their performance analysed (see Hellerman [1967] and Knuth [1975]). We will now look briefly at these two methods.

S-Access Memory Interleaving

S-access interleaving stands for Simultaneous access. In this model, low-order interleaving is used. The memory is organised in $M = 2^m$ blocks (see *Figure 5.11*). The

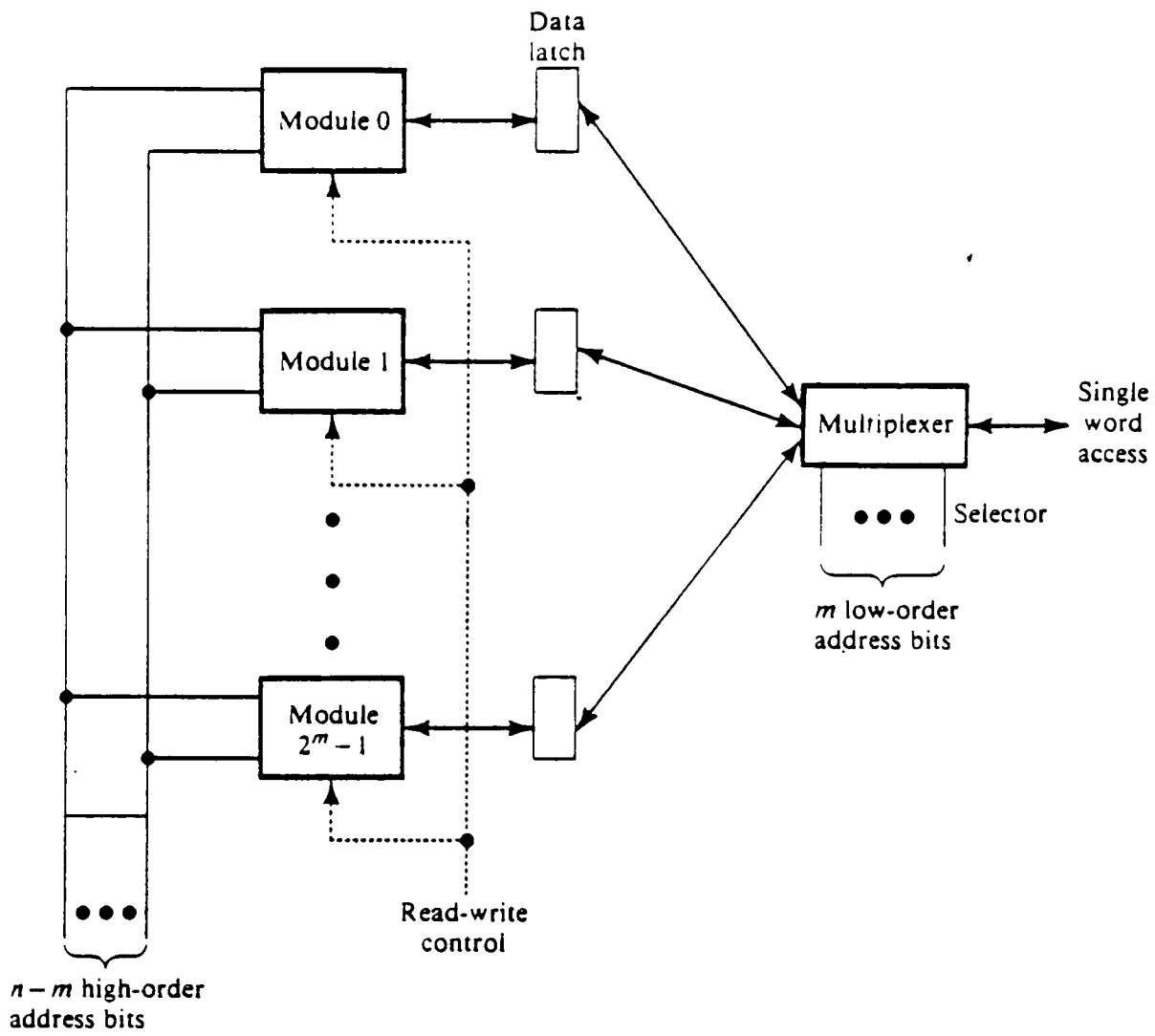


Figure 5.11: The S-Access interleaved memory configuration. Often used to speed memory access. The memory is organised in blocks, all of which are simultaneously addressed. The low order address bits then enable one particular module.

higher $(n - m)$ bits of the address are applied to all M memory modules simultaneously. This returns M consecutive words which are latched into registers. Using the low order m bits the data from particular modules may be accessed. With a memory access time T_a and latch time τ, the time to access a single word is $T_a + \tau$. However, to access k consecutive words in sequence, where k is less than M, the time taken is only $T_a + k\tau$, making this mode very useful where consecutive access of data is required, ie for use with many types of image data (or CPU instructions, for which the method was developed).

C-Access Memory Interleaving

The C-Access method is a Concurrent access method. It is used more often in general computers, but we mention it here for comparison with the S-access method,

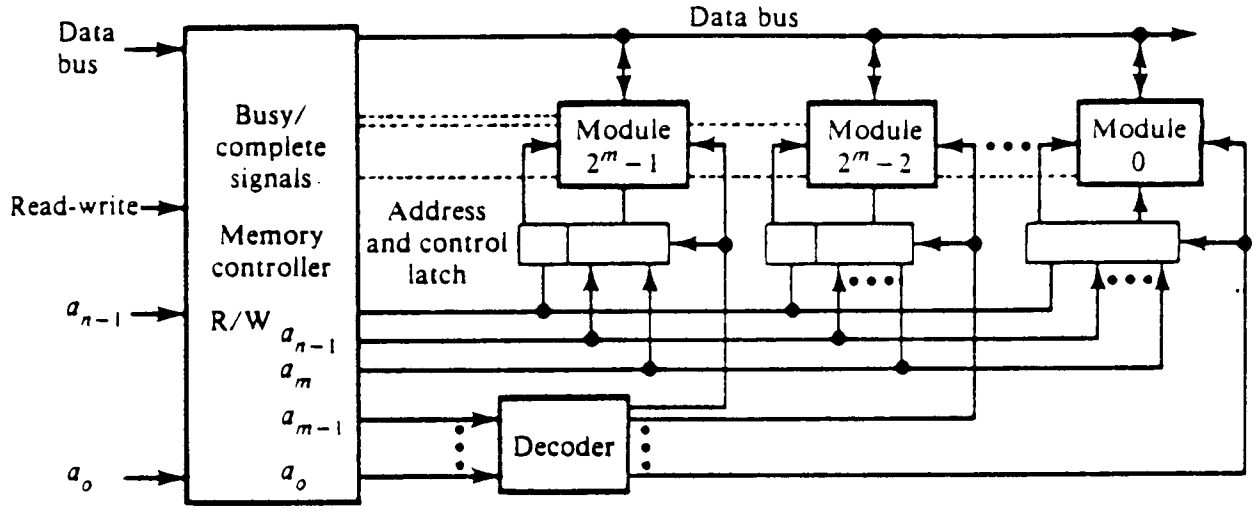


Figure 5.12: The C-Access interleaved memory configuration. Another way of speeding memory access, this method accesses memory modules concurrently. Each memory bank is active for a shorter time than a module, thus allowing memory banks to share access to a module.

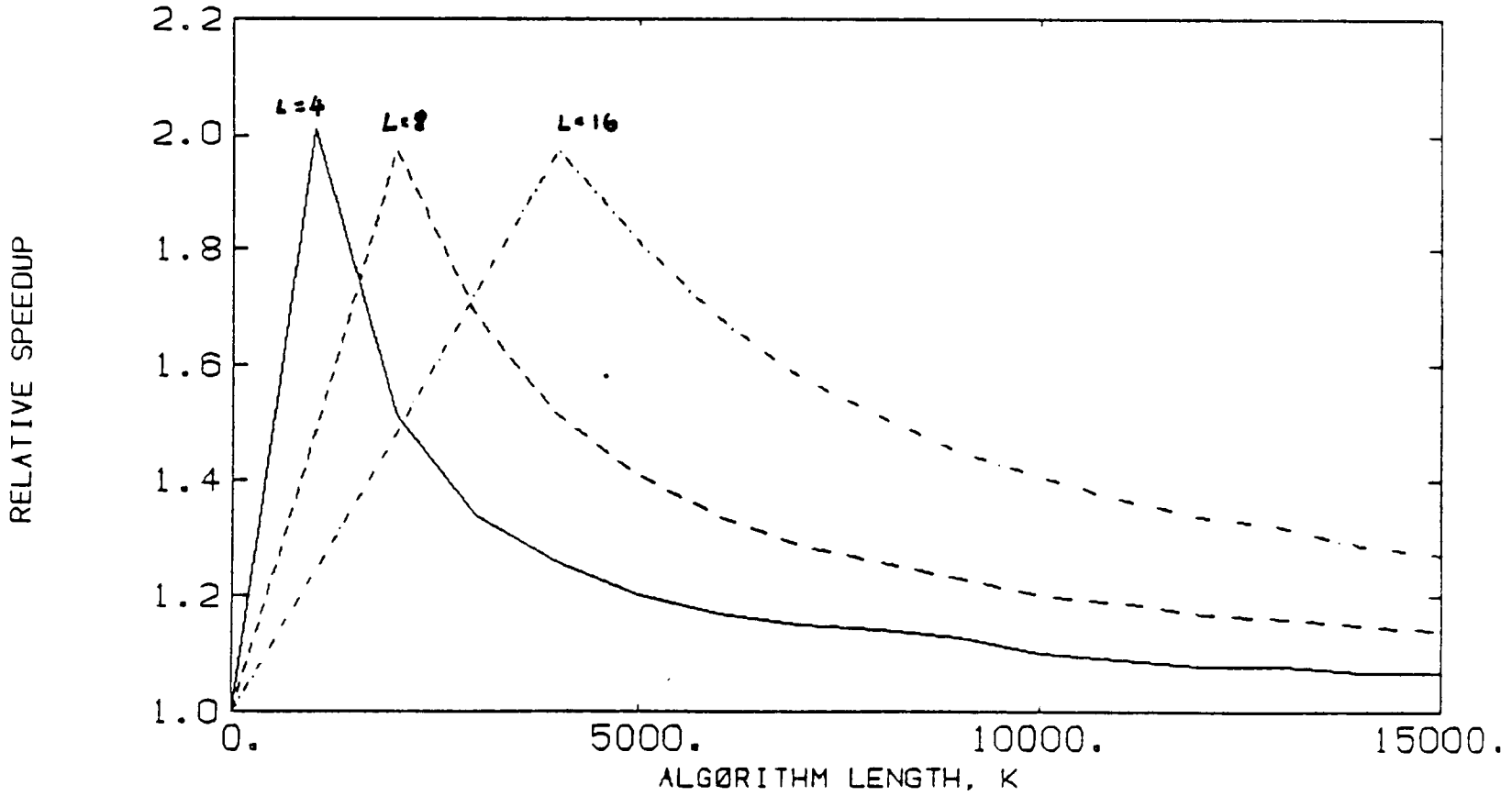


Figure 5.13: Relative speedup of overlapped column I/O with processing and column I/O vs algorithm length for differing values of bit length L. The relative speedup is that obtained by overlapping the array I/O operations with processing, over the time taken to perform these operations separately.

and also because hybrid interleaving is sometimes employed. This is shown diagrammatically in *Figure 5.12* . In this method, when a memory operation is initiated in a module it causes the bank to be active for T_a seconds, and the module to be active for T_c seconds. If $T_a << T_c$, the initiated module uses the bank for less than one memory cycle for access. Therefore, more than one module can share access to a bank. The low order m bits are used to select the module, and the remaining $(n-m)$ bits select an element from within that module.

This method is effective in addressing the elements of a vector. Consider an s-element vector $V[0:s-1]$ in which every other element is accessed, ie the skip distance is two. If we assume that element $V[i]$ is stored in module $i(\text{Mod } M)$ for $0 < i < s-1$. Then after the initial access, the access time for each sequential element is one per every 2τ seconds, where $\tau = T_a/M$. In general, if an address sequence is generated with skip distance d, and there are M modules arranged in a C-Access configuration, such that M and d are relatively prime, then elements can be accessed at the maximum rate of T_a/M per word. A comparison with the S-Access scheme, shows that for an address generated with skip distance d, the average data rate is dT_a/M when $d < M$ and T_a when $d > M$.

This method could be potentially very useful for hierarchical vision algorithms, where operations are performed first on a coarse grid (ie the skip distance on a row vector would be high) and then on a fine grid, where the skip distance would be less.

The scheme can also be easily expanded to more than one dimension. For a two dimensional array $A[0:R-1,0:C-1]$ the index element in vector $A[i,j]$ is given by $iC+j$ (this is called column-major form) or $jR+i$ (this is row-major form).

Other Memory Methods

Other designs have also attempted to ease the memory bandwidth problem so as to provide smooth data flow. An increasing number of current designs use dual port

RAM chips. Thus in an array of processing elements, each PE would be connected to one set of RAM ports, while the other ports would be connected to a high speed I/O bus. This again attempts to achieve temporal overlapping of memory functions.

As already mentioned, many existing designs overlap memory fetch with processing. One classic example of this is the MPP. We can quite easily show the speed up that is obtained quite easily. As shown previously the processing times for an array machine using column-parallel I/O and column-parallel overlapped with processing are

$$T_{a,c} = [2L(n+1) + K](\frac{M}{m})(\frac{N}{n})T_c \tag{5.26}$$

$$T_{a,co} = [2L(n+1)](\frac{M}{m})(\frac{N}{n})T_c \quad (K/Ln) = k < 2 \tag{5.27}$$

$$= [2L + K](\frac{M}{m})(\frac{N}{n})T_c \qquad k \geq 2 \tag{5.28}$$

We define a relative speedup RSU as

$$\text{RSU} = \frac{T_{a,c}}{T_{a,co}} = 1 + \frac{K}{2L(n+1)} \simeq \frac{1+K}{2} \quad k < 2 \tag{5.29}$$

$$= 1 + \frac{2Ln}{K+2L} \simeq \frac{1+2}{k} \quad k \geq 2 \tag{5.30}$$

A plot of the relative speedup against algorithm length K for different values of bit length L is shown in *Figure 5.13*. The relative speedup peaks at a value of 2 when $K = 2Ln$ (here the array size $n = 128$). Methods such as this can thus be of significant use to the designer.

5.8 Multi-Dimensional Pipelining

Perhaps the biggest challenge to pipelining arises in extending it to two or more dimensions. As already discussed, the majority of designs for vision computers are now turning to two or more dimensional processing arrays, often with complicated interconnection networks (e.g. the Connection machine, see Hillis, [1985]). Together with the need for interface to two dimensional frame stores, this poses a need for pipelining techniques applied to two or more dimensional data.

Some simple extensions to two dimensions have already been made. A two dimensional array for performing ALU functions that uses pipelining has been built by Kamal [1974], and, more recently, a design has been proposed by Comon and Robert [1987]. In this context, an array pipeline is a two-dimensional pipeline with multiple data flow streams. The pipeline is usually constructed with a cellular array of arithmetic units (often suitable for VLSI implementation). One possible such design, proposed by Hwang and Briggs [1984], is illustrated in *Figure 5.14*. Each cell has three inputs a, b, c and three outputs $a' = a,\ b' = b,\ d = a \times b + c$. The array was designed to perform the multiplication of two dense matrices $A.B = C$. This design concept is very similar to the new generation of systolic arrays where the data is pulsed through a two dimensional array on a regular basis. Here the problem of getting the data into and out of the array still remains in many cases.

Whereas the above method shows one way of using pipelining to keep an array supplied with data, it still leaves the designer with the problem of how to get the data from the memory to the array at sufficient rate. In the above design there were two input and one output data streams from the array. This would require a memory that allowed three-way parallel access to its contents. Such a design could perhaps be accomplished by interleaving the memory as previously discussed, or alternatively by having a memory that ran at three times the array speed. This latter option would be difficult to achieve, as the present maximum memory access rate is about 35ns, limiting our design to an array rate of 105ns.

Thus it appears that memory interleaving would have to be used in this case to provide a sufficient memory-array bandwidth, with dual-port memories possibly used to speed access further. The designer's problem is then changed to how best to arrange and format his data so that the memory can be effectively interleaved. Using this method, the pipelining into the array could easily be extended to three dimensions, with of course an added complexity imposed upon the memory design. Unfortunately, this then places a severe interconnection problem for the designer.

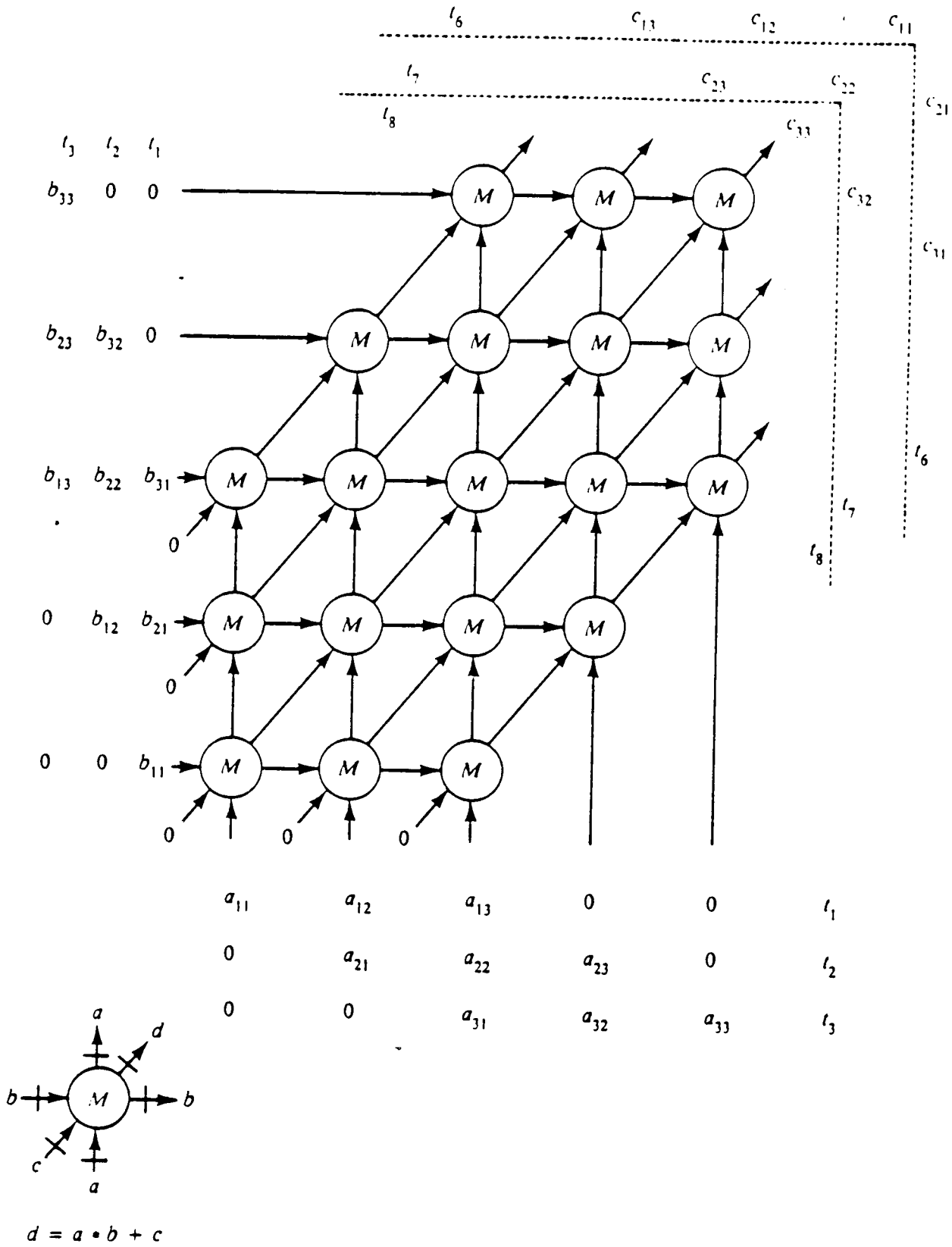


Figure 5.14: A cellular array for pipelined multiplication of two dense matrices. This two dimensional pipeline is very similar to the concept of a systolic array

Once more we see that the bandwidth problem has changed from being one of inappropriate architecture to one of physical interconnection density.

5.9 Direct Camera Interface

One way in which part of the interface problem may be solved is to integrate the processing electronics with the sensing device. Although current CCD cameras store data in an array format, they then lose advantage of this parallelism by outputting the data in serial format. This suggests that it might be possible to interface with the sensor more directly so as to increase processing speed. Knight [1983] has recently explored this idea. He noted that CCD cameras are basically analogue devices, and that the p-n junctions used in the integrated circuits behave as capacitors. He used analog processing on the sensor board itself to produce an edge map directly from the camera output. This is a very powerful design concept, as reducing the data at the camera source is the easiest way to decrease the bandwidth demand later in the processing.

Knight produces his edge map by convolving the image with the Laplacian of a Gaussian function and looking for the zero crossings of the output, e.g:

$$\nabla^2 G * I = 0$$

As we have seen, this can be approximated by convolving with the difference of two Gaussians

$$(G_{\sigma 1} - G_{\sigma 2}) * I = 0 \tag{5.31}$$

$$\Rightarrow G_{\sigma 1} * I = G_{\sigma 2} * I \tag{5.32}$$

Knight used an array of resistors and capacitors connected to the CCD cells and allows the charge to diffuse over the array for a time proportional to $\sqrt{\sigma_1}$. The charge in the photodiodes diffuses through this R-C circuit according to the diffusion equation

$$v = V_0 e^{-t^2/2RC} \tag{5.33}$$

Thus $\sigma = \sqrt{RC}$. Sample and hold devices then sample the charge distribution at this time T_1. The charge is then allowed to diffuse further to a time proportional to $\sqrt{\sigma_2}$ and is then resampled. The difference between the two sampled signals is then a function of the edge map of the image.

Using this method, Knight produced an image edge map at video rates directly from his camera, thus integrating some processing within the camera itself, and also reducing the data bandwidth necessary between the camera and further processing devices. This design is very promising, and it may be the best direction for front end vision processing to follow. However, the problem that Knight faces with his design is that I.C. technology cannot currently support a realistic image size (ie 512×512) with this sort of sensor. Indeed Knight's sensor was only 8×10 pixels, obviously too small for practical use.

The designer might also consider outputting the camera data directly in digital format. The current data rate for internal data transfer within a CCD chip is about 20MHz. Using a 20MHz flash A/D converter (just possible) on the camera itself, and outputting the data as bytes at 20MHz, would lead to a data rate of 160MBit/s which could be achieved using a fibre optic link. The 256K bytes of a 512×512 sensor could then be transferred in 13ms instead of the usual 16.6ms for 60Hz video, giving extra processing time between frames (enough for 30000 operations at a 100ns clock rate). The main problem with this method would be the A/D converter, which forms a bottleneck between the CCD array and the digital memory. In the ideal case each CCD cell would be connected directly to a digital memory cell via a converter, thus allowing the sensor to output serial analog data or parallel digital data by memory mapping the digital sensor store to the array PE-RAM's. Unfortunately at the moment such a design would be impossible because of technology constraints.

Chapter 6

DESIGN VIGNETTES

6.1 Introduction

We looked in Chapter 2 at a selection of intermediate and low-level vision algorithms, and found that most of the operations required in the calculation of these algorithms were in the form of local neighbourhood convolutions. The emergence of such algorithms led to the development of a new class of array processors, which use large networks of simple, usually bit-serial, processing elements. By exploiting this parallelism, it is possible to obtain significant performance gains over a serial computer executing the same program.

However, we showed in Chapter 4 that the design of such array computers does not completely solve the vision computing problem. This is because of the requirement to transfer large amounts of data to and from the actual processing array, and at some stage, to interface with a serial camera and TV monitor. Under certain conditions, the time taken to perform these I/O operations can completely dominate the processing time. As expected, any serial data transfer here, either from a camera, or between frame stores and array, can increase the total processing time by orders of magnitude.

In the last chapter, we noted that pipelining is the main technique for overcoming data bottleneck problems in serial computers. It was seen that the same techniques can be applied to array computers for vision, to increase the rate at which these local convolution operations can be performed. We follow this with a discussion of

three designs for vision processors, all of which apply pipelining (in differing ways) to the problem of executing this type of vision algorithm.

6.2 A Serpentine Array

This array processor was designed to perform the type of iterative local operations that were described in Chapter 2 at video rates. It is optimised to perform 3×3 or 5×5 filtering on images to provide local support for algorithms such as Canny's edge finder, optic flow, stereo etc. I designed this array processor around the following aims:

- It should be based around a serpentine array approach.
- With the additional aim of using a minimum of hardware.

Figure 6.1 shows the design. It consists of an array with as many columns as there are columns in the image to be processed. The array has a number of rows equal to the number of rows in the required neighbourhood size plus 2 (or more). For example, if a 3×3 operation was required on an array of size 512×512, then the computer would need to have at least five 512-element rows. The array is designed to perform neighbourhood operations of the sort mentioned in Chapter 2 on image data, and works as follows:

1. Data is read in pixel-serially (ie from a camera) into the top row of the array.
2. Once this row is full, the data is then clocked down by one complete row (during camera horizontal retrace) and the next row is read in. If the required neighbourhood size is 3x3, data is clocked in in this manner until the three rows beneath the input row are full.
3. When these three rows are full, the required operations are applied in parallel over the rows.

4. Once this stage is complete, the rows are clocked down one row further to the output row, and this row is read out serially to the display device whilst another row is clocked in serially to the input row.

5. Processing and I/O thus continue in overlapped fashion.

One of the main ideas behind the design is also to keep down the cost of hardware, so usually the processing rows would be limited in number to the neighbourhood size, but the system would obviously work with more rows, allowing processing to take place on two or more row neighbourhoods at once.

An analysis of the design shows it to be quite efficient for the operations for which it was intended:

Let z be neighbourhood dimension

T_a be array cycle time

Then to load the array we find

$$T_{in} = zNTio + zT_a + T_a$$

where we load the input row, and then drop by one row z times. The extra T_a loads the input array for the next time.To output the image from the array we find that:

$$T_{out} = T_a + NT_{io}$$

In this analysis we assume that we clock the last processing row down to the output row, and then read it out serially. It follows that the execution time is KT_a where K is the number of algorithm steps. Execution and I/O now happen in overlapped fashion. Every KT_a seconds, the array inputs another row and clocks it down. Processing must stop as the row is clocked down, so the total time to process a row is $KT_a + T_a$, the extra T_a being the clock down time. Thus the total processing time for an $M \times N$ array is

$$T_{tot} = \{zNT_{io} + zT_a + T_a\} + \{M(KT_a + T_a)\} + \{zT_a + zNT_{io}\} \quad (6.1)$$

$$= (KM + 2z + M + 1)T_a + 2zNT_{io} \quad (6.2)$$

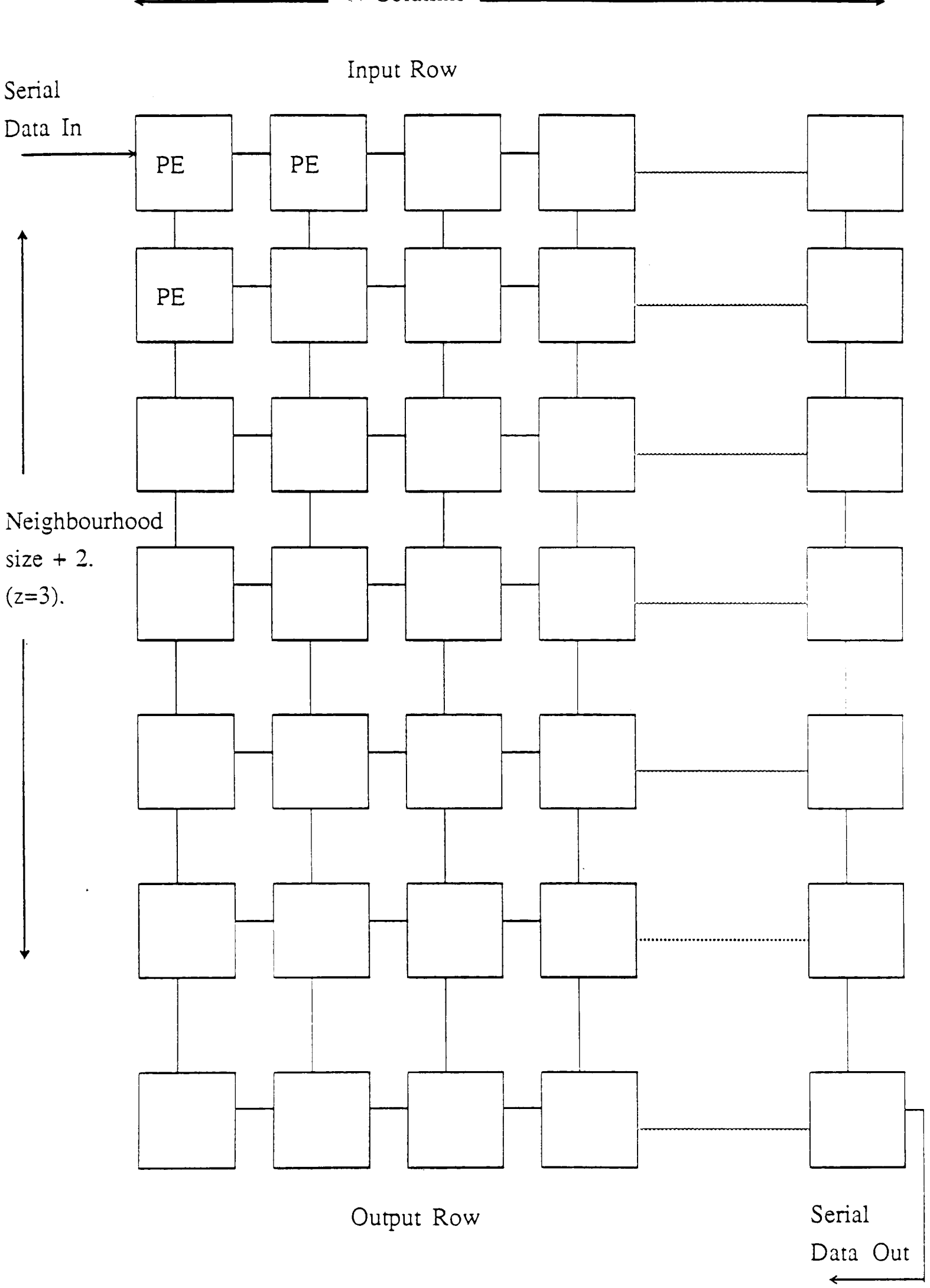


Figure 6.1: Serpentine array computer. This shows a way of using a serpentine memory to input data from a serial device, and to process this data whilst in the memory, before it is clocked out again.

For the case when $M = N = 512$, the neighbourhood size=3, $K = 500$ steps, such as in a Gaussian smoothing, which we have used in previous examples, and letting $T_{io} = 2T_a$ then

$$T_{tot,500} = 262663T_a$$

We will also calculate the time for a simple operation such as a Sobel filter with $k = 10$ steps. Here we get

$$T_{tot,10} = 11783T_a$$

We will calculate the time for the same operation on a standard 128×128 array to compare the performance of our vision machine with a "standard" machine. We will take as our example a machine using column-parallel I/O and storing the image before and after processing as would have to be the case with this I/O method. We get (using equation 4.12)

$$T_{500} = 1639156T_a$$

for $k = 500$ steps and

$$T_{10} = 1638666T_a$$

for $k = 10$ steps. The processing time for the new array is thus only 16% of that taken on a very large parallel array, and that for a 10 step algorithm is only 0.7%, and for a far smaller hardware cost. This illustrates the fact that if a large parallel array is constrained in its operation by I/O time, its performance can be significantly improved by a smaller array that is more highly optimised for I/O operations.

My design is only effective however, when

- The target and starting data formats are byte serial, a reasonable assumption as all current CCD cameras and TV monitors use a byte serial format.

- The image does not need to be processed on a recursive basis, which is sufficient for most vision algorithms, very few algorithms requiring the original image later in processing.

Making the assumption that the necessary processing can occur within the time for one video line plus flyback (not the case if data is coming from and going to a single port memory, as would be so if the machine were connected to a serial line). This demand gives approximately $64\mu s$ for processing, which gives a maximum 640 cycles on a 100ns cycle time.

This would perhaps pose the greatest problem to the user of the array, in limiting the number of operations possible. One way around this would be to add further stages to the array, and pipe the image through a further z rows, thus doubling the available processing time whilst delaying the output image by one frame. Clearly, an alternative design which counters the objection would be available on a modular basis, with "plug-in" row modules, allowing the user to expand the system for different neighbourhood sizes and processing requirements. One aspect of the array that I have not mentioned is the ability to deal with interlaced video. At present it is envisaged to use a non-interlaced format, but interlaced I/O could be provided for by using extra rows in the array, whilst still offering a significant performance advantage over a standard design.

The current design provides a low cost modular processing system, highly suited to iterative local support in vision processing. The user would expand the system according to his individual needs. The major benefit would be an increase in performance over standard array machines for algorithms which can be performed using local operators.

6.3 A Dynamic Memory Array

The previous design attempted to solve the memory-processor bottleneck by increasing the I/O bandwidth to the processing array. This next design attacks the other side of the problem, ie the memory I/O bandwidth. This problem occurs in two ways, the first being the access speed of the memory, and the second the degree of parallelism within the memory itself. Because most memories are single port, it

requires a significant amount of interleaving to provide any degree of parallel access to the memory.

With current technology, the minimum access time for a bipolar static RAM is about 25nS, on a chip that uses internal ECL circuitry. This allows the designer four memory accesses from such a chip within a 100nS PE cycle time, which is the current benchmark. However, the annual decrease in memory access time appears to be slowing, and it is unlikely that memory chips will become significantly faster than this (they will however continue to become larger). One possible way around this is to use a register memory, ie to convert our vision machine from a memory-memory architecture to a register-register one.

Such a design would work as follows. The memory of a 512×512 frame store would consist of a corresponding number (256k) of registers, arranged either as a linear structure or possibly as an array. This seems very similar to a design such as the Connection machine. However, it is not intended to actually have any processing capability in the design, which is a pure memory device. Obviously processing could be incorported directly into the design if sufficient hardware technology was available. Otherwise, column, if not image, parallel I/O should be used, thus making the array effectively three dimensional (x,y, and down to the processing elements).

The array or line would be arranged so as to have wraparound at the ends, so the data would continually circulate around the memory. This would be done so that it was possible to reformat the data within the memory, (i.e. to translate an image spatially), to format data before transferring it to the processing array, to input or output data in any format (e.g interlaced or non-interlaced) and other similar operations. Because of the large number of registers needed, the design would have to be based upon a set of custom VLSI chips, each containing a number of registers. At the current benchmark of 10k gates per chip, and with say 10 gates per register, each chip could hold up to 1000 registers, requiring 256 chips for our frame store. As technology pushes the gate/chip count higher, so the number of

chips would decrease. This would, however, lead to increasing complexity in the interconnections required between memory chips.

What would be the advantages of such a "dynamic memory" for a vision machine ? The first one would be speed. If the registers were ECL ones, register-register transfer time could be of the order of a few nanoseconds, ie about a factor of ten faster than the current fastest memory. Perhaps a more important benefit still would be in the organisation of the registers themselves. Because the registers could be arranged in any format appropriate to the data being processed, they could be optimised for high speed access. As well as this, parallel access to such a memory array would only be limited by the number of physical connections that could be made between the memory and PE arrays, rather than by access to the memory itself. In a standard video memory, the number of elements that can be accessed directly is limited by using, at most, dual-port memory. This means that the memory must be interleaved. By using registers, any number of elements could be accessed in parallel, provided that a suitable interconnection network could be found.

The design of the memory would then be dependent upon the selection of a suitable interconnection network, from a simple serpentine array type connection to a hypercube or other complex topology. As the array is designed purely as a memory, the interconnections should be made as simple as possible, but there should be enough data channels to the memory to allow the processing elements to be accessed in parallel. If some form of switching network were incorporated, then a degree of programmability could be obtained.

This ability to dynamically reprogram the memory could prove extremely useful, for instance in storing interlaced video and then converting it to non-interlaced video suitable for processing, by altering the wrap-around connections on the array. As a less trivial example, the switching network could provide increased access rates when irregular pattern memory addressing is required, as when conducting a line

following process.

As proposed, this sketch of a dynamic memory array provides a way of obtaining very high memory access rates. Obviously such an array could form the basis of an array computer, by mapping either the entire array or a subsection of it onto an array of processing elements. Again the problem with this approach is that of the physical number of interconnections which can be made between the elements. This problem will decrease as integrated circuit technology advances, allowing more memory and/or processing elements to be packed into a single device. However, the problem of interconnecting the chips themselves becomes more complex as the register density is increased, leading to problems with pinouts etc. With current technology limits, providing a large number of interconnections between the memory and processor could prove impratical, but the flexibility of such a memory array could make it very useful for vision applications.

6.4 A Parallel-Pipelined Convolver

The final design addressed the following objectives:

- Low hardware cost
- Processing on stored images only
- Performance suitable for industrial applications
- Capable of executing most front end processing operations.

It was intended that the total cost of the machine including the camera and monitor should be less than £5000. It is designed for use in an industrial environment for tasks such as automated inspection and robot guidance. Because of this, it was estimated that the processing rate should be of the order of ten images per second, as frame rates higher than this are not frequently required. It was decided to operate on stored images, rather than on a real time video stream. This

also allowed the design to concentrate on minimising $T^2_{i/o}$ and to ignore $T^1_{i/o}$. The original specification also called for a machine which would provide simple front end processing such as Sobel filtering and other simple edge detection and noise removal operations. Since then this has been extended to cover support for most local (iterative) operations, thus expanding the potential capability of the machine to include many of the algorithms mentioned in Chapter 2 of this paper.

The machine interfaces to an IBM PC to allow further processing and will act as a fast pipelined convolver for computation intensive operations. Because of the specification for low hardware cost, an array-type processor was impractical, so it was decided to use a hardware convolver with a 3×3 window, and then to move this window across the image.

While the specification did not call for video rate processing, the aim was to decrease processing time as much as possible by using pipelining techniques to speed the data flow through the convolution window. To achieve this (see *Figure 6.2*) the system has two 512×512 frame stores, one for use as a data source and one for use as a data target (although processing can be achieved using a single frame store). The memories are 3-way interleaved, using the S-access method to output three bytes of data in parallel to the convolver. Maximum memory bandwidth is 100MBits/sec (see *Figure 6.3*)

Microprogrammed memory address sequences allow for interface to interlaced or non-interlaced devices, the PC I/O channel and control data flow through the convolver. The whole system is controlled by a bit-slice microcontroller,(see *Figure 6.4*) controlling a $4k \times 75$ bit microword memory. This runs at slightly over 4MHz, and controls banks of bit-slice DMA controllers, which generate the frame store addresses. The use of such a microword memory will allow the system to be expanded in future as new applications appear.

Although it has been necessary to use some quite complex hardware to generate memory addresses at the required speed, the system remains flexible because of the

use of address look-up tables. Thus different address sequences can be easily programmed and implemented at speed. Because of the attention paid to the memory addressing and interface, it is possible to keep the convolver board supplied with data at the maximum rate the the multipliers can sustain. This allows a full convolution over a 256×256 image in 31.5ms, or just within frame rate. Convolution coefficients are stored in a further look-up table, or may be downloaded from the PC. In addition to convolving an image with a coefficient window, the convolution board can threshold the image or select the maximum or minimum of two input pixels, allowing local thresholds and similar functions to be applied.

Whilst being fairly basic in its computational abilities, the design's 3×3 convolution format allows both the simulation of larger convolution windows, as, for instance, developed by Burt [1981]. He was concerned with the problem of passing a gaussian filter with a large window size, say 27×27, over an image with only local computational support, say a 3×3 convolution mask. In his paper he shows via the central limit theorem that by repeated iteration of a specific 3×3 local convolution mask it is possible to approximate to the larger Gaussian convolution to arbitrary accuracy.

Thus the design provides general front end processing power for an industrial vision system at near real time rates, yet at a comparatively low hardware cost. This has been achieved using pipelining techniques for the image memory, where memory interleaving is used to supply the parallel input convolver with data at sufficient rate, the instruction prefetch of the microcontroller, which uses a pipeline to double the instruction rate of the micorcontroller, and by piping stages within the convolver board, doubling the throughput of the convolver. This design has been built, and its operation is described in detail in the next chapter.

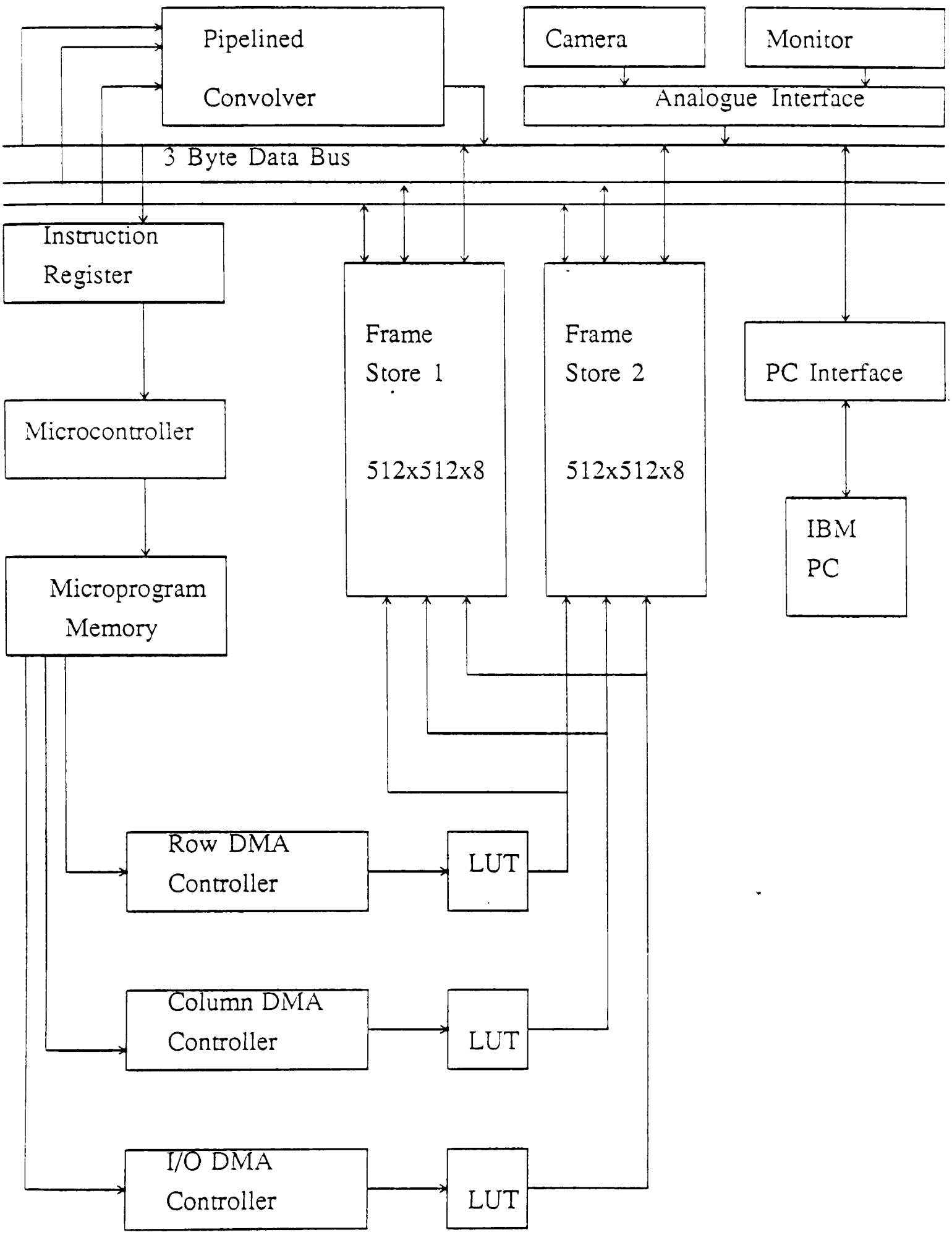


Figure 6.2: A Parallel-Pipelined convolver. A design for a general purpose convolver for vision use. Parallelism is employed in the convolver, and the unit is pipelined to a small extent to speed data flow.

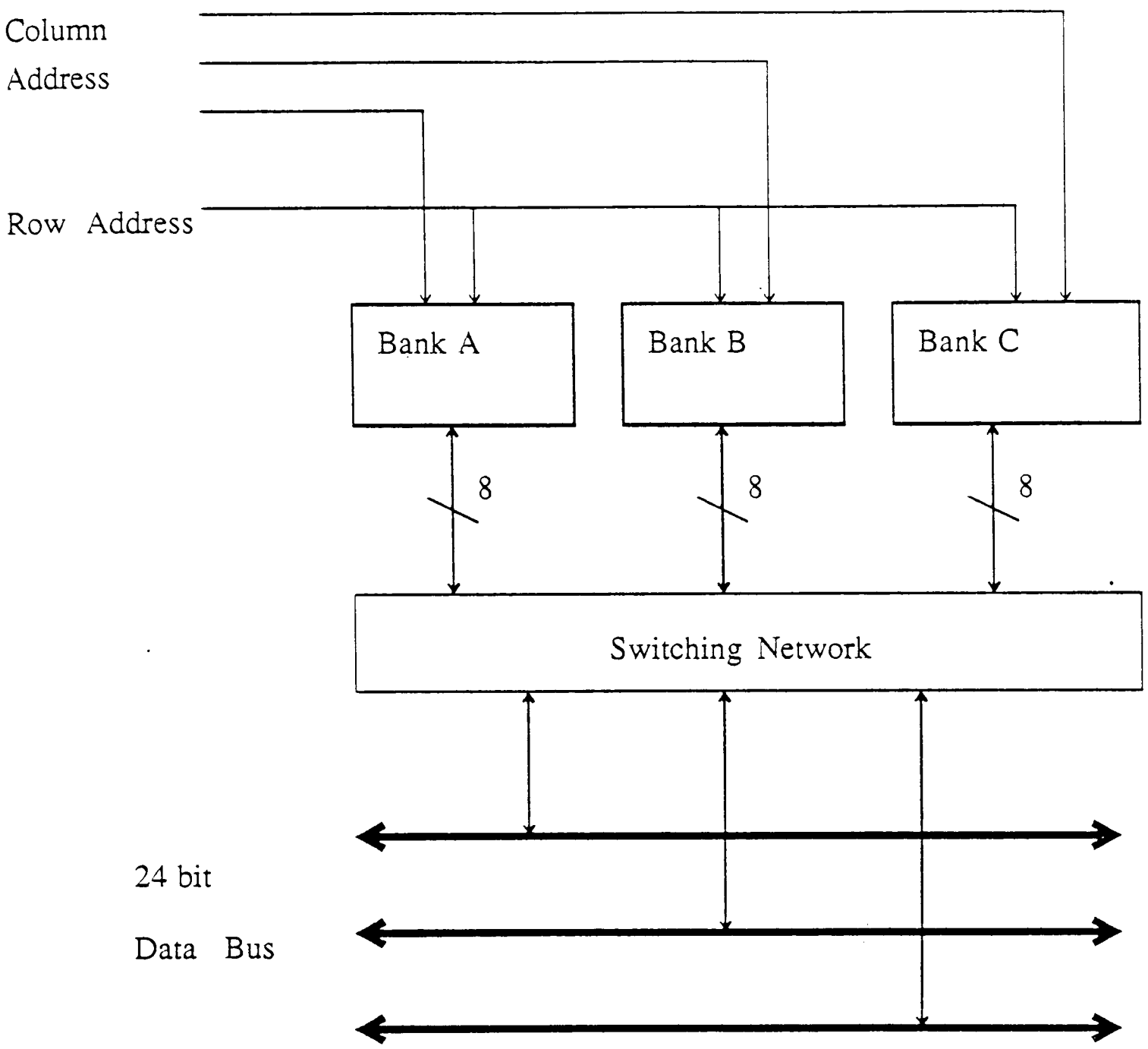


Figure 6.3: Parallel-pipelined convolver : memory structure. The memory of the device is organised in three banks, storing alternating data columns. These are accessed using a hybrid simultaneous access method so that high speed parallel (3 byte) access can be made to the memory.

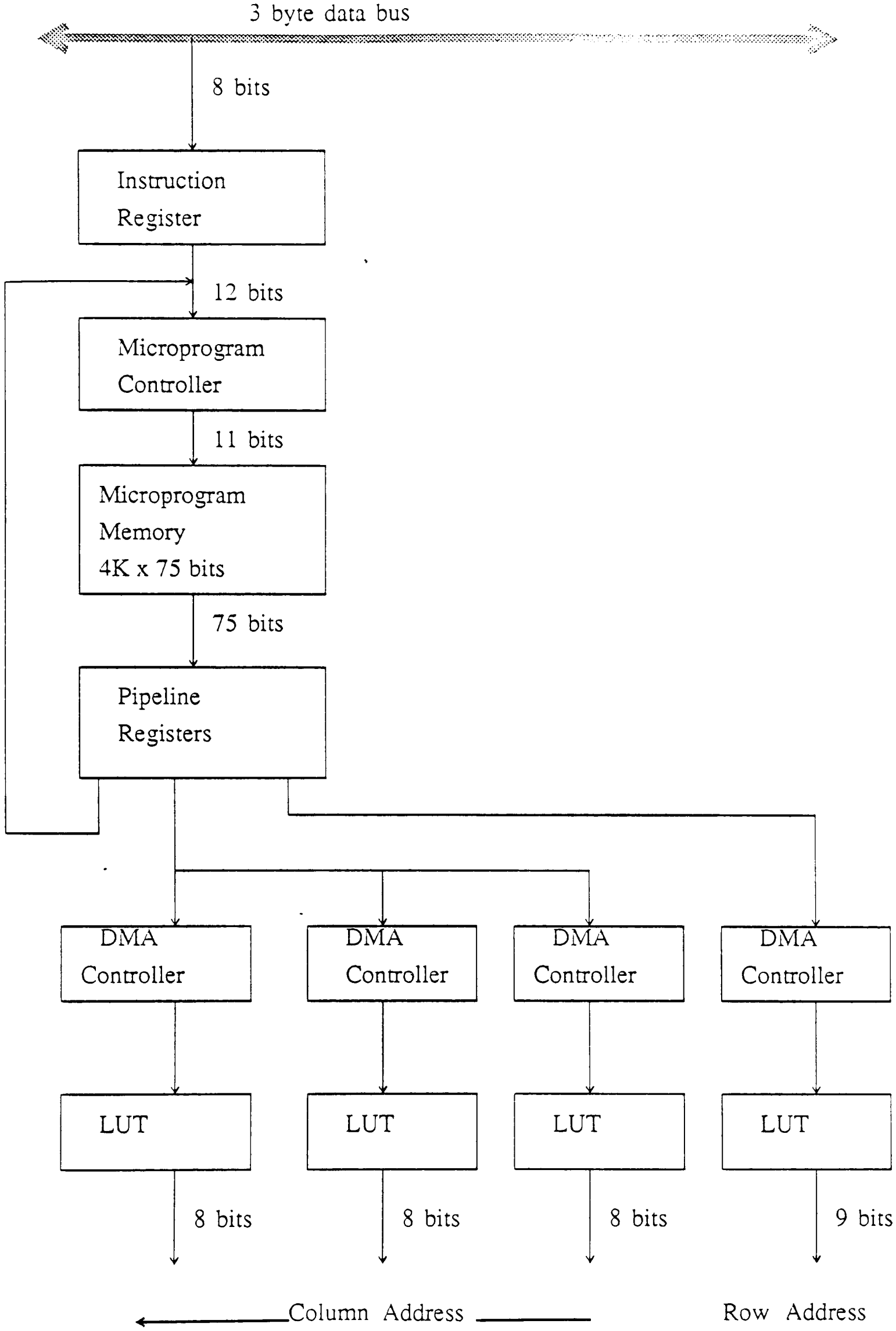


Figure 6.4: Parallel-pipelined convolver : system controller. The system controller for the parallel-pipelined convolver has an instruction look-ahead to increase throughput, and drives a microprogram memory.

Chapter 7

DESIGN OF A PARALLEL PIPELINED CONVOLVER

7.1 System Overview

In this chapter we decribe in detail the design and operation of the Parallel-Pipelined convolver that was featured as the third of the design vignettes in the last chapter. The system has been built in conjunction with IBM's UK Laboratories.

The system is designed to be a fast convolver of a digital image, the image to be up to 512 x 512 pixels in size, with each pixel at 8 bits of grey level resolution. Each pixel occupies one byte of memory, leading to an image size of 256K bytes. Convolutions are to operate on a 3x3 window of the image, this small window size being chosen to minimise the hardware requirements, while still allowing a very large subset of potential operations to be performed. In order to speed the convolution throughput, the convolver is designed to read in three pixels in parallel, and for this reason, the two image memories are each designed in three "blocks" to allow parallel access to 3 bytes. The convolver itself is pipelined to increase throughput, as is the controlling microprocessor.

At the time of the design (1985), there appeared to be no commercially available chips or cards on the market which could perform the main functional blocks of the system. The choice of controller for the system amounted to a choice between a standard microprocessor, such as a Motorola 68000, or a simple (typically bit–

slice) microcontroller. The standard microprocessor would offer a readily available software set (to make debugging easier), available busses and interface options, and a tested design format. The choice of a microcontroller would be more ambitious, but would mean less wasted processing power, a customised instruction set, the ability to design the system as a no–compromise solution to the problem at hand, and a possibly faster cycle time. As well as this, the microprocessor presents the user with a virtual machine, which does not allow the user to get below a certain level within the machine. This would make the study of pipelining far more difficult with this type of design. In the end, I opted for a microcontroller, as being both the fastest option once running, and as one which wasted as little processing power as possible, whilst allowing maximum flexibility to experiment with pipelining.

From this it seemed obvious that the system was going to have to be designed at a quite basic level. The memory addressing circuitry was also going to have to run at high speed, both to pass data to and from the convolver at real time rates, and to accept images direct from a camera (at $\approx$ 60ns/pixel). As I had already decided to opt for a bit-slice controller for the system, I decided to make the entire design "bit-slice", where the designer isn't restricted to a fixed bit-length, as is the case with a standard microprocessor architecture, but may decide on the bit-length which is optimal for the system.

The leading family of commercially available bit-slice devices appeared to be Advanced Micro Devices Am2900 series. This series contained the necessary sequencers, DMA generators, registers etc that a preliminary design involved in my system. The system is thus largely based around this family, and consists, as already mentioned in Section 6.3 of the functional blocks shown in *Figure 7.1.*

As already mentioned, the convolver has been designed to interface to an IBM PC. This was to be an important part of the system, both to act as a high level controller, and also to perform post processing on images which have been passed through the convolver (for this reason there had to be some form of image transfer

utility between the system and the PC). As a result, the eventual design consists of a three card set. The first card is a standard PC sized card, which takes up a full expansion slot in a PC-XT. This was necessary to provide easy system interface to the PC's I/O channel. The rest of the system could also have been built on PC cards, but due to the expected number of components, it was thought that this would lead to a large number of cards (indeed, probably enough to fill the entire PC!). Hence larger cards were selected. There are two of these "planar" cards, which measure $14'' \times 9''$. These two are linked via ribbon cable to each other, and to the PC card. The card set is illustrated in *Figure 7.2.* The PC card takes its 5V supply from the PC I/O bus, and has an external +/-12V,-5V supply. The two planar cards both have an external 5V supply.

The functional split between the cards is illustrated by the dotted lines in *Figure 7.1.* This split was motivated by the need to have all of the PC interface circuitry on the PC card, along with all of the clock generation. Because of the necessary analogue/digital circuits to provide system interface with cameras and monitors, it was also thought best to concentrate any analogue circuits on this card, so that the analogue signal paths would not be too close to the main data bus lines. Of the two planar cards, one was to be used for the two frame stores and for the memory address circuitry (keeping these together reduced the number of data and address lines linking the cards). This left the other planar card with the microcontroller and microprogram memory, and all of the convolver circuitry.

To summarise, the main logical units on the three cards are as shown in table 7.1.

In the following sections, we deal in turn with each of the three cards, detailing the design of each card, and finally we discuss the overall function of the system.

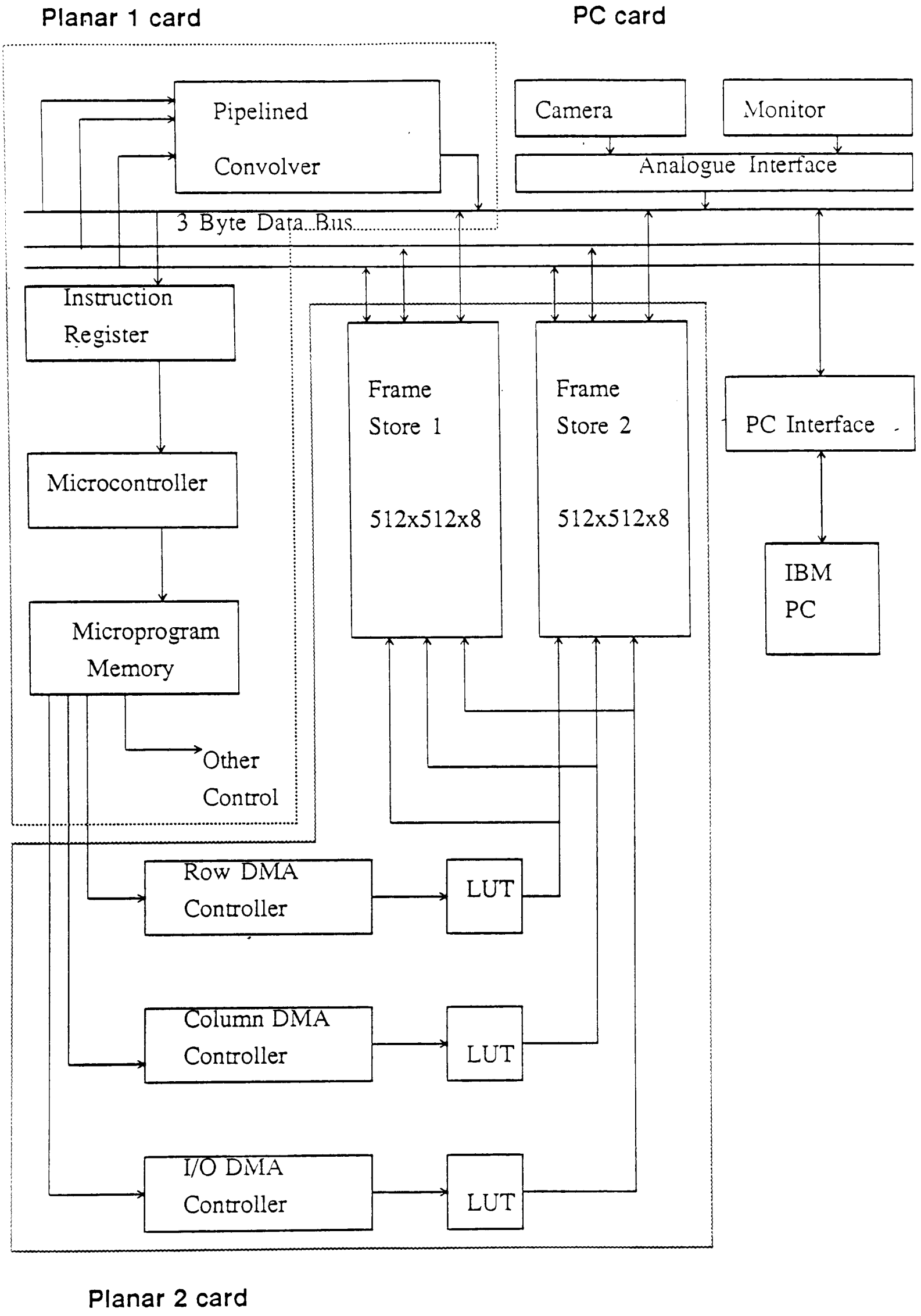


Figure 7.1: Functional Block Diagram - This shows the entire circuit on a functional basis, including the microcontroller, memory and addressing, convolver, PC and analogue interfaces

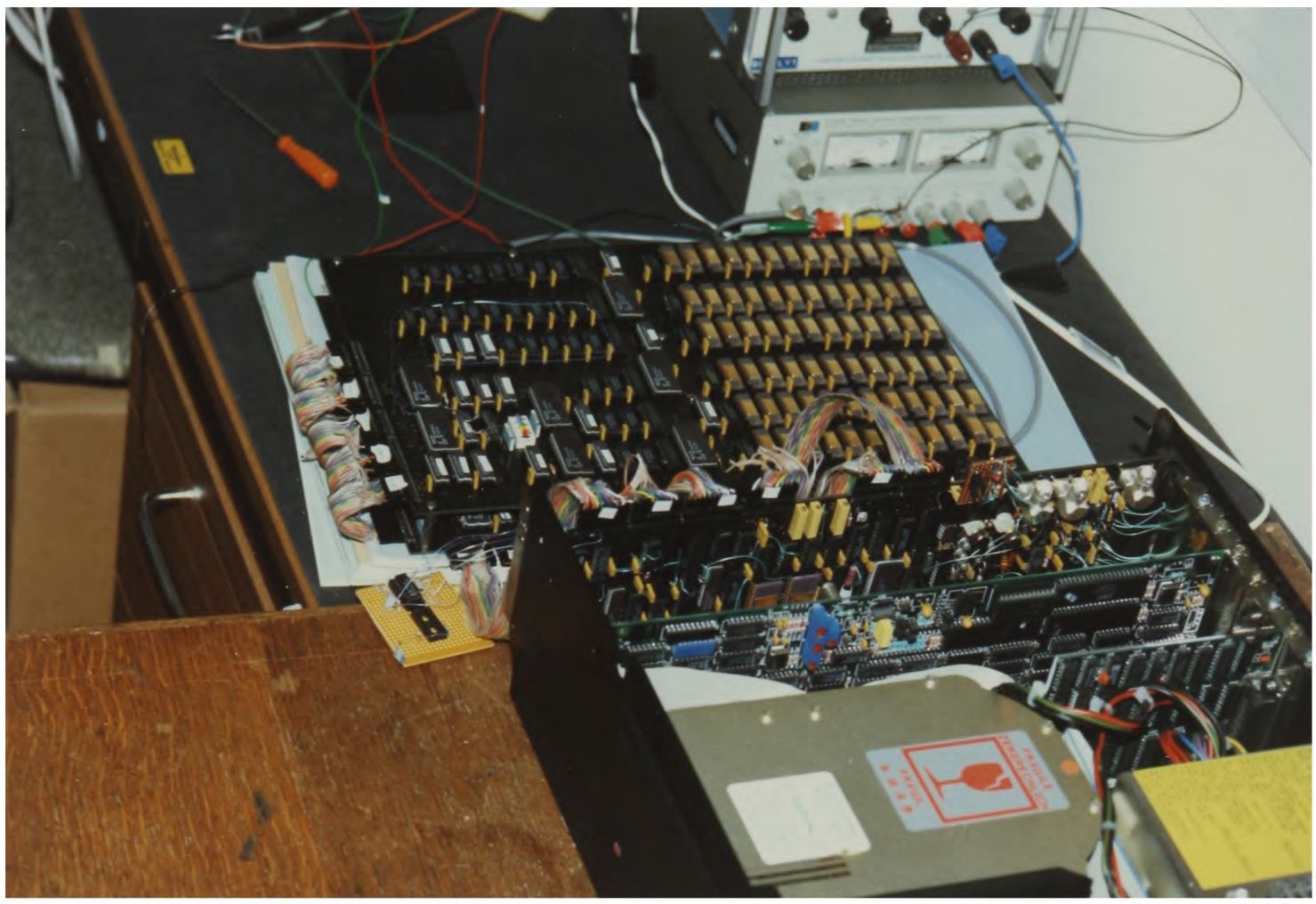
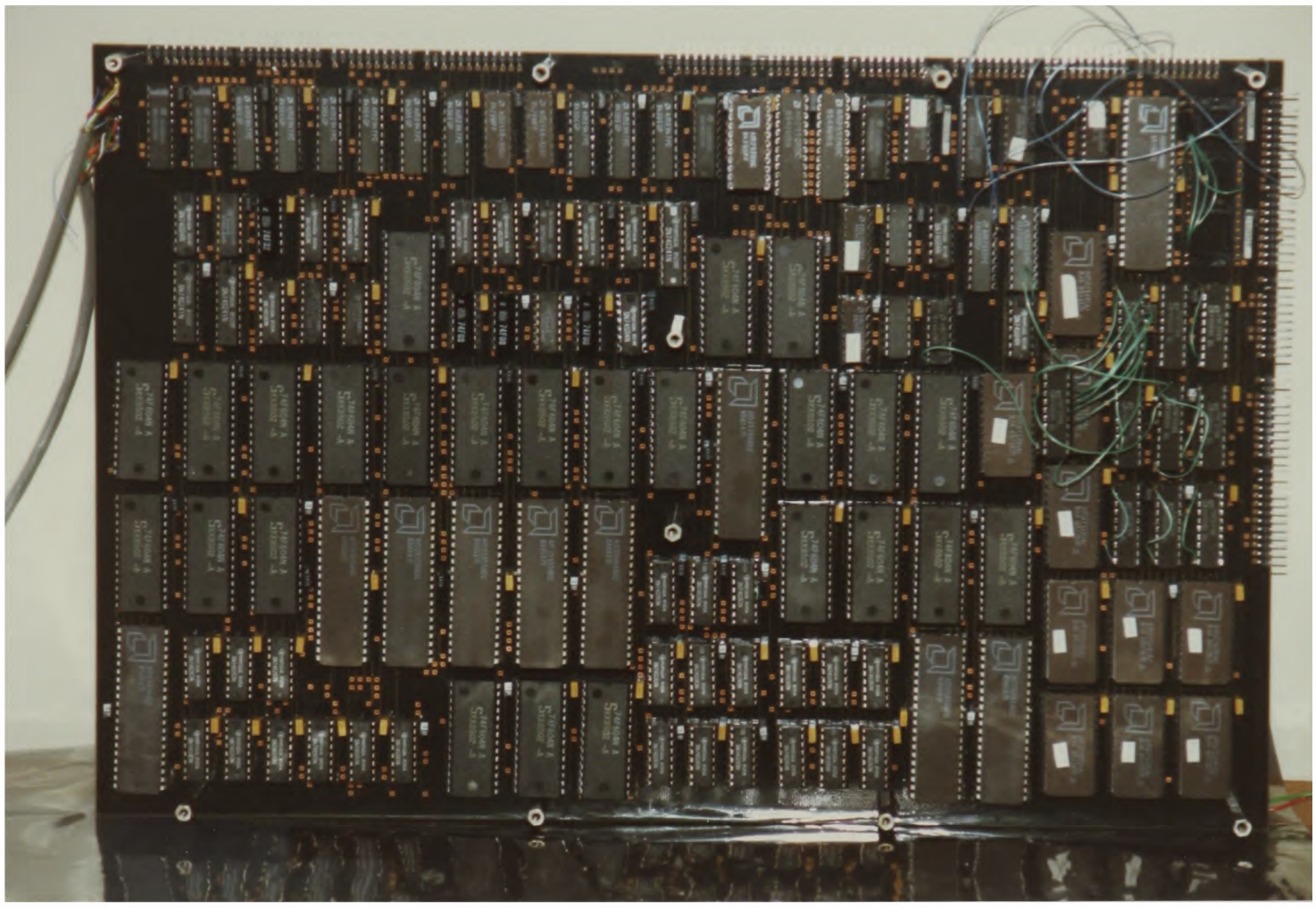


Figure 7.2: Convolver Card Set– shows pictures of the three cards that comprise the parallel-pipelined convolver, with their controlling PC, and of the PC card.



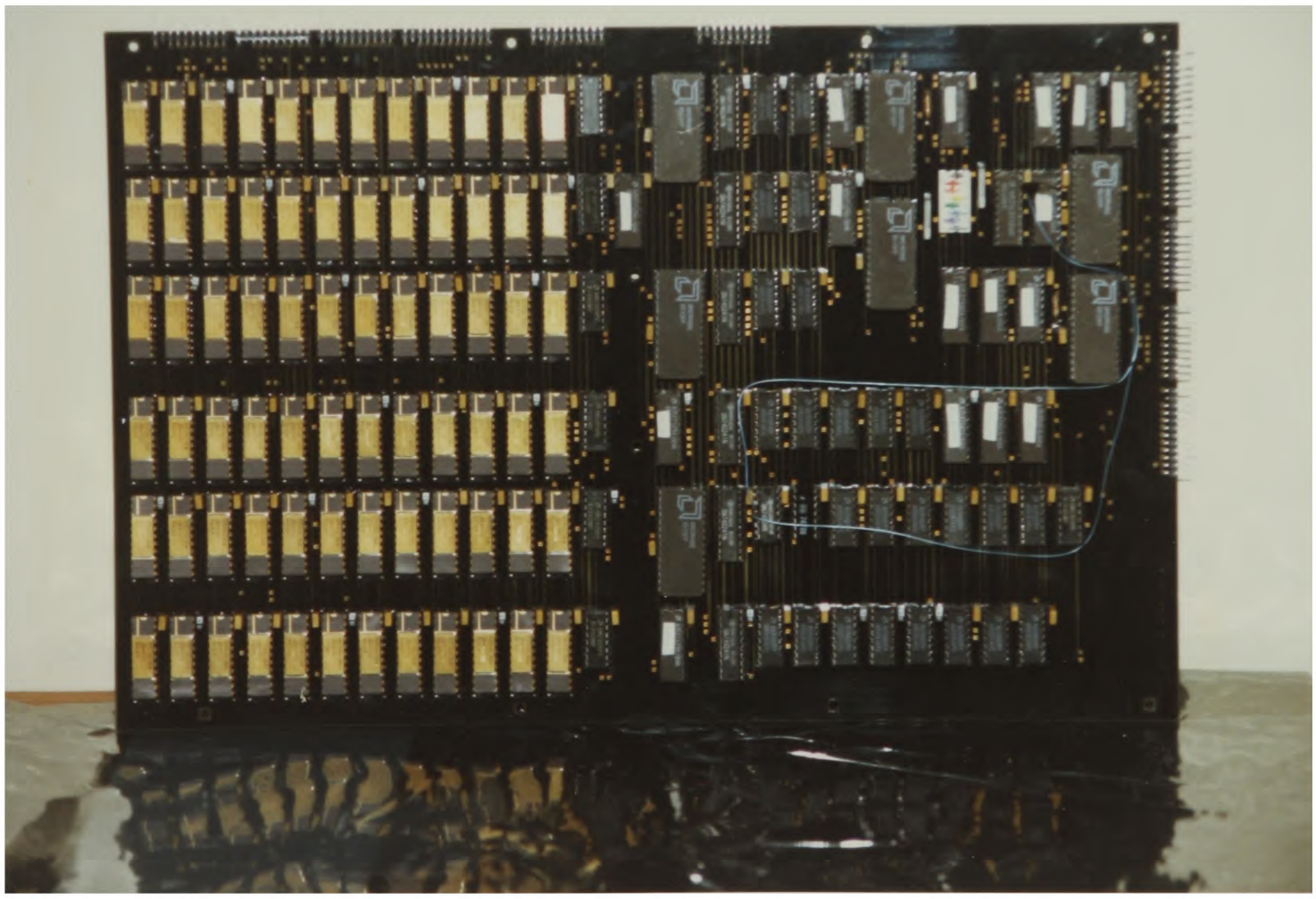


Figure 7.3: Convolver Card Set– Detailed pictures of the Planar 1 card (top) and of the Planar 2 card.

PC	Clock Generation PC Interface A/D and D/A converters TV Sync splitting and regeneration
Planar 1	Microcontroller and Microprogram memory Data Multiplexers Convolver
Planar 2	Frame Stores Address Generation

Table 7.1: The Physical card split for the Parallel-Pipeline Convolver

7.2 PC Card

7.2.1 Clock Generator

The choice of an AMD Am2910A bit-slice microcontroller as the main processor in the system limited the design to a maximum clock frequency of 20Mhz. However, I determined that the memory addressing and convolution timing would be more efficiently performed by running phased clocks with a lower frequency, so a maximum frequency of just over 4MHz (240nS period) was selected. This was mainly to avoid the crosstalk problems inherent in fast clock–cycle designs, and also to aid in debugging the system. From this 4MHz main clock three phased clocks were to be generated, thus giving an effective clock rate of almost 13MHz. This 240nS clock cycle would give a maximum of 166666 program steps within a video frame period. As the microprogram to convolve an image took 2 steps per pixel (plus a few steps for overheads) this would allow the convolution of a 256×256 image at video rates.

Two independent clock systems were to be used;

1. To control the microcontroller, the convolver and all of the digital circuits.

2. To control the analogue side of the circuit, which could then be gen-locked to an attached camera.

It was also necessary to set up the clocks so that the system could both be operated at below maximum frequency, and also so that it could also be single stepped, to aid debugging. The AMD series provided a solution to this problem

with the Am2925 programmable clock generator, which may be single stepped. It also provides a programmable set of four different clock signals from a master input system clock. The system was thus set up as shown in *Figure 7.4*.

The system clock signals are provided by an SN74S124 dual Voltage Controlled Oscillator(VCO). This is adjusted so as to give a range of output frequencies roughly between 6.6 MHz and 27 MHz (clock periods of between 150ns and 37ns). The system is designed to operate with this clock set to a maximum rate of 25MHz (a clock period of 40ns). With this clock set to 40nS, the main clock output from the Am2925 is 240nS, as designed. Two sets of delayed phased clocks are provided from this master clock signal delayed by 80nS and by 160nS. All timings, such as signal delays, quoted in this chapter are with reference to this 40ns clock period. If the system runs at a slower clock frequency, then the delays etc are correspondingly longer.

To provide the required clock signals, the output of the VCO is input to the Advanced Micro Devices (AMD) Am2925 programmable clock generator. This provides four separate clock outputs based upon the input waveform, which we will denote as C1M-C4M, these signals having a 240nS period as mentioned. The first of these signals, C1M is fed through a Mullard 74F164 register to delay it by 80ns, the output being C1M'.The fundamental frequency output from the AM2925 (F0) forms the input to a second Am2925. Outputs C1,C2 and C3 from this chip are fed through further pairs of 74F164's to provide signals delayed by 80 ns (C1',C2',C3') and by 160 ns (C1",C2",C3") – these provide the phased clock signals already described.

The first Am2925 is the master clock generator for the system. The second Am2925 is a slave clock generator, which is used when data is being input from a camera or output to a monitor. The slave Am2925 may be halted by a -WAITREQ signal and restarted on a -READY signal, these signals being derived from the camera. Generation of these two signals is described later in this section. When the

slave timing generator is halted, the master Am2925 provides timing for the rest of the circuit.

The notation used for the clocks in the system is shown in table 7.2.

Letter	Denotes
M	Master clock signal[1]
C*n*	True clock
CLK*n*	Inverted clock
'	Delayed by one F0M clock period
"	Delayed by two F0M clock periods
D	Inverted and delayed by one F0M clock period
DD	Inverted and delayed by two F0M clock periods

Table 7.2: System Clock Notation

Some clock signals are inverted as follows:

True	Inverse
C1	CLK1
C1'	CLK1D
C1"	CLK1DD
C2	CLK2
C2'	CLK2D
C2"	CLK2DD
C1M	CLK1M
C1M'	CLK1DM
C2M	CLK2M
C4M	CLK4M

Table 7.3: Inverted Clock Signals

The memory enables are also clocked from this part of the system. The timing of these enables is variable and dependent upon the operating state of the system. Hence the clock requirements are quite complex, in that the switch network may operate in either I/O mode (for analogue interface) or non I/O mode (for convolution etc). Three switch lines, -CHIPA/B/C must be derived. If in non I/O mode, these must be high if lines -CSA,-CSB and -CSC are high. However, if any of these go low, the appropriate output must follow clock signal C2'. If in I/O mode, then -CSA etc, are ignored and the output is dependent upon the read/write line, to

[1]No "M" indicates a slave signal

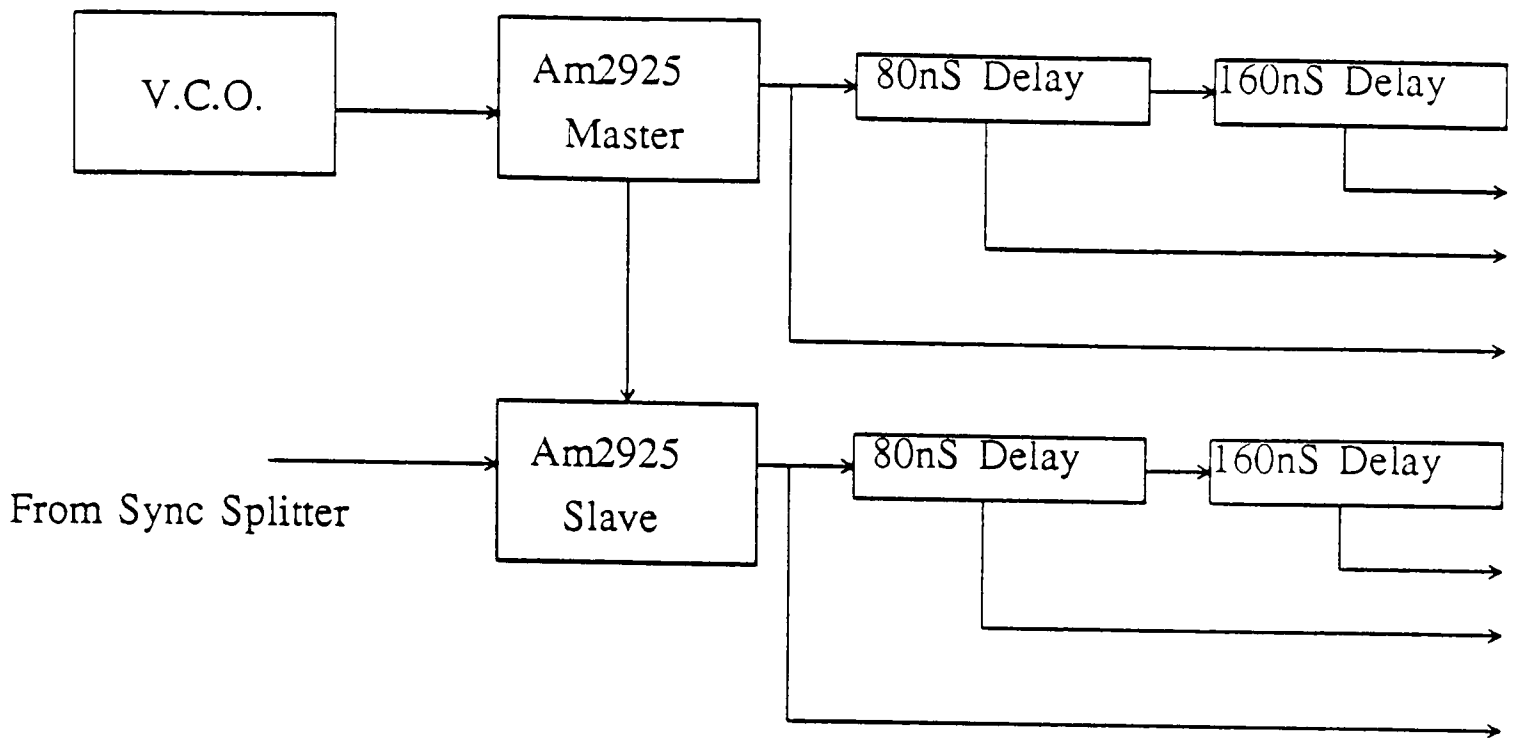


Figure 7.4: System Clock Block Diagram–Shows both system clocks. The master clock runs constantly to drive the planar 1 and 2 cards. The slave clock may be stopped and restarted by the attached camera, to genlock the analogue circuits to the camera.

map the output to either C1 or C3 signals. This is developed as follows (see also *Figure 7.5*).

Memory enable signals are controlled from the connected output of two buffers, with one of three sets of input lines being enabled to the output. The buffer outputs are the memory enable lines -CHIPA, -CHIPB and -CHIPC and the selected mapping is controlled by pipeline bits -I/O and R/W. If -I/O is high, then the chips are mapped in phase with C2',dependent upon the -CSA,B,C signals. If -I/O is low, then for a read (R/W high) enable is via C3 clocks, and for a write (R/W low) enable is via C1 clocks.

In order to clock some of the DMA controllers on the Planar 2 card, some of the 80nS period phased clock signals are used. These clocks are necessary so that the DMA controllers can generate new memory addresses at the correct frequency to capture a 512×512 image from a camera. The clock timing for the Am2940

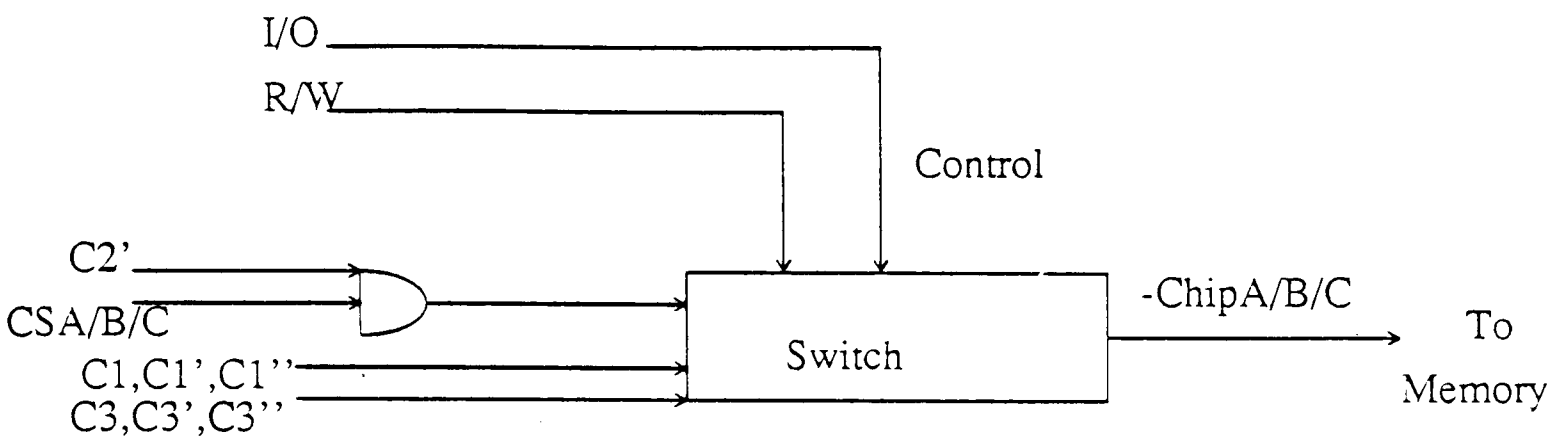


Figure 7.5: Memory Enables – Provides the enables for the three memory banks, the enables being dependent upon pipeline enable bits, and on whether the operation is an I/O one or not.

I/O DMA controllers is also dependent upon whether a read or write operation is required.This is again selected by an SN74S241 buffer controlled by the R/W line. If R/W is high (a read) then the A,B,C chips are clocked on C3',C3" and C3 respectively, and if R/W is low for a write the chips are clocked on CLK2,CLK2D and CLK2DD respectively.

In similar fashion, the A/D and D/A converters need to be clocked on an 80nS period. The clock signal for the D/A converter is provided by or'ing clock lines CLK1,CLK1D and CLK1DD (to provide an 80ns clock). That for the A/D converter is the same as the above only delayed by 40ns via a 74F164 register.

7.2.2 PC Interface

The system is designed so as to interface with the IBM PC through one of the interface slots.It is configured so as to appear to the PC as a prototype card (see IBM),ie it is mapped to Hex addresses 300-31F in the I/O map. The system was designed in this way for ease of programming, and so that no unecessary loads are placed on the PC's I/O channel. Address and control word decoding is done as on

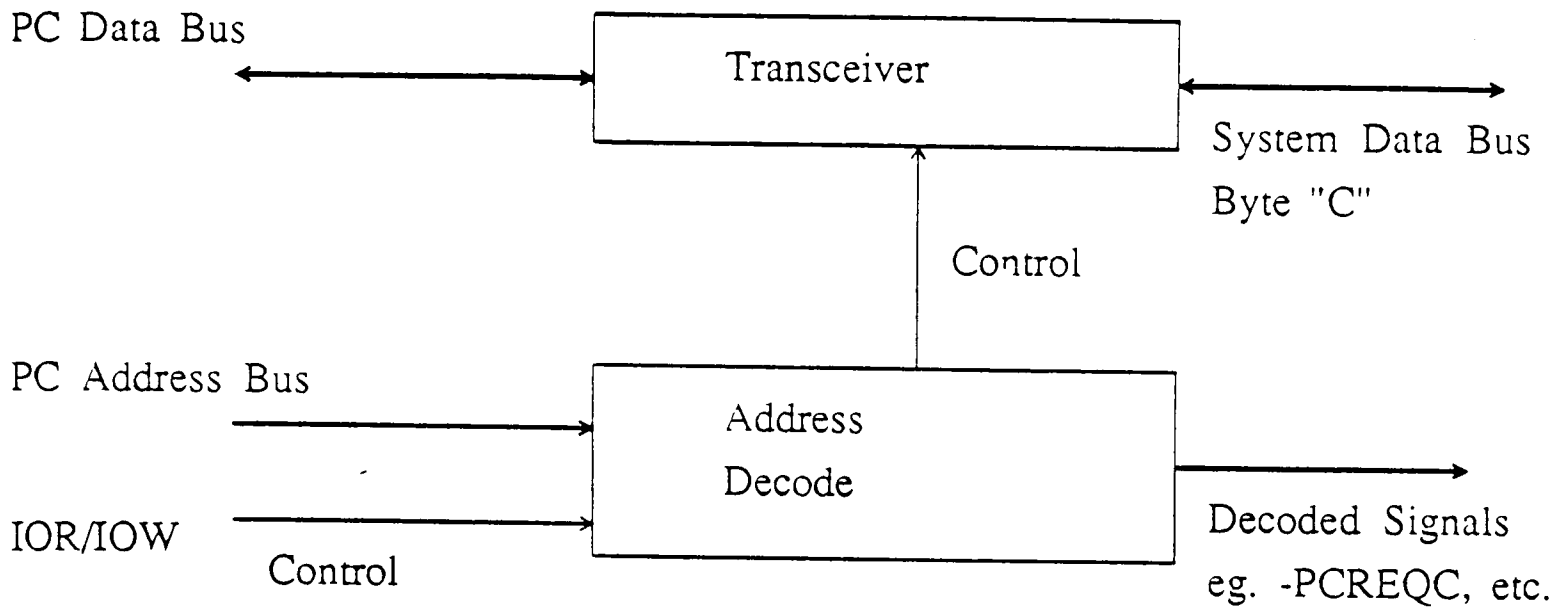


Figure 7.6: PC Interface – Provides an interface to a PC-XT, configured as an IBM prototype card. This allows system command from the PC, and provides the basis for image transfer between the PC and system.

the prototype card via TTL gates and two SN74S244 buffers. See *Figure 7.6* for a diagram of this part of the circuit. To ensure that the PC and system data buses are usually separate, data passes through an SN74S245 transceiver, which is only enabled when a read or write to the correct address is detected. PC data bits (0-7) are mapped to system bits 0-7 on byte c of the 24 bit system data bus. Additional decoding is performed on address bits 0,1,2 to decode the function required. Five system control signals are decoded from the combinations of the three low order address bits. The mapping is as in table 7.4

Address		Bits			Signal
Hex	Dec	+Addr0	+Addr1	+Addr2	Decoded[2]
300	768	0	0	0	-PCREQC
301	769	1	0	0	-PCREQD
302	770	0	1	0	-PCREQA
303	771	1	1	0	-PCREQ0
304	772	0	0	1	-PCACK

Table 7.4: PC Interface - Shows Address Decoding

The 5 control signals are used as follows.

[2]These signals are active low

-PCREQA signals transfer of an **A**ddress from the PC to the system.This is used to tell the system a starting address to locate a coefficient set or other data. It is used to control the enable (in one direction) on the Am2952A I/O port,and is also connected to input D1 on MUX B.

-PCREQC is used by the PC to signal that it wishes to send a **C**ontrol instruction to the system. This signal controls the input enable to the Am2920 instruction register,and also sends an interrupt to input P0 on the Am2914 interrupt controller. Data is thus clocked into the Am2920 for reading by the system microcontroller.

-PCREQD controls **D**ata transmission from the PC to the system.It controls the chip enable (in one direction) on the Am2952A data I/O port, and is also connected to D0 on MUX B.-

-PCREQO is used to request **O**utput from the Am2952A data port to the PC and also is connected to input D2 on MUX B.

-PCACK is used by the PC to request **ACK**nowledgement of a data transfer or operation, and controls the output enable on the Am2952A I/O port in the direction from system pipeline bits BR6-11 to the PC.

7.3 Analogue Interface

7.3.1 Data Interface

Part of the design specification of the proposed system was that it should be able to take in data from an interlaced,according to RS-170 standard CCD or vidicon camera and output data from the frame stores to a TV monitor. There is, therefore, the necessary interface circuitry on the PC card (see *Figure 7.7*). Also on the card is a circuit to phase lock the camera and the memory, when inputting an image, as well as circuitry to generate video sync signals for the output video if no camera is attached.

Data is input to the system from the composite video signal from a RS-170 CCIR standard camera. Data may be output to a monitor, again using the same interface standard, on a 625 line system. To ensure that the A/D converter has an accurate reference for the image black level, data from the analogue input to the system is buffered via an emitter–follower circuit and is then a.c coupled into the hold capacitor input of an LF398 sample–and–hold amplifier. Wired in this way, with the normal signal input grounded, the output will follow the hold capacitor input whilst the control line (pin 8) is low and will clamp to zero volts when it is high. The output of this part of the circuit is thus the analogue video signal with image black at 0V.

To provide a reference for the video signal, the +SET BLACK LEVEL signal occurs during two black, non picture lines, at the start of each field. The output from the LF398 is thus the analog video signal with its black level referenced to zero volts. To provide frequency stability to the circuit, and to satisfy the input drive requirement of the A/D converter, this d.c restored signal is then the input to a bipolar wideband operational amplifier followed by an NPN transistor buffer. The buffer has a gain of minus two, increasing the 1V p-p video input signal to the recommended 2 V p-p input for the A/D converter.

In order to digitise an analog 512×512 camera signal at video rates, an analogue to digital converter was needed which operates at greater than 14Mhz. The analogue to digital converter used is a TRW TDC1048 device, which is an 8 bit flash converter capable of operating at up to 20 megasamples/second. To provide stability to the circuit, all five V_{in} pins are connected together close to the device package, with the buffer amplifier feedback loop being closed at that point. The bottom reference voltage to the A/D converter V_{rb} is supplied by an inverting amplifier, buffered with a PNP transistor. The transistor sinks the current flowing through the reference resistor chain, and provides a low impedance source. The bottom reference voltage can be adjusted to cancel the gain error introduced by the offset voltage E_{ob}. The

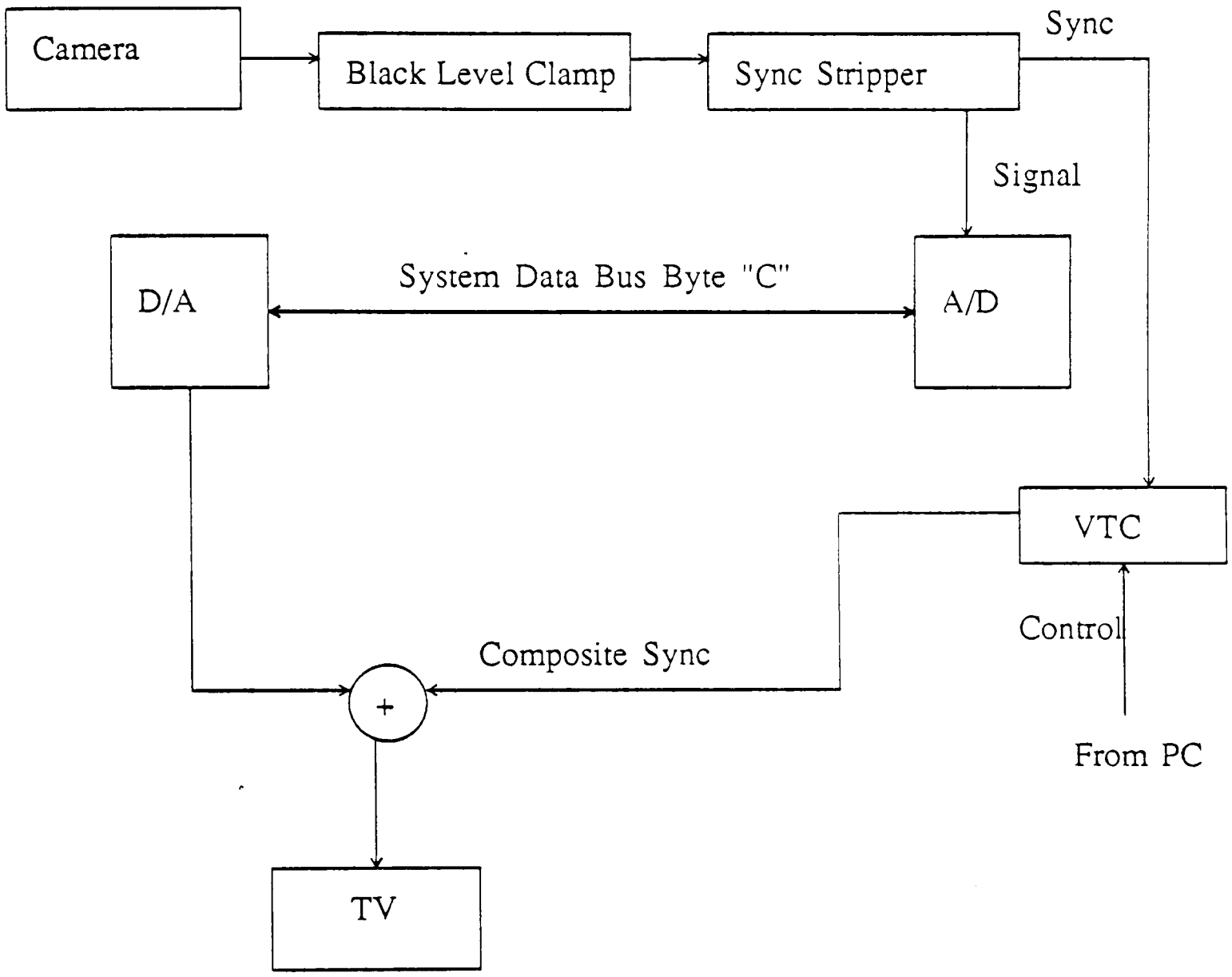


Figure 7.7: Analogue Interface – Provides analogue interface to the system from a camera, and from the system to a TV monitor. Also has the ability to regenerate composite sync signals to drive a monitor when no camera is attached.

bottom reference voltage (usually set to $\approx -2V$) may be adjusted via R15 on the PC card, and the input offset voltage may be adjusted via R14.

The output from the A/D converter is connected to data bus byte c. The data output is routed via a 74S241 buffer, to disable it from the bus if required. The enable is controlled by pipeline bit PL47. The converter is clocked by the A/D clock line. Separate analogue and digital ground paths are provided to keep noise to a minimum.

Similar performance was required from the D/A converter as from the A/D converter. Analogue output is managed via a TRW TDC1016 D/A converter, which is a 10 bit converter (only 8 bits are used) capable of operation at 20 megasamples/second. This part of the circuit must also deal with the recombination of the analogue video signal and the necessary synchronisation pulses. Eight bit data input to the D/A converter is taken from data bus byte c, and the converter is clocked on the D/A clock line (see timing circuits). The timing signals COMPOSITE SYNC and BLANKING are added by injecting current from the Wilson current source into a resistor divider circuit at the output of the TDC1016. This acts as a current mirror, where the current flowing from the emitter of Q6 is mirrored from the emitter of Q5. Balance of this part of the circuit is controlled by R36 and R38.

The D/A output and currents from the SYNC and BLANKING inputs are summed and amplified by the HA2539 wideband operational amplifier. To provide the correct output level, this has a inverting output with a gain slightly larger than 2. The output of the circuit is a composite video signal, complete with the necessary synchronisation and equalisation pulses. The reference for the TDC1016 is generated by dividing the output voltage from a two-terminal band-gap voltage reference. System gain is calibrated by adjusting variable resistor R1. The offset of the output voltage is set via resistor R26. The output is terminated with a 75Ω resistor.

7.3.2 Internal Synchronisation Signal Generation

In order to allow the system to output images to a monitor when the camera is not connected, the system must be capable of generating the composite synchronisation signals required by the monitor. If a camera is connected, the monitor's composite sync signals must be locked to the camera sync signals.

The necessary synchronisation signals are provided from an AMD Am8158 Video Timing Controller. This is a programmable timing controller which generates all of the necessary horizontal and vertical synchronisation signals. It may also be phase locked to an incoming video signal. The programmability of the device allows it to generate sync signals for differing image sizes, frame rates etc.

To allow output devices to be synchronised to input devices, the Am8158 may be locked to the sync signals from an attached camera. If a camera is attached, the incoming video signal to the board is routed to a sync-stripper circuit. The basis of this circuit is a comparator using two transistors. This comparator produces a 0V to 12V sync signal, with the horizontal and vertical syncs at slightly different levels. By using a LM393 comparator on the output of this circuit a TTL vertical sync signal can be derived from the incoming camera signal. This signal is fed to the EVSYNC input of the Am8158, and locks the output of this device to a camera when attached. If no camera is attached, the VTC free runs to provide output sync signals. This vertical sync signal is also used to drive a monostable to provide the correct pulse length for the SET BLACK LEVEL signal, required to DC clamp the incoming video signal.

The Video Timing Controller operates in the following manner, and is clocked from a 74LS321 oscillator at 3.03MHz. Before becoming operational, a number of internal registers must first be programmed from the PC. These control the length and proportions of the output synchronisation signals (see Am8158). The device is programmed by first inputting the address of the control register to be programmed, and then sending the actual data. Decoding of the PC address bus is done so that

the control register on the Am8158 is loaded by writing to address 308Hex, and the data register is loaded by writing to address 30AHex.

Thus to load register 1 with the value five, one would output from the PC a 1 to address 308, followed by a 5 to address 30A. The necessary register values for a 512×512 image at 25Hz are given in table 7.5.

Address	Value	Instruction
30A	10	Disable Display
30A	10	Select CCKR
308	6	Load 6
30A	11	Select HSRE
308	3D	Load 3D
30A	12	Select HSFE
308	5	Load 5
30A	13	Select HBRE
308	3C	Load 3C
30A	14	Select HBFE
308	9	Load 9
30A	15	Select VSREL
308	34	Load 34
30A	16	Select VSREH
308	1	Load 1
30A	17	Select VSFE
308	6	Load 6
30A	18	Select VBREL
308	2E	Load 2E
30A	19	Select VBREH
308	1	Load 1
30A	20	Select VBFE
308	19	Load 19
30A	21	RESET
30A	0B	RESET-ENABLE

Table 7.5: Values to be loaded into the Video Timing Controller

Once programmed, the VTC generates horizontal and vertical sync pulses which are exclusive OR'ed together to provide the composite sync output. The HSYNC output is inverted and is used as the WAITREQ control signal for the slave timing generator. The READY line for the slave timer is taken from the EBLANK output

on the Am8158,as is the video blanking signal.

This was the first part of the system to be constructed (see later section). Most of the major components used were based on the Am2900 series, e.g.

- Am2925 Timing Controller
- Am8158 Video Timing Controller

The main test of the correct choice of architecture for the system would come in the next board to be debugged, the Planar 1 card, which we will go on to describe in the next section. In this part of the design, however, the choice of just two main components, together with the necessary registers etc, has provided all of the necessary timing and video interface signals.

7.4 Planar 1 Card

7.4.1 General

The system board consists of two main sections: the program controller and the convolver. As already mentioned, in accord with the overall design approach, AMD's Am2900 family was chosen for most of these functions. Because a 3×3 convolution is being performed, the system is based around a 3 byte wide data bus to allow parallel data transfer. The data bus bytes are labelled a,b,c for convenience.Byte c is always used whenever single byte transfer is required. For example, byte c of the data bus is linked to the PC card, and is used to transfer data to/from the analogue side of the circuit.

7.4.2 Program Controller

The design of the system controller was to be pipelined to increase throughput. As already mentioned, I had decided to opt for a microprogram sequencer, rather than a microprocessor, running from a microprogram memory. The choice of a microprogram memory, which, if desired, could be easily changed, to give the machine a new

instruction set at a future date,would be a useful feature. The controller should be capable of conditional branching, and also of unconditional branches, so that the PC could signal a desired program segment to run.

Program sequencing is under control of an AMD Am2910A microcontroller (see AMD, and Mick 1980). This was chosen from the Am2900 family as a device which offered the required speed and addressing capability. The Am2910A can address 4K words of microprogram memory via twelve address lines Y0-11. It is programmed via 4 instruction bits I0-3. The output from the Am2910A addresses 10 Am27S191A-35 PROMS,which store the microprogram. These are fast PROMS, with an access time of 35nS. The output of this memory is fed into a pipeline register, composed of 10 Am2918 registers. The tri-state enable of 12 of the register output bits (BR0-11) is controlled by the -PL output on the Am2910A. Bits BR0-11 are fed back to the input of the Am2910A to provide branch addresses. The microcontroller is clocked from one of the main clock signals at a minimum clock period of 240ns.

As the controller would be required to execute program branches, there was a hard limit to the extent to which it could be pipelined. Thus it was decided to use an instruction prefetch for the controller, but not to pipeline more extensively. As the system executes the present instruction on the output of the pipeline register,the Am2910A fetches the next instruction. The presence of the pipeline register renders the machine instruction-data based. An instruction-data based architecture is illustrated in *Figure 7.8* The fact that the machine is only pipelined by one instruction means that there is only a small loss in speed when a conditional branch is executed, as is discussed in the next Chapter with reference to the analysis in Chapter 5.

It was also required that the machine should be able to branch on the result of a test, e.g. the end of a row had been reached . The Am2910A has a condition test input (-CC),which can indicate that a test is passed.This allows the Am2910A to execute conditional jumps.The condition test input is fed from the output of two

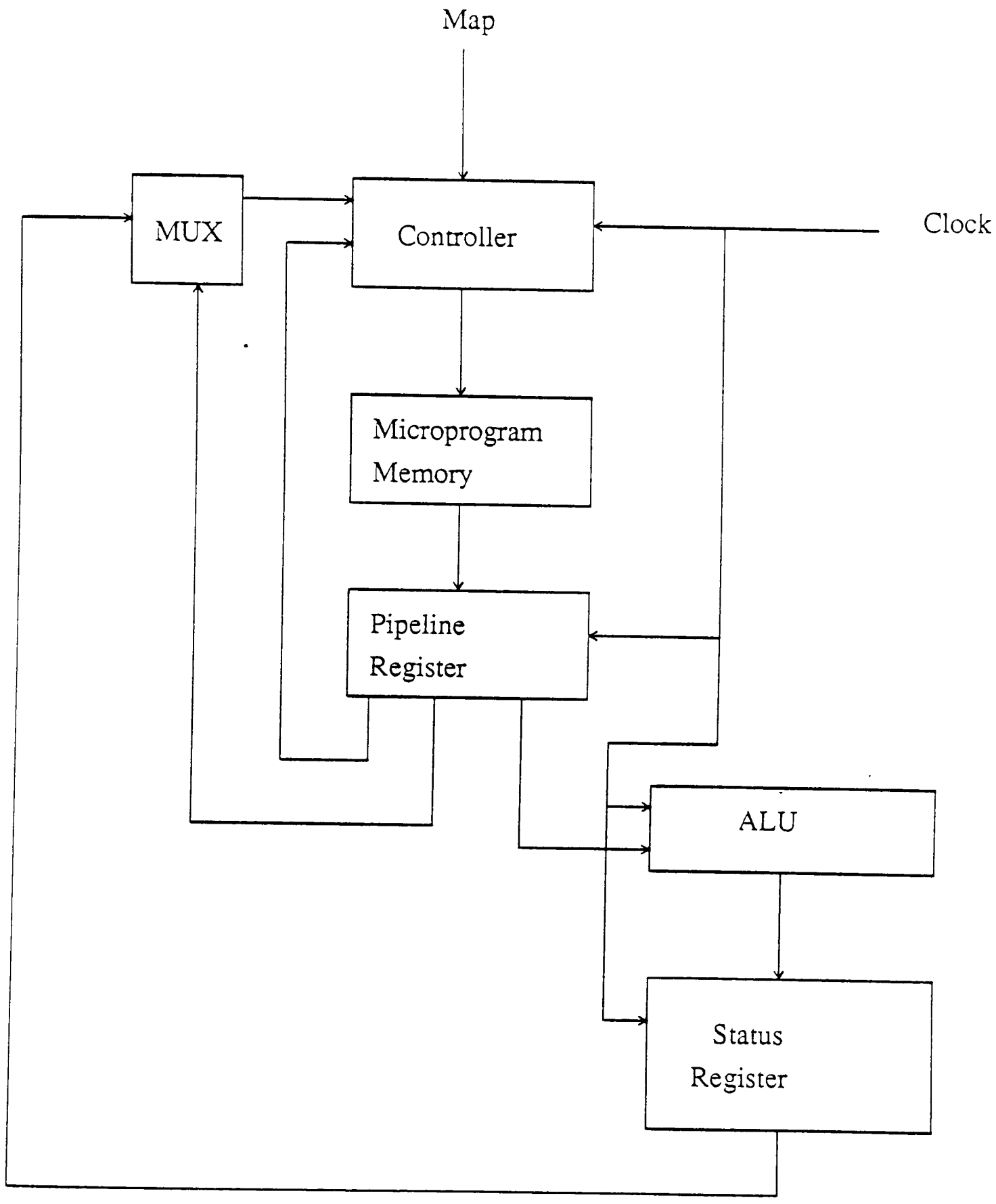


Figure 7.8: An Instruction-data based architecture. This type of architecture is the basis of the system.

Am2922 condition code multiplexers, each having 8 inputs. All signals which should be able to affect the choice of a branch are fed to one of the Am2922 inputs. The system controller is shown in *Figure 7.9.*

To give the controller the ability to respond to asynchronous interrupts without polling, an extra device was required. This would, for instance, allow the PC to interrupt the system whilst in the middle of executing a program. The machine has a direct interrupt capability via the Am2914 priority interrupt chip. This has 8 levels of interrupt. In this case only the lowest level is used,and is driven on the signal -PCREQC(ommand). This signal is generated by an OUT command from the PC to a specified port (see PC Interface). On receiving an interrupt,the Am2914 signals an interrupt and new branch address to the Am2910A controller. The Am2914 is programmable if desired to allow masking of interrupts in future developments of the system.

As mentioned, the controller had to respond to branching statements from the controlling PC. This would allow the PC to select a number of microprogrammed subroutines for the system to execute. This would be equivalent to a higher level language for the system. To allow the sequencer to perform this sort of branch, the Am2910A can take a branch address from the 3 Am27S21A map PROMs, functioning as look up tables (LUT's), which it can enable via the -MAP output. The address to these map PROMS is taken from the output of an Am2920 instruction register,which is connected to the c byte of the data bus.The enable input to the Am2920 instruction register is controlled by the -PCREQC line.The output enable for the register is controlled by pipeline bit PL27. Thus when the register is loaded from the PC via line -PCREQC, this also signals an interrupt to the Am2910A controller, to signal the presence of an instruction. The mapping proms are then used as a look up table for the eight bit PC instruction, and are decoded to provide an unconditional branch address for the Am2910A.

The system may be reset either via a push switch on the PC card, or automat-

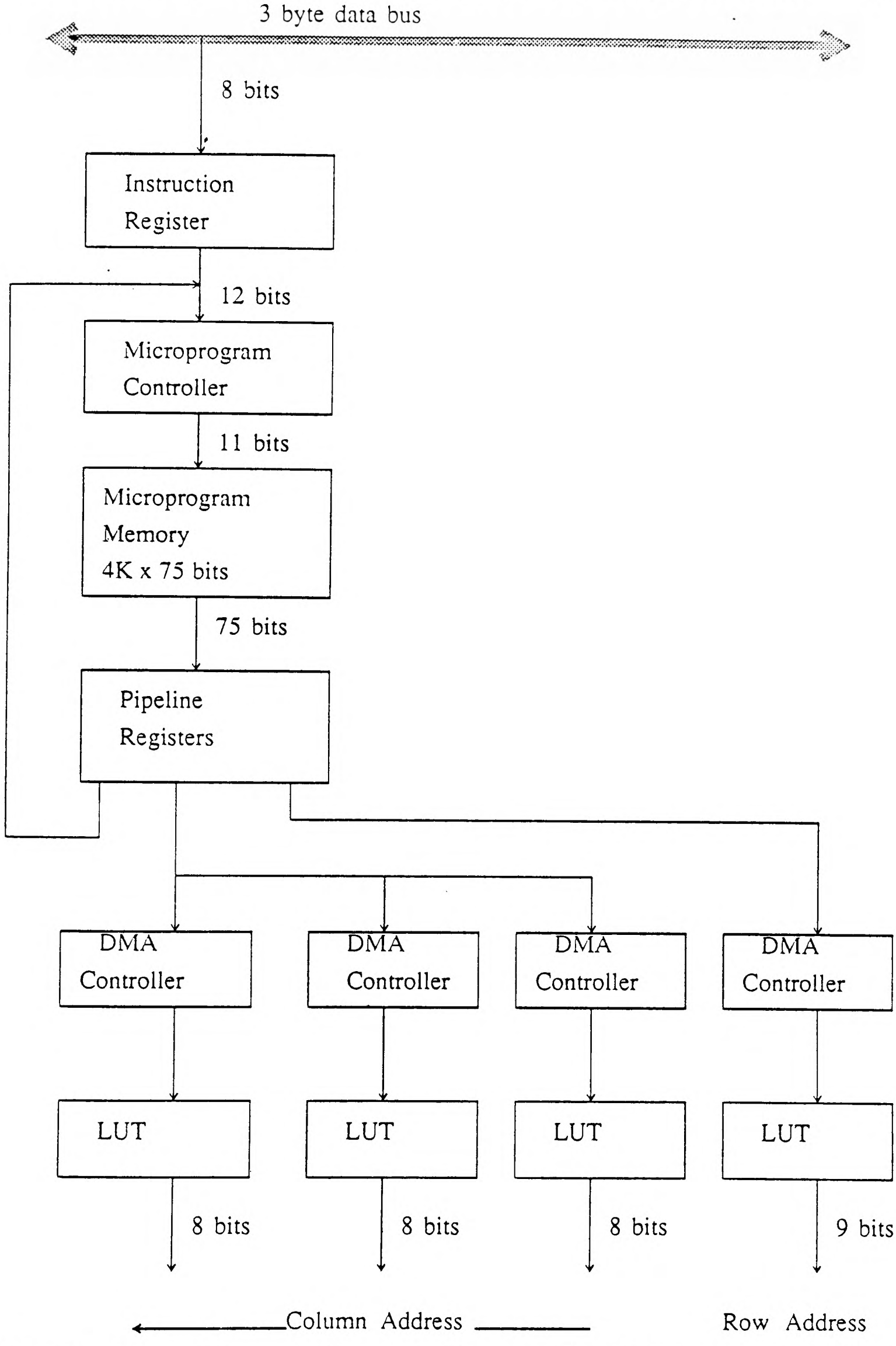


Figure 7.9: System Microcontroller – Provides control for all of the parallel-pipelined convolver. Takes branch commands from PC and from state parameters within the system, to control a microprogram memory.

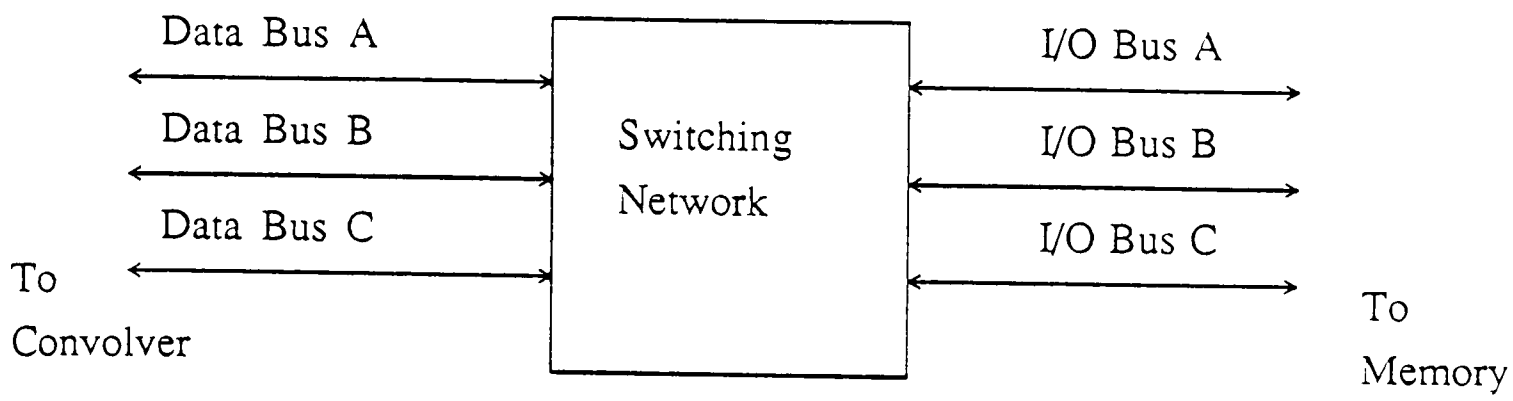


Figure 7.10: Memory Multiplexers– This controls the mapping between the three memory data I/O lines and the three convolver data bus lines, to input data in column order into the convolver registers

ically when the PC is rebooted.

7.4.3 Memory Mapping

As it was necessary to map data from the frame stores in column order into the convolver, and so that captured image pixels could be stored in the correct frame store bank, a multiplexer is used between the memory bank I/O lines and the system data bus. This takes the form of a switching network which takes data to or from the two frame stores on the Planar 2 card, and multiplexes it onto the correct bus lines for either the convolver, or the analogue I/O circuitry. This is shown in *Figure 7.10.*

The three 8 bit data input/output lines to the frame stores I/O(0-7)A,B,C are connected from Planar 2 to Planar 1 via cable and are then connected to 9 Am2947 bidirectional buffers, each line being connected to the A inputs of 3 Am2947's.The B lines from the Am2947's are connected in threes, so that by enabling suitable trios of Am2947s it is possible to map lines I/O(0-7)A,B,C to data bus bytes a,b,c on the Planar 1 card in any combination. This combination is controlled by the

three outputs of an SN74S241 buffer. If pipeline bit -I/O is high, implying that the operation is not an input/output one, then lines -OE1-3 control the data mapping such that the mapping is as shown in table 7.6

Line	maps	to
-OE1	I/OA	bus c
	I/OB	bus a
	I/OC	bus b
-OE2	I/OA	bus b
	I/OB	bus c
	I/OC	bus a
-OE3	I/OA	bus a
	I/OB	bus b
	I/OC	bus c

Table 7.6: Memory Mapping, showing the decoding pipeline bits

When -I/O is low, putting the machine in input/output mode, the three clock lines control the data mapping. The lines OE1/2/3 which are controlled from the microprogram, are simulated by some of the system clock signals. The correspondence is shown in table 7.7

Equivalent to	
Map Line	Clock
OE1	C2'
OE2	C2"
OE3	C2

Table 7.7: Memory Mapping in I/O Mode

The direction of data transmission is controlled by pipeline bit PL62 (R/W).When this is low,data transmission is in the direction from the data bus to the I/O(0-7) lines,ie in the direction of a write to the frame stores. When high, the system operates in the other direction, to read data out from the frame stores.

7.4.4 Convolver

The convolver board performs the function of the arithmetic and logic unit (ALU) in the system.Its basic function is to take a set of 9 data values $d_1 - d_9$, with 9 predetermined but programmable coefficient values $c_1 - c_9$, and provide the sum of the products

$$I_{out}(x,y) = \left(\sum_{i=1}^{i=9} d_i c_i\right) + \lambda \tag{7.1}$$

The value of λ may be programmed by the user. Typical values for C_i might be $C_i = 1,2,1,0,0,0,-1,-2,-1$ for a Sobel filter, or $C_i = 1/12$, $1/6$, $1/12$, $1/6$, 0, $1/6$, $1/12$, $1/6$, $1/12$ in Horn and Schunck's optic flow algorithm. The 9 data values are taken from a 3×3 window around a specified point on the image,and the sum of products then replaces the center value of the window. We have already seen many uses of this type of processing, from simple edge detectors, such as a Sobel filter, to some of the more advanced examples shown in the chapter on vision algorithms.

To add to its functional capability, the board also has the ability to add the contents of a further "add" register to this sum,and output this value,or to output either the maximum or minimum of the sum of products and the add register,or to output the binary value of the sum of products thresholded with the contents of the add register. The functions performed by this part of the system are shown in *Figure 7.11.* The output and input functions are controlled by bits OSEL1-2 and ISEL1-2 (PL56-7 and PL54-5 respectively) and are shown in table 7.8.

The input to the convolver is 3 bytes wide, so that loading the coefficients c_i and data d_i is accomplished in units of 3 bytes.When the add register or coefficient register is to be loaded,the data is presented on byte c of the data bus.The eight bit data output is also connected to byte c of the data bus.

Data storage on the convolver is provided by two sets of 9 Mullard 74F164 registers. Of these registers, one set stores data and the other stores coefficients. The registers are arranged as 3×3 arrays, with the first column in each array being connected to the input data bus bytes a/b/c, and the 2nd and 3rd columns chained

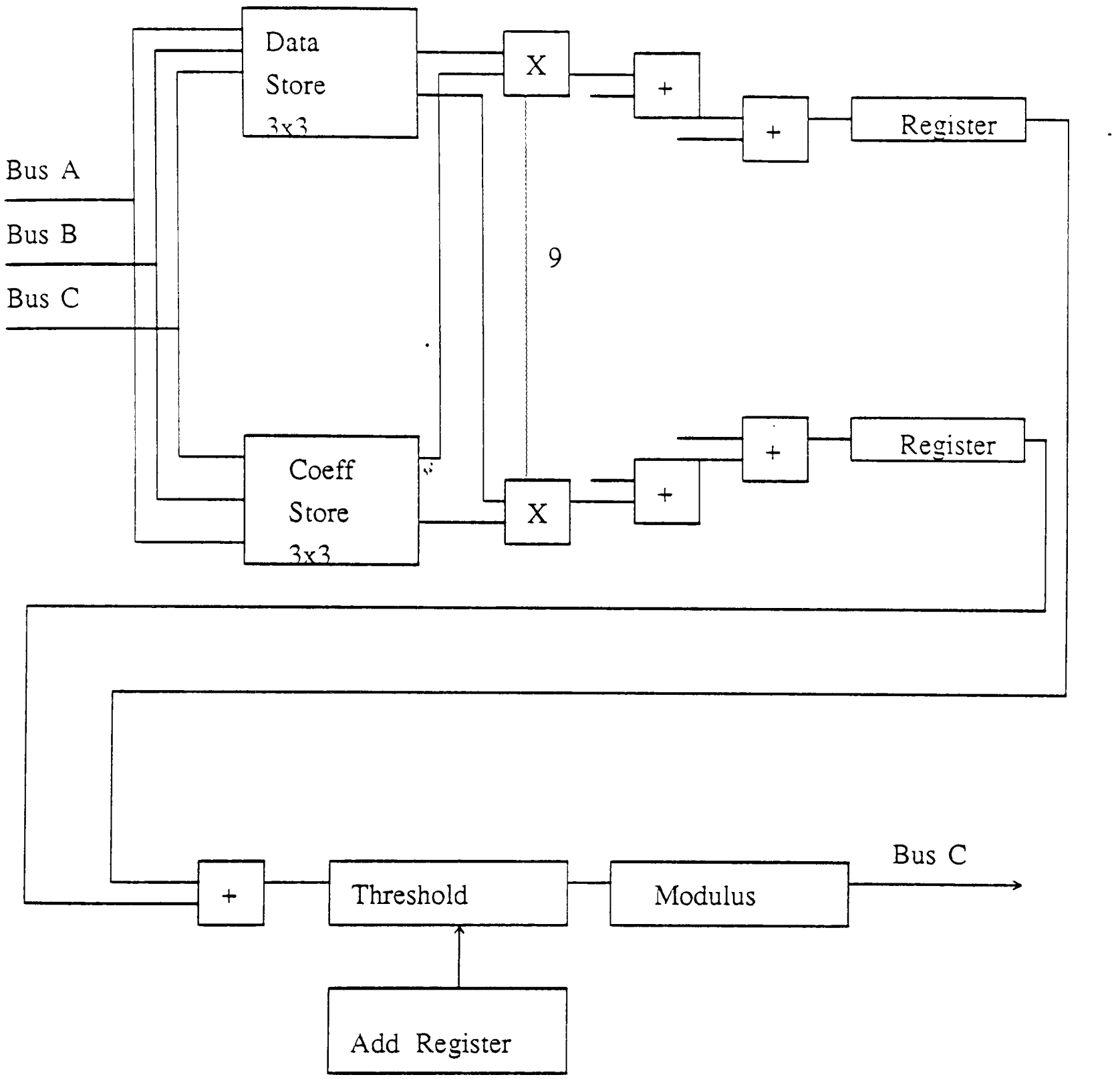


Figure 7.11: The Convolver – contains nine parallel multipliers and the necessary adders to perform $I_{out}(i,j) = \sum_{i=1}^{i=9} d_i c_i + \lambda$. The convolver is loaded in parallel and is internally pipelined.

OUTPUTS		
OSEL1	OSEL2	OUTPUT
0	0	maximum of sum of prods and add reg.
1	0	minimum of sum of prods and add reg.
1	1	sum of sum of prods and add reg.
INPUTS		
ISEL1	ISEL2	INPUT
0	0	coefficient register
1	0	add register
0	1	data register
1	1	coefficient center pixel.

Table 7.8: Convolver Functions and Decoding

to the 1st and 2nd respectively.

To allow an arbitrary offset to be added to the convolution product ,there is also an additional register,denoted the ADD register,which is loaded from byte c of the input data bus. To allow the user to load a single central coefficient, so that a simple scaling of the image is performed, the center coefficient register, as well as being connected to the previous coefficient register, is also connected to input byte C to allow direct loading.

The registers are clocked on the low to high clock transition,and have an enabled output when the clock signal remains high. A combination of TTL gates ensures that all registers usually have an enabled output,with an Am2924 demultiplexer decoding the ISEL1-2 inputs to clock a particular register set,when the input clock enable CLOCK CONVI (PL59) is low. The low to high clock transition is delayed for the second and third sets of registers to allow data to ripple through.

Due to the number of addition stages required to add the nine multiplication products together, it was found that to accomplish the necessary multiplications and additions within the system clock cycle of 240nS would have been impossible even using the fast components chosen for the task. I thus decided to pipeline the convolver as well. As we saw in Chapter 5, as long as the pipeline is kept full, a speedup equal to the number of pipeline stages will be obtained. A two stage pipeline is used, which doubles the throughput of this part of the circuit, to bring

it within the specified cycle time.

Data from corresponding pairs of data and coefficients registers is input to one of 9 Am25S558 multipliers. These are set up via control inputs Ru,Rs,Xm,Ym to accept data in unsigned format,coefficients as two's complement signed numbers,and to produce a rounded 8 bit two's complement output.The output of pairs of multipliers is then used as input to sets of 3 SN74S283 4 bit fast adders. The low order 4 bits from the multipliers are the inputs to the low order '283,the high order 4 bits to the middle '283,with the inputs to the high order '283s being the padded sign bit. Additions of the multiplication products are then done as 12 bit additions,to avoid overflow problems. There are three sets of additions, which are done asynchronously within a 240ns system clock cycle. On the next low to high transition of the CLOCK CONVI clock,the partial results from these additions (two 12 bit numbers) and the contents of the add register are clocked through 3 more 74F164 registers. This means that the convolver board is itself pipelined, with a latency of one system clock cycle.

Finally, adding λ to $\sum_i c_i d_i$ is done from the outputs of the second stage registers, and the contents of the add register is then added into this sum. The high order 8 bits are then selected from the sum of products,and its sum with the add register. At this stage, if the output selected is the convolution result with the addition of the add register contents, this result is clocked into the output register. When the maximum or minimum of the convolution result and the add register contents is requested, the sum of products and the add register are then compared using two SN74S85 magnitude comparators and, depending upon the outcome of this comparison and the selected output mode,the correct sum is enabled out of one of two further 74F164 registers.

To allow for the use of negative coefficients in the convolver, which can obviously produce a negative convolution result it was necessary to have the ability to convert negative numbers back to the range 0-255. Depending upon the MOD input signal

(PL58), 128 is added to the result to convert back to an unsigned number.If MOD is high,the output is unsigned,and vice versa.The result of thresholding the sum of products with the contents of the add register are also enabled out to bit 7 of data bus byte b. The output from the convolver is then enabled back via the mapping network to one of the two frame stores.

7.5 Planar 2 Card

7.5.1 General

The second planar card contains both of the system frame stores, and the address generation circuitry for the memory. We first discuss address generation.

7.5.2 Memory Address Generator

The memory address is segmented into a row and a column address. As such there are two address generators,one for each part of the address.In addition there is a third address generator,which is timed via phased clocks to allow the very fast addressing necessary to input a 512x512 image at video rates. It was required that the address generators should be programmable, so that differing lengths of address sequence could be generated. For this reason, and also for compatability with the sequencer, all three address generators are based on AMD Am2940 DMA controller chips (see AMD).

It was intended that the microcontroller should be able to start the DMA controllers in operation and then leave the system alone until such stage as the required address sequence had been achieved. The DMA controllers can be loaded with a starting address and a word count and will then operate independently until the required number of words have been addressed. This design is illustrated in *Figure 7.12.*

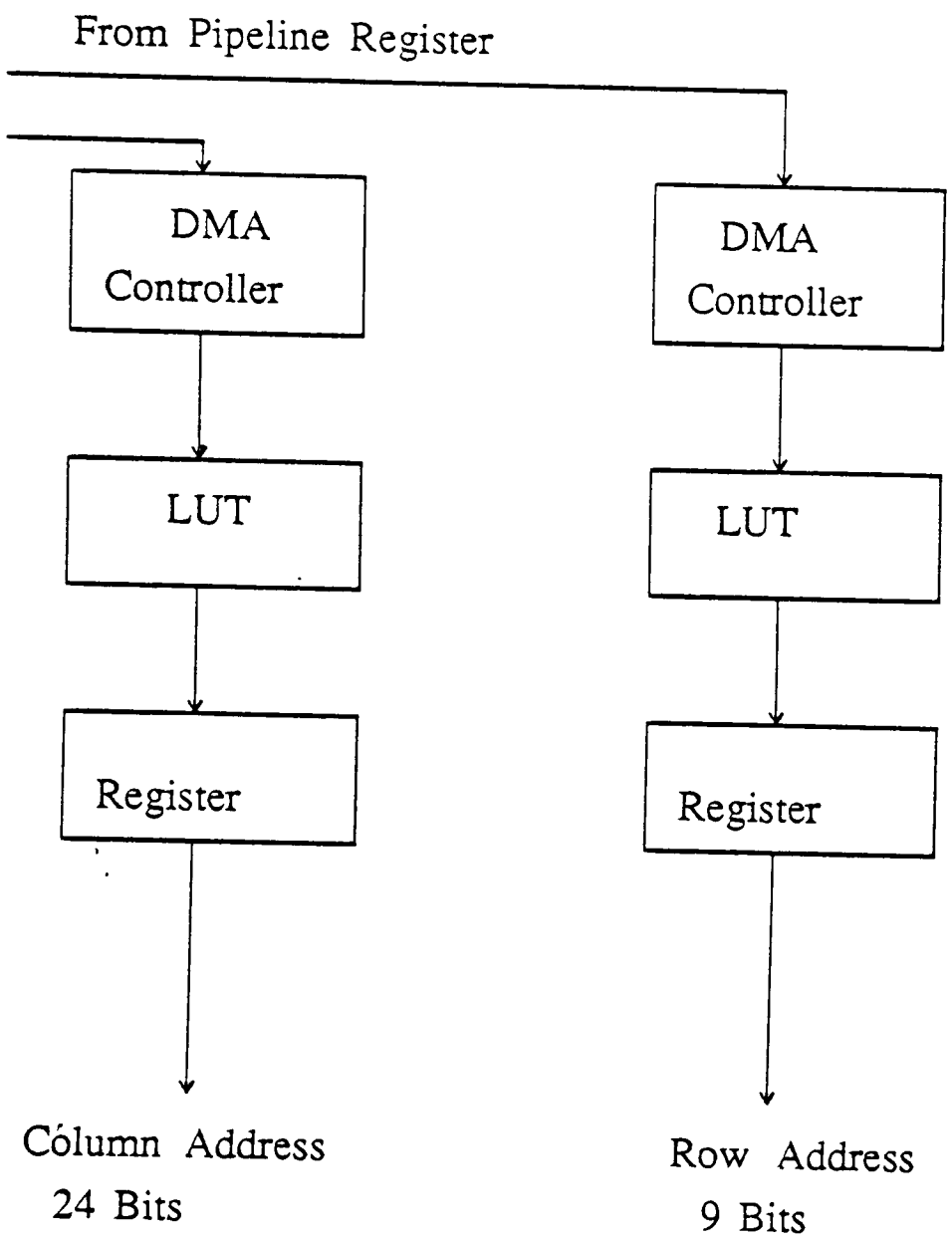


Figure 7.12: Address Generation – shows row and column address generation for the frame stores, using programmable DMA controllers, and address look up tables

Row Address Generation

The row address generator is based around two Am2940 DMA chips,each of which can generate 8 address bits.However,only 12 bits are required by the address Look Up Table (LUT), attached to the outputs of the controllers. This look up table is used so that complex, non-consecutive address sequences can be generated. Thus the DMA controllers simply generate a series of consecutive addresses, which are then decoded on one of two banks of 3 Am27S41A-35 PROMS.A bit of the high order Am2940 is used to select which one of the banks is enabled.

The Am2940's are fully programmable via three instruction inputs, provided from the microprogram memory. Details such as the addressing mode, word counts and starting count may be loaded. The Am2940's take initial data from a pair of Am27S29A - 35ns access PROMS,which are addressed either directly from the Am2920 instruction register,or via a SN74LS244 buffer from pipeline bits BR6-11. Instruction input to the two Am2940's is provided via pipeline bits PL34-6. The chips have an output enable controlled by pipeline bit PL37 which is active low. Pipeline bit PL33 (-CNT) is connected to the -ACI and -WCI inputs of the Am2940 and enables address counting when low. The two chips are ganged together with full look ahead carry generation.

It is sometimes necessary to address a row that is not the one containing the current center pixel (ie when writing back a convolved image onto the original). For this reason the row generator has a means of outputting previously generated addresses. When the Am2940's are enabled, one of the two banks of mapping Proms will be active. The addresses that they provide are 9 bits long and are taken as the input to a bank of 15 AM2918 registers,arranged as a pipelined bank of five 12 bit registers. The registers are arranged as a 5×3 array, with the input to the top of the array, and the successive rows are then ganged together. Data is clocked along down the array.

One bank at a time may be enabled to the frame store row address inputs,via the

output of an Am2924 demultiplexer,controlled by pipeline bits PL66-8.This allows either the current row address to be sent,or one generated up to four clock cycles previously.

Once programmed for an operation, the Am2940 indicates that the operation is complete by signalling a high on the DONE output. The outputs from both chips are or'd and are connected to input D0 on multiplexer A.This line goes high when the Am2940 has completed its word count on a sequential address operation, thus signalling to the controller.

The row address generator also provides the address for the coefficient store RAM,which is composed of two Am2148 -35 ns access RAMs, capable of storing 1024 bytes,or 113 sets of 9 coefficients.The address for this chip is taken from bits A0-7 of the low order Am2940 and bits A0-1 of the high order Am2940.Pipeline bit PL63 controls the chip enables, with bit PL62 (R/W) controlling read(high) or write (low). This will allow the PC to download special coefficient sets if required.

Column Address Generation

The column address generator is basically the same as the row address generator.It consists of two further Am2940 DMA chips,loaded from two Am27S29A - 35 PROMS. The address for these two PROMS is generated from pipeline bits BR0-5. The column address is generated in the same way as the row address, via look-up PROMS. The address is input to a five-deep bank of registers. Three output bytes from these registers provide the three eight-bit column addresses required for the frame stores.

The completion of a column address operation may be detected in one of two ways. The first is from the DONE outputs of the two Am2940's, which is connected to input D1 on MUX A. The generation of a correct address sequence may also be detected on the output of the two Am27S41A's which is connected to lines A0-7 on an Am25LS2521 8 bit magnitude comparator. Lines B0-7 are connected via a

dip-switch which allows choice of a finishing value. The output from this chip is connected to input D3 on MUX A.

I/O Address Generation

As already mentioned, it is necessary that the I/O address generators, which provide the column address in I/O mode, should be able to cycle with a shorter cycle time than the usual 240nS. Special clocking arrangements are employed in this case. The input/output address generator is arranged around three individual Am2940's. The different layout of these Am2940's is to allow an 80ns cycle time for the column address. This is achieved by phasing the 240ns main clock and running the three Am2940's on three different clock signals. The three Am2940's share common instruction lines on pipeline bits PL38-42. Each Am2940 is loaded from the same Am27S29A - 35 PROM ,which is addressed on pipeline bits BR0-5 (the same as the column address Am2940's).The output from the three Am2940's is used to address three individual Am27S29A - 35 ns access PROMS.The 8 bit outputs from these three mapping proms A,B,C are connected directly to column address lines A, B, and C.There is an active low enable on the three map proms, -I/O, which is pipeline bit PL70.The I/O DMA generators signal indicate that they have completed an operation by signalling the usual done signal.

7.5.3 Main Memory

Frame Store Mapping

The two frame stores as mentioned earlier are divided into three sections each, denoted A,B and C. Columns of the image are stored as follows in table 7.9

Section	Columns	Generalisation
A	$0, 3, 6, 9, \cdots 252, 255, \cdots 510$	$0 + 3n; n = 0, 1 \cdots 85 \cdots 170$
B	$1, 4, 7, 10, \cdots 253, 256, \cdots 511$	$1 + 3n; n = 0, 1 \cdots 84 \cdots 170$
C	$2, 5, 8, 11 \cdots 254, 257, \cdots 509$	$2 + 3n; n = 0, 1 \cdots 84 \cdots 169$

Table 7.9: Storage of Columns in Frame Stores

Memory is arranged in this fashion so that 3 adjacent columns can be read out of the memory in parallel and into the convolver.This arrangement also speeds memory access for interface with cameras and TV's so that adjacent columns can be accessed at high speed.

Frame Stores

There are two frame stores on the system,each capable of storing a 512×512 square image with 8 bits of grey level per pixel. The two stores are identical in all respects. Very fast access times were required of these frame stores, meaning RAMs with access times about 35nS, or more memory interleaving with slower devices. Each frame store consists of 36 Am99C164-35ns access static RAM chips,which are arranged in three blocks of 12 chips each.Each memory chip stores 16Kx4 bits,so they are arranged in pairs to store 8 bit values.Although the actual memory required for each frame store need only be 32 chips,there are 36 because of the column address decoding. The maximum column address can be 171 decimal, or 10101011 binary. The low 5 bits of this address are connected directly to the memory chips, whilst the high order 3 bits control the selection of individual chips within a particular block. This scheme is shown in *Figure 7.13*. These high order bits can occur in only the following 6 combinations if the maximum address is 171(D) ;

000
001
010
011
100
101

Because there are 6 combinations, there must be six sets of 2 chips in each block, making 36 chips in total. There is thus a small amount of redundant storage in the system.

All chips in each of the three sections share the same data bus line, giving three data bus lines into each store,a total of 24 bits of data. The lines are designated I/O(0-7)A, B and C, and are carried over from the switching network on the Planar

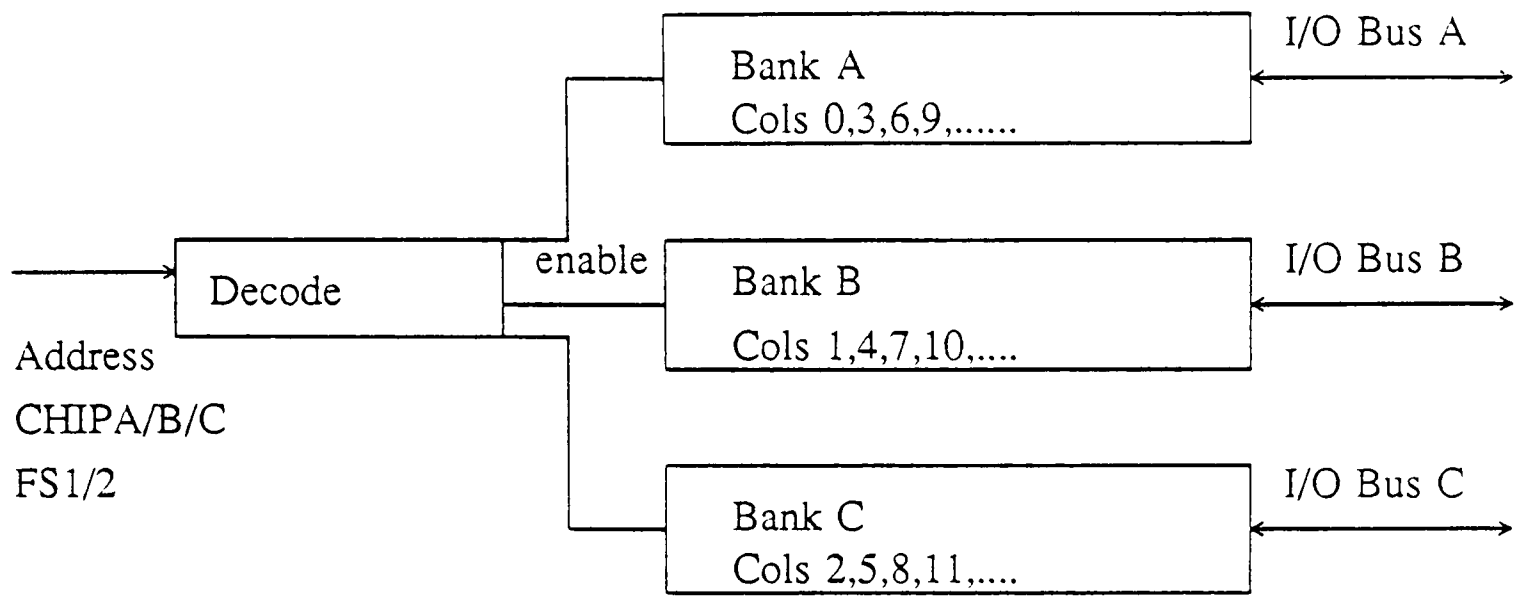


Figure 7.13: Memory Layout – Shows that both frame stores are layed out in three blocks, storing alternating columns. This speeds access to the memory.

1 card.

The 9 row address lines from the main board are connected to all 36 memory chips.They are connected to address lines A0-8 on the RAMs. There are three sets of column address lines (A,B,C) each of 8 bits. These are generated from the column or I/O map proms as already detailed. For each of the three sections of the memory,the relevant column address lines bits 3-7 are connected directly to the chip address lines A9-13.The low order three bits of the input address A0-2 are connected to the A,B,C inputs respectively of the appropriate one of three Am2924 demultiplexers.The first 6 outputs from each Am2924, bits Y0-5 are connected to the -E (enable) lines of the relevant pair of memory chips in each section.

The demultiplexers are controlled in the following manner, with each Am2924 having three enable lines. Line -G2B is connected to the appropriate -ChipA, -ChipB or -ChipC input line,to enable individual addressing of a single bank of memory. These inputs are generated on the PC card as already mentioned. On frame store one,line -G2A from all three 2924's is connected to the input signal FS1/2 (bit PL61),with input G1 being held high. On frame store two,input G1 is connected to FS1/2, with -G2A held low. This is the only difference between the

two frame stores,and means that frame store one is active when FS1/2 is low and vice versa.

7.6 System Operation

7.6.1 Input/Output to TV or Camera

Because of the need for fast memory addressing when inputting or outputting large images (up to the system maximum of 512×512) it is necessary to use the Am2940 I/O DMA controllers for the column control,together with the Am2940 Row controllers. Since the camera is interlaced, camera output is carried out on odd field followed by even field, and the system should start using row series 1, as the appropriate row address sequence in the row map prom (see Appendix A). The basic principle of operation is as follows.The row controller selects the first odd row of memory.The three Am2940 I/O controllers then sequence through columns in order,using the phased clock inputs to the Am2940's (80 ns delay between clocks). This is illustrated in *Figure 7.14*. It should be noted that although the system is designed to operate its I/O on video signals with a 2:1 interlaced format, it would be possible to change this to non-interlaced format by using row series 0 instead of row series 1 and 2 for the input/output.

All three of the I/O map proms contain the same values, so address 0 from Am2940A addresses column 0,address 0 from Am2940B gets column 1,address 0 from Am2940C gets column 2 and so on,until the end of the row,when a new row is selected.When all of the odd rows have been covered, row series 2 is selected, which inputs the even rows, and the complete process is repeated.The three Am2940's are driven from clock signals CLK2, CLK2D, CLK2DD (input) or C3',C3",C3 (output) respectively,which are phased by 80 ns. These clock signals are generated from the slave Am2925 timing controller.This allows this slave controller to be stopped at the end of a row or field,when the memory controllers either reset (columns) or increment (rows).

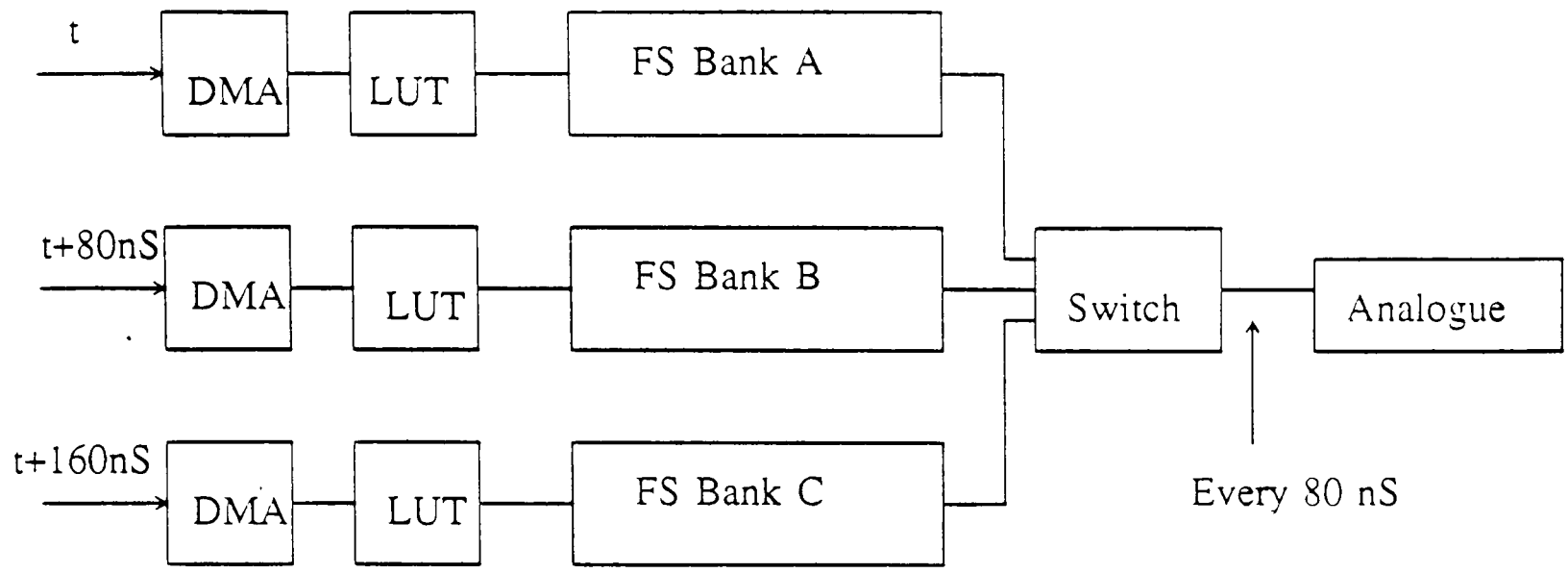


Figure 7.14: The Use of Phased address controllers allows high speed column addressing during interface to camera and monitor.

Stopping is controlled by either the horizontal or vertical camera syncs going low, which is input to the slave Am2925's -WAITREQ input. This causes a clock halt after the next high to low transition of the Cx input,which is connected to clock signal C1". The clock is restarted by the -READY line going low.This is driven from the EBLANK output of the video timing controller.

Although each memory section is addressed individually by the appropriate Am2940,they have to be enabled in the correct sequence onto bus line c,to connect them to the A/D converter. Enabling of the appropriate memory section is accomplished by driving one of the -CHIPA, -CHIPB or -CHIPC lines low.These are driven (when -I/O is low), by clock signals C1', C1" and C1 (input) or C3' , C3" and C3 (output) respectively. These occur 120 ns after the corresponding Am2940 clock transition,which allows the Am2940 to present its new address to the memory. As well as the chip enables, the data multiplexer is also clocked,to map the correct memory section to data bus line c and hence the A/D or D/A converters. When in I/O mode (ie -I/O is low), this mapping, which is usually controlled by OE1-3, is clocked by clock lines C2', C2" and C2 respectively. These lines are enabled 40ns before the memory is enabled, so that the correct data path is open for data transfer

when writing to the memory from the A/D converter (ie input). When outputting to the D/A from the memory, the relevant memory section is enabled 40ns before the correct mapping is set up.The D/A converter is clocked 40ns after this (to allow set up to the D/A) at the same time as the next memory section is being enabled (this doesn't affect the D/A as the output from this section of memory isn't yet mapped to bus byte c).

7.6.2 Input or Output to Controlling PC.

The speed of this data transfer is limited by the speed at which the PC can transfer data (slightly under 1MHz). Because of this, Am2940 row controllers and Am2940 column controllers are used.. The Am2940 row controller uses row series 0 for this operation.Memory sequencing is required to select consecutive rows,and to cycle along all of the columns in a row before moving onto the next row.To achieve this, the system is operated in convolve mode, ie -I/O is high.

The Am2940 column controllers clock the first three values from the column map prom into the first three sets of Am2918 registers, and the first row address is clocked into the row register. Line OE4 is held low, to enable addresses from the first three column registers to the three memory sections. Then -CHIPA is set low via pipeline bit -CSA, OE1 is set low to map memory section A onto bus byte c and data transfer takes place.Then -CHIPB and OE2 are set,and transfer of the second column takes place. Finally -CHIPC and OE3 allow transfer of the third column. Three more column values are then clocked into the shift registers,and the process repeats itself,until the entire row has been covered,when the Am2940 row controller clocks another row address (consecutive) into the row register and the process repeats,until the last row has been sent.

With the correct memory chip enabled onto bus byte c,data transfer takes place in the following way. For data transfer from the system to the PC, transfer is done via the Am2952A I/O port (data).Memory is addressed as described above.After each memory read, the data read is clocked into the Am2952 "s" register. This

happens on the Planar 1 card. The system then signals to the PC that data is ready for transmission by loading the Am2952A I/O port from pipeline register bits BR6-11 with data 010101.The PC latches this data out using an INP statement to I/O address hex 304,which enables the -PCACK line, thus controlling the output enable on the Am2952, and allowing the data to pass via the PC card, to the PC I/O bus. Once this data is received by the PC,it then reads the data from the data register using an INP to address 303, enabling the -PCREQO line, and enabling the output of the data register. -PCREQO also is input to D0 on MUX B which allows the system to sense when data transfer is made.The next memory element is then accessed and the process repeated.

For data transfer from the PC to the system, the process is essentially the same. The system sends the same return code as before through the Am2952 port. The PC on receipt of this code then does an OUT command to address hex 301, which enables line -PCREQD and clocks the data into the r-register on the Am2952 data port. This is sensed via input D2 on MUX B, allowing the system to enable the output of the data register (-OEbr;PL74) and write the data into memory.

7.6.3 Convolution using a 3x3 Window.

The first stage in the 3x3 convolution process is for the system to indicate the required coefficient set. The program starts in the same way as before,with the system sending the standard return code to the PC. The PC then responds by sending the starting address of the required coefficient set in an OUT statement to hex address 302. This enables line -PCREQA which clocks the starting address into the Am2952A port and signals this to the system via line D1 on MUX B.The system then enables the output of the port via pipeline bit PL72 addressing the Am27S29A-35 row controller load proms. These load the starting address into the Am2940 row DMA controllers, which can then address the Am2148-35 coefficient RAM via address lines A0-7 on the low order controller,and lines A0-1 on the high order controller. The coefficient RAM is then enabled and the first coeffcient clocked

into the convolver. The Am2940 then increments the address and loads the next coefficient into the convolver, and so on until all 9 coefficients are loaded.

Once the coefficients have been loaded, the convolution may start. This again requires the use of special addressing so that 3 bytes may be loaded into the convolver in parallel. The convolution takes place by columns, ie the 3x3 convolution window moves from top to bottom of the image, replacing each center pixel as it goes, before resetting to the top of the image and moving one column to the right.

The addressing scheme for this operation works as follows. Row series 0 is used. The first three values from the column map prom are clocked into the first three column registers.The first value from the row map prom is then clocked into the first row register. Then, with OE3 enabled from the pipeline register, bytes 0,0 from all three memory sections may be clocked into the convolver ,where 0,0 is row0,col0. The next semi row may be clocked when required into the convolver by clocking a new row value into the row registers,after incrementing the row Am2940's. When the bottom of the image is reached, it is necessary to reset to row 0 the row Am2940's and to increment the three columns being loaded by one. To do this, one further column value is clocked from the column map prom into the column registers. This means that the first column is addressed in memory section A, whilst it is still the 0th columns stored in sections B and C. These correspond (looking at the memory map) to image columns 3,1 and 2. In order to ensure that these columns are loaded in the correct sequence bit OE3 is disabled, and bit OE1 is enabled,to give the sequence 1,2,3. The convolution then proceeds down these three columns until the end, when a further value from the column map prom is clocked into the column register. This then gives the first columns stored in sections A,B together with the 0th from section C, ie columns 3,4,2. Enabling OE2 sends these to the convolver in the correct order. This procedure then repeats until the entire image has been covered.

The main program senses the end of a row by detecting the DONE signal from

the Am2940 row controllers, and can detect the last column in either the same way,or via an Am25LS2521 parity detector, which may be set up to detect decimal 171.This corresponds to the last column in a 512x512 image (3x171=513).

Thus the data is loaded into the convolver in parallel mode.The convolver itself is a pipelined device, with a latency of one clock cycle. This means that the convolution result appears after the controllers have advanced by two rows. However,because of the row registers,it is possible to address any row with a latency of up to 4 clock cycles, via pipeline bits ROW0-2 (PL66-8). The center column is, of course, unchanged when writing the result back into the memory. The convolver registers are clocked at the correct times by means of the pipeline bits -E and -OE (PL59-60).

There are two ways of storing the result from the convolution. The resulting image can either be stored in the second frame store,thus preserving the original data, or stored over the original data in the first frame store. If the result is to be stored on top of the original data, then the resulting image must be offset one column to the left and one row up from the current center pixel, to avoid corrupting data which has not yet been used. Selecting one column to the left of the center pixel for the write operation is made easy by the fact that both columns to the side of the center pixel are already addressed in the frame store, and it is just a question of enabling only the section of the memory which contains the low order column and disabling the other two sections. To obtain the correct row address, the system must either address the center pixel row, or one row previous to this, for convolution without and with replacement respectively. Allowing for the one cycle delay in the convolver the correct row to select is either 2 rows previous or 3 rows previous for convolution without and with replacement respectively. This is done via pipeline bits PL66-8, which control which bank of the row address registers is enabled. Both methods have microcode programs.

7.7 Design Summary

We have now looked in some detail at the design and use of a parallel-pipelined convolver. This design was developed from one of the design vignettes mentioned in the last chapter. Pipelining was a major feature of that design vignette, and it has been used a number of times within the system. We will proceed in Chapter 9 to describe the construction and debugging of the system. Further details of the system, such as the pipeline register bits are detailed in Appendix A, as are the values of the look up proms, and the inter-card cabling. We proceed in the next chapter to relate this design to the theoretical analysis of the previous chapters, showing that performance gains of the order of six times over a non-pipelined device are obtained.

7.8 APPENDIX A

7.8.1 Pipeline Register Bits

The pipeline register (microprogram memory) bits are used as follows:

BIT(s)	TITLE	EXPLANATION
PL0	-CCEN	This is the -CCEN input on the Am2910A controller. When low it causes a test to be passed.
PL1-4	I0-3	These are the instruction inputs on the Am2910A controller.
PL5-16	BR0-11	These are the branch inputs to the data input on the Am2910A controller. Pipeline branches take their address from this field. Bits BR0-5 are also connected to the 27S29A load proms for the Am2940 col and I/O DMA chips. Bits BR6-11 are also connected to the 27S29A load proms for the Am2940 row DMA chips.
PL17	-IE	This is the -instruction enable input to the AM2914 priority interrupt controller.When low the values on bits PL18-21 are loaded.
PL18-21	I0-3	These are the instruction inputs to the AM2914.
PL22	POL	This bit controls the output polarity of the 2922 multiplexer.When high the output is noninverted. It is connected to both Am2922's.
PL23-5	A,B,C	These are the 2922 multiplexer control inputs. They are connected to both Am2922's.
PL26	-OEb	This bit controls the output enables on the two Am2922 multiplexers.When low multiplexer B is enabled,when high A is enabled.
PL27	-OE	This bit controls the output enable on the Am2920 instruction register.When low the output is enabled.
PL28	-CNT	This bit when low enables the count on the Am2940 column DMA chip.
PL29-31	I0-2	These are the instruction inputs to the Am2940 column DMA chip.
PL32	-OE	This controls the output enable on the Am2940 column DMA chip.When low the output is enabled.
PL33	-CNT	This bit when low enables the count on the Am2940 row DMA chip.
PL34-36	I0-2	These are the instruction inputs to the Am2940 row DMA chip.
PL37	-OE	This controls the output enable on the Am2940 row DMA chip.When low the output is enabled.
PL38	-CNT	This bit when low enables the count on the Am2940 I/O DMA chip.
PL39-41	I0-2	These are the instruction inputs to the Am2940 I/O DMA chip.
PL42	-OE	This controls the output enable on the Am2940 I/O DMA chip.When low the output is enabled.

Continued Overleaf

Table 7.10: Pipeline Register Bits

Continued from last page

BIT(s)	TITLE	EXPLANATION
PL43	-OE	This is the output enable on the Am27S29A proms. Output is enabled when the bit is low.
PL44-6	-CSA -CSB -CSC	These three bits control the manual enable of each of the three parts of each frame store.When low, the relevent section(s) is enabled.
PL47	-E	This controls the output enable of the analog to digital converter.It is active low.
PL48	-E	This controls the output enable of the digital to analog converter.
PL49-53	-OE1-5	These three bits control address and data mapping. Bits PL49-51 (-OE1-3) when low map the three frame store sections to the three data bus lines.Bits PL52-3 (-OE4-5) map the values in the five column shift registers to the column address lines on the frame stores.
PL54-5	ISEL1-2	These two bits map the data inputs to the convolver to the relevant internal convolver registers.
PL56-7	OSEL1-2	These bits control the output format from the convolver.
PL58	MOD	This bit controls whether the modulus of the convolution output is taken.A high value takes the modulus and outputs an unsigned number.
PL59	-E	This bit controls the input enable to the convolver.It is active low.
PL60	-OE	This bit controls the output enable from the convolver.It is active low.
PL61	FS1/2	This selects the active frame store.When low,frame store one is active.
PL62	R/W	This selects read or write to or from the selected frame store.When low,write is selected.
PL63	-CS	This is the chip select for the Am2418 coefficient store RAM.It is active low.
PL64	-OE	This controls the output enable of the 74LS244 linking bits BR6-11 to the row 27S29A's.It is active low.

Continued Overleaf

Table 7.11: Pipeline Register Bits – continued

Continued from last page		
BIT(s)	TITLE	EXPLANATION
PL65	-SR	This bit when low enables the clock for the column address store shift registers.
PL66-8	ROW0-2	These bits select which output of the row address store shift registers is enabled.
PL69	-CLR	This bit when low enables the 3 Am2920's onto the 3 byte data bus, applying 0's to all bits.
PL70	-I/O	This bit controls the clock input signals to the row and I/O Am2940's and the mapping bits. When low input output mode is selected, with convolve mode selected when high.
PL71	-CEs	This controls the input enable to the Am2952A I/O port, transfer B/A. It is active low.
PL72	-OEbr	This controls the output enable from the Am2952A I/O port, transfer A/B. It is active low.
PL73	-CEs	This controls the input enable to the Am2952A data I/O port, transfer B/A. It is active low.
PL74	-OEbr	This controls the output enable from the Am2952A data I/O port, transfer A/B. It is active low.

Table 7.12: Pipeline Register Bits – continued

PROM Values

Read only memories within the system are programmed with the following data values.

Column Map Prom (Am27S41A-35)	
Addr	Value
000	0
001	0
002	0
003	1
004	1
005	1
006	2
...	...
255	85
256	85
257	85
...	...
509	169
510	170
511	170

Table 7.13: Column Map Proms

Row Map Prom (Am27S41A-35)					
Addr	Value	Description	Addr	Value	Description
000	0	Row series 0	1027	1	Row Series 3
001	0		1028	2	
002	1		1029	3	
003	2		1030	4	
004	3		1031	5	
005	4		1032	1	
...	...		1033	2	
256	255		1034	3	
...	...		1035	4	
512	511		1036	5	
513	1	Row series 1	1037	2	
514	1		1038	3	
515	3		1039	4	
516	5		1040	5	
517	7		1041	6	
...	...		...	...	
641	255		3527	251	
...	...		3528	252	
769	511		3529	253	
770	0	Row series 2	3530	254	
771	0		3531	255	
772	2		3532	251	
773	4		...	...	
774	6		6091	511	
...	...		6092	507	
898	254		6093	508	
...	...		6094	509	
1026	510		6095	510	
			6096	511	

Table 7.14: Row Map Proms

I/O Map Proms (Am27S29A-35) *Prom A*	
Addr	Value
000	0
001	1
002	2
...	...
170	170
171	171

Table 7.15: I/O Map Proms

Prom B and Prom C are identical to Prom A.

Row Load Proms (Am27S29A-35)		
Addr	Value	Notes
000	0	
001	256	
002	9	
003	513	Row series 1 start
004	770	Row series 2 start
005	1026	Row series 3 start
006	0	First coefficient start address
007	9	Second coefficient start address
008	18	Third coefficient start address
...	...	...
118	1008	Last coefficient start address

Table 7.16: Row Load Proms

Column Load Proms (27S29A-35)	
Addr	Value
000	0
001	256
002	512
003	86
004	171

Table 7.17: Column Load Proms

Input/Output Load Prom (27S29A-35)	
Addr	Value
000	0
001	86
002	171

Table 7.18: I/O Load Prom

Vector Map Prom (27S19)	
Addr	Value
000	3

Table 7.19: Vector Map Prom

The Am27S21A-30 mapping proms are loaded as follows

Instruction Map Proms (27S21A-30)		
Addr	Value(Dec)	Instruction
000	000	Get Instruction
001	003	Branch to Instruction
002	010	Coefficient Load Subroutine
003	023	Data Transfer PC to Board subroutine
004	030	Transfer 256^2 Image, PC to Board
005	052	3×3 Convolution on 256^2 - Subroutine
006	080	Snap to FS1
007	100	Direct Camera to TV
008	105	FS1 to TV (512^2)
009	126	3×3 Convolve and Replace on 256^2
010	154	Multi-iteration 3×3 convolution
011	181	Convolve and Threshold on 256^2
012	214	Convolve and add FS2 data on 256^2
013	244	Clear FS1
014	261	Move FS1 to FS2
015	280	Data Transfer Board to PC - Subroutine
016	286	Transfer 256^2 Image Board to PC
017	309	Transfer 512^2 Image Pc to Board
018	331	Transfer 512^2 Image Board to PC
019	354	3×3 Convolution on 512^2 - Subroutine
020	382	FS2 to TV (256^2)
021	403	3×3 Convolution on 256^2
022	409	3×3 Convolution on 512^2
023	415	Load Coefficient set from PC
024	429	Move FS2 to FS1
025	160	10 iterations for convolution
026	165	20 iterations for convolution
027	170	30 iterations for convolution
028	175	40 iterations for convolution

Table 7.20: Instruction Map Prom

Inter Board Connectors							
Connector	Links	Pin#	Signal	Connector	Links	Pin#	Signal
U1	P1/P2	1	+5V	U9	P1/P2	1	GND
		2	Blank			2	Blank
		3	116 Bit0 Am2940R			3	I/O Chip B.bit7
		4	115 Bit1 Am2940R			4	I/O Chip B.bit6
		5	114 Bit2 Am2940R			5	I/O Chip B.bit5
		6	117 Bit3 Am2940R			6	I/O Chip B.bit4
		7	118 Bit4 Am2940R			7	I/O Chip B.bit3
		8	119 Bit5 Am2940R			8	I/O Chip B.bit2
		9	120 Bit6 Am2940R			9	I/O Chip B.bit1
		10	122 Bit7 Am2940R			10	I/O Chip B.bit0
		11	123 Bit8 Am2940R	U10	P1/P2	1	GND
		12	121 Bit9 Am2940R			2	Blank
U2	P1/P2	1	+5V			3	I/O Chip C.bit7
		2	Blank			4	I/O Chip C.bit6
		3	Y4			5	I/O Chip C.bit5
		4	Y3			6	I/O Chip C.bit4
		5	Y5			7	I/O Chip C.bit3
		6	Y2			8	I/O Chip C.bit2
		7	Y1			9	I/O Chip C.bit1
		8	Blank			10	I/O Chip C.bit0
		9	Y0	U11	P1/PC	1	+5V
		10	Blank			2	Blank
U3	P1/P2	1	+5V			3	Bus C.bit0
		2	Blank			4	Bus C.bit1
		3	Blank			5	Bus C.bit2
		4	Row Done			6	Bus C.bit3
		5	Col Done			7	Bus C.bit7
		6	-EOUT			8	Bus C.bit6
		7	Blank			9	Bus C.bit5
		8	Blank			10	Bus C.bit4
		9	I/O Done				
		10	Blank				

Continued Overleaf

Table 7.21: Inter Board Connectors–Section 1

Inter Board Connectors–Continued							
Connector	Links	Pin#	Signal	Connector	Links	Pin#	Signal
				U12	P1/PC	1	GND
						2	Blank
U4	P1/P2	1	+5V			3	-WAITACK
		2	Blank			4	-PCACK
		3	PL8			5	-PCREQD
		4	PL9			6	-PCRFQC
		5	PL7			7	CLK2M
		6	PL10			8	-PCREQA
		7	PL6			9	CLK4M
		8	PL5			10	-PCREQO
		9	Blank	U13	P1/PC	1	Blank
		10	Blank			2	Blank
U5	P1/P2	1	+5V			3	-RESET
		2	Blank			4	Blank
		3	PL28			5	Blank
		4	PL35			6	+RESET DRV
		5	PL29			7	GND
		6	PL34			8	+5V
		7	PL30			9	+5V
		8	PL33			10	GND
		9	PL31	U14	P1/PC	1	+5V
		10	PL32			2	Blank
U6	P1/P2	1	GND			3	PL44
		2	Blank			4	PL45
		3	PL36			5	PL46
		4	PL43			6	PL47
		5	PL37			7	PL48
		6	PL42			8	Blank
		7	PL41			9	PL62
		8	PL38			10	GND
		9	PL39				
		10	PL40				
				U15	P1/PC	1	+5V
						2	Blank
U7	P1/P2	1	GND			3	PL70
		2	Blank			4	C2
		3	PL68			5	C2"
		4	PL52			6	C2'
		5	PL67			7	Blank
		6	PL53			8	CLK2M
		7	PL66			9	Blank
		8	PL65			10	GND
		9	PL61	U16	P2/PC	1	Blank
		10	PL62			2	Blank
		11	PL70			3	-CHIPC
		12	Blank			4	-CHIPB
U8	P1/P2	1	GND			5	-CHIPA
		2	Blank			6	I/O clockC
		3	I/O Chip A.bit7			7	I/O clockB
		4	I/O Chip A.bit6			8	CLK1M
		5	I/O Chip A.bit5			9	I/O clockA
		6	I/O Chip A.bit4			10	CLK1DM
		7	I/O Chip A.bit3				
		8	I/O Chip A.bit2				
		9	I/O Chip A.bit1				
		10	I/O Chip A.bit0				

Table 7.22: Inter Board Connectors–Section 2

Chapter 8

PARALLEL-PIPELINED CONVOLVER – THEORETICAL FRAMEWORK

In this chapter we analyse the design of the Parallel-Pipelined convolver in the light of the analysis in Chapters 4 and 5. We saw in Chapter 4 that the total processing time for an array computer was composed of two parts, that taken to execute the algorithm, and that taken to perform I/O operations. Referring back to equation 4.2:

$$T_a = T_e + T^1_{i/o} + T^2_{i/o} \tag{8.1}$$

The convolver was designed so as to avoid the first stage of I/O operations, $T^1_{i/o}$. This is achieved by the simple expedient of operating solely on stored images. This was a consequence both of the need to reduce processing time, but also of the fact that in the original design criteria for the device, sub-video rate processing was required. This meant that only the data transfer between the memories and the convolver ALU needed to be optimised.

We can classify the resulting architecture as follows: using Handler's second classification we can see that the design utilises an arithmetic pipeline in the convolver, which is a two stage pipelined device. As explained in Chapter 7, a set of registers in the middle of the add-net give the convolver a latency of one clock cycle, but mean that there is twice the throughput from the convolver than would be attainable in a serial device with the same clock frequency.

As well as the arithmetic pipeline, we note that the device has an instruction pipeline, with the registers after the microprogram memory providing a one-stage pipeline to the CPU. This allows the next microprogram instruction to be prefetched and, in the case of jumps in the microprogram not being necessary, doubles the instruction rate. However, because the DMA controllers must be reset when the end of a row or column is reached, some branches in the microprogram are necessary. This does not cause excessive pipeline problems with only two pipeline stages, but would obviously have been significantly less acceptable had the instruction pipeline had more stages.

A third type of pipelining is used in the device in the frame stores, although this is not covered in Handler's classification. Within each frame store, S-access memory interleaving is used (see Chapter 5). By accessing all of the memory modules simultaneously, and then demultiplexing some of the column address bits to decode one memory module onto the data bus at once, we can save a significant portion of the RAM access time. This I/O is performed in column-parallel mode, with the processing array (in this case only a 3×3 array) being loaded with three columns in parallel. Extra parallelism was not used in this case because of the increased hardware cost. However, the column-parallel I/O used is also overlapped with processing in the convolver (as in the MPP) by using phased clock signals. This again saves one memory access, and means that the data is ready to be clocked into the convolver as soon as it is needed.

If we wish to classify the design using Handler's first scheme, we see that $K = 1$, as there is only one PCU in the design. Although this PCU is pipelined, $K' = 1$ (so is omitted), as PCU's cannot be chained together (as one might do in the PIPE computer). Although there is only one convolver in the system, there are nine parallel multipliers in the convolver, so $D = 9$. The value of D' is taken as 3, as banks of three multipliers are pipelined together to load data into the system. Finally, $W = 8$, as the number of bits in each pipeline is eight, with each pipeline

operating on one byte. The value of $W' = 2$ is defined as such because there are two stages to the ALU pipe.

This gives a triple of

$$T(PPC) =< 1, 9 \times 3, 8 \times 2 >$$

Compared with table 5.2, we see that the Parallel-Pipelined convolver is unusual in having $W = 8$, most of the other machines being bit-serial, but that this is offset by the low value of D, designed to keep the hardware cost of the machine down.

If we look at Ramamoorthy and Li's classification, we see that the machine contains unifunction, static pipelines, which do not change their function. Though this is not optimum for pipelining, this approach was necessary because of the cost constraints on the machine, which really prohibited the use of a more adaptable pipeline. Though the instruction pipeline is scalar, that between the frame stores and the convolver may be considered to be a vector pipeline, as three bytes are transferred in parallel along this pipeline.

The pipelines in the Parallel-Pipeline convolver do not use feedback, other than the instruction pipeline, which is a closed loop. Feedforward is employed in the convolver pipeline to allow data to bypass the first stage in the convolver and to load directly to the central pixel register in the convolver. The architecture is a register to register one, as data from the memory is transferred to storage registers in the convolver before it is processed.

The speedup obtained by the convolver pipelines is given by equation 5.3.

$$\frac{(9 \times 512^2) \times 2}{(9 \times 512^2) + 2 - 1} = 2 \text{ times} \tag{8.2}$$

The same applies to the instruction pipeline.

The memory achieves a speedup of three times over a serial memory, either by outputting three memory bytes in parallel, or by using phased clocks on the three memory banks, and then outputting the data serially at three times the standard clock rate.

We look finally at the instruction pipeline, in the light of the analysis in chapter 5. In equation 5.18, we saw that the average number of clock cycles necessary to execute m instructions in an n stage pipeline is:

$$\frac{n+m+1}{n} + mpq\frac{(n-1)}{n} \tag{8.3}$$

where n is the number of stages in the pipeline, equal to 2 in this case. The number of instructions to be performed is m, which, for a 3×3 convolution over a 512×512 pixel image, is approximately two instructions per pixel in the current microprogram, plus about 30 instructions as overheads. Thus:

$$m = (512^2) \times 2 + 30 = 524318$$

The variable p is the probability that a branch instruction occurs in any instruction. In this case $p \approx \frac{1}{2}$, as the system must check after every row or column increment to see if the end of the column has been reached, or if the entire image has been covered. The probability of success of these tests is then the number of pixels in a row (or column), so $q = \frac{1}{512}$.

Applying this data to the analysis of Chapter 5, we can calculate that the average number of cycles taken to complete a convolution over the image will be

$$\text{number of cycles } = (n+m+1)/n + (mpq)(n-1)/n = 262415 \tag{8.4}$$

The average number of instructions completed per pipeline cycle is given by:

$$\text{av no. } = \frac{n}{1+pq(n-1)} = 1.998 \tag{8.5}$$

As can be seen, the use of the two stage pipeline provides a speedup over a serial computer of almost two, as would be expected if branching does not significantly affect the program. Because the controller only has to branch at the end of every row, this proves to be the case here.

To summarise the discussion in this chapter, the design for the Parallel Pipelined convolver has incorporated pipelining in three areas; in the instruction pipeline, in

the arithmetic pipeline (the convolver), and in the memory convolver interface. The first two of these applications both achieve a two-fold increase in speed over a serial computer at little extra hardware cost or complexity. The last application of pipelining, to the memory-convolver interface, achieves a three-fold increase in speed over a serial interface, via the use of interleaving techniques in the memory. By using pipelining, performance gains have been made from the design at a low extra cost in hardware or complexity.

Chapter 9

CONCLUSIONS

9.1 Current Work

This thesis began with an overview of a number of vision algorithms that are considered representative of the next generation of algorithms to be used in industry. It was shown that in these algorithms, many of the calculations of apparently global properties can be performed on a local basis. This has been expressed as Global State from Local Interaction. In this sort of algorithm, simple, often repetitive, local operations are iterated to provide a final global parameter value. We saw here the recurring role of convolution as an operation, even in algorithms which are considered to be highly non-linear. With the current rapid increase in the design and development of large array computers, it seems likely that yet more of the latest algorithms for vision will be expressed in this form.

We proceeded to examine some of the older generation of array computers, all of which are now being used extensively for vision applications. Most of these machines are very similar in principle, with only array machines such as the Connection Machine being significantly different, because of its router network. One other class of machines was examined, namely the pipelined computers such as the Cytocomputer and the PIPE. This type of machine has been far less widely used and developed than the array computer, possibly due to a lack of flexibility in the designs themselves, although the PIPE computer is now finding an increasing market.

The performance of array computers was then discussed. Our conclusions are that even these devices, which appear to be ideally suited to the computational demands of computer vision, are degraded in their performance by the time taken to read image data in and out of the array. In the context of serial processing, pipelining has been proposed as one of the most useful technique to reduce the bandwidth problem between processor array and memory or I/O device. This technique has been used for a number of years in serial computers, and we explored various aspects of its application to vision machines, especially to the design of an otherwise conventional convolver. As long as the main thrust of the hardware design is on producing high speed "number crunchers", with less regard for the overall data flow, we will not achieve the full performance available. A decade ago, real-time machine vision was held back by technology constraints; today it is largely being constrained by the lack of suitable architectures and algorithms.

In chapter 5, we looked in more detail at pipelining in general terms, offering a number of ways of defining a pipelined system. The performance of pipeline systems was calculated, and the potential problems highlighted. We then went on to present three design vignettes, all of which use pipelining to address the bandwidth problems that besets all vision computers. This bandwidth problem affects the vision computation problem in a number of ways: in memory bandwidth, interfacing to the "outside world", performing the calculations at sufficient speed, and transferring the data between the various subsections of the computer.

A number of designs were considered for further study, as gedanken-experiments. The most promising of these was developed further and built with the help of IBM. The detailed design and construction of this device is described. This used a number of the pipelining techniques mentioned earlier in this thesis in its construction, to increase its potential processing speed, at no extra hardware complexity. It is designed to exploit many of the local properties of the algorithms mentioned in Chapter 2. As an industrial project, it was also planned as being a very practical

system, with a number of real target environments, rather than as a more academic exercise, that might perhaps have shown a slightly greater insight into pipelining. However, pipelining was used in the design as shown in Chapter 8 to provide a six fold increase in speed over a simple serial device.

We conclude the work described in the previous Chapters as follows:

1. Many vision algorithms which were thought of as too complex to execute on simple parallel machines may be executed by linear shift invariant operations.
2. The speed of many current vision processors which perform such linear shift invariant operations is restricted by the I/O bandwidth of the processor.
3. Pipelining may be a solution to many of these I/O problems.
4. The detailed design of such a parallel pipelined vision processor is described. This provides a six-fold increase in performance over a similar parallel processor.

9.2 Future Work

We concentrate first on work following on directly from that presented in previous chapters. The Parallel–Pipelined convolver design that we described in Chapter 7 was built using the Electronic Design System at IBM U.K. Laboratories between March and December 1986. This involved the construction of three printed circuit boards, each with ground, voltage and two signal planes. The first card, the Planar 1, described in section 7.4, was delivered in September 1986, with the other cards following in November and December. On their arrival in Oxford, I assembled all three cards, using I.C. sockets for all devices to help with debugging.

The cards were then debugged in the following order over the first four months of 1987.

- PC Card
- Planar 1 Card - Microcontroller and Switching

- Planar 2 Card - DMA Controllers
- Planar 2 Card - Frame Stores
- Planar 1 Card - Convolver

The system is now 90% operational, with the remaining problems still to be tackled in the software microprogram for the system.

Simulation

It had been intended that the system would be simulated on IBM's E.D.S. computer. This would have allowed the main logical faults to be tracked before construction, and, perhaps more importantly, would have allowed me to debug and develop the system software. However, due to unforseen circumstances at IBM, there was not sufficient development time to perform any simulation (hardware or software) on the system.

This left the somewhat complex task of debugging three cards with:

- 317 Integrated Circuits
- 438 Discrete (mainly passive) devices.
- Approximately 10,000 interconnections

This debugging had then to be performed on the device as it was built. To facilitate this, the system was debugged in small sections, with each section being checked (e.g. the DMA controllers) before the next logical device was inserted (e.g. the memory). Perhaps the main problem in debugging the device in this fashion was that the software could not be guaranteed as correct, so it often took some time to actually isolate the cause of a problem before a solution could be attempted.

Fortunately, in most cases, it has been possible to check that the system is logically correct in its operation, although refinement of the microcode may prove necessary. I detailed the errors that I discovered in a log, which is summarised in the next section.

Errors

I had expected that the PC card would prove the most troublesome to debug, as it contained hybrid analogue/digital circuitry and all of the system interfaces to the "outside world". This indeed proved to be the case, with most of the problems on this card. The planar 1 card proved the next most difficult, due mainly to the sequencer. This in particular was made more difficult because of the ambiguity between a potential hardware or software error. After these cards, the second planar card proved relatively simple to debug.

The problems found are summarised by category in table 9.1

	Problem Type			
	Construction	Design	Logical	Software
Errors	6	20	1	4

Table 9.1: Errors Found In Debugging

All of the design problems that were found were minor in nature, usually due to an inverted signal, a wrong value for a resistor, or similar problem. These have all been corrected.

The software problems that have been identified so far have not, on the whole, been fixed yet because of the expense and time involved in coding new Proms. Three of these will prove easy to correct. The fourth software problem is in the handshaking protocol between the PC and the system when attempting frame store transfers. This needs significant tightening, as some data bytes are being missed when transfer is attempted at the moment.

The construction problems are, with one exception, of a trivial nature, being confined to the wrong pitch holes on PC cards, etc. The exception to this is a crosstalk problem on the Planar 1 card. This affects the correct operation of the microcode sequencer when video rate data is being transferred on byte c of the data bus, causing false branches to occur. It appears to be attributable to crosstalk between this bus line and the ground plane on the planar 1 card. This sort of

problem can prove difficult to debug (it took some three weeks to track it down), and even more problematical to correct. However, this can be cured with more careful power supply routing on the planar 1 card.

The one logical design problem occurs with the coefficient loading to the convolver registers. These are designed to load coeffcients in the same way as data, i.e. loading three bytes in parallel. However, due to an omission on my part, there is only one coefficient store RAM on the system, attached to byte c of the data bus. This obviously means that coefficients cannot be loaded in the designed manner. The best solution to this problem is not, in fact to add two further coefficient RAMS, but instead to use some of the spare space in the frame stores to store coefficients. Using this method, there is room for over 3500 nine coefficient sets. This solution also has the benefit that no hardware changes need to be done, and the correction can be done entirely in the software.

A simple rebuild of the circuit is planned at IBM's T.J.Watson Research Center at Yorktown, U.S.A., correcting all of the faults discovered in the design, and finishing the software coding. It is not anticipated that this will be a major undertaking, as no major faults have been uncovered in the design.

Summary

We have described the debugging of a hardware realisation of one of the design vignettes presented in Chapter 6. The hardware of this device is now operational, and future work will entail debugging the software and then benchmarking the system on some of the algorithms which were discussed in Chapter 2. The one problem with the hardware that may need rework is the ground plane crosstalk. This may involve redesigning the physical layout of the planar 1 card (although not the logical design).

Areas where improvement could be made on a second design iteration are as follows:

- More test points in the analogue part of the PC card and improve the overall testability.
- Separate control bus, instead of shared control and data buses.
- Use fast EPROMS for microprogram memory and look-up tables. These devices were not available in the required speed versions when the original design was done.
- Use larger memory devices (again, these were not available at the time of the design)
- Redesign the convolver section if possible using more compact devices.
- Map system to VME bus as well as PC bus to increase versatility and also for higher speed data transfer.

Some of the above are modifications that would normally be made in the course of a "phase two" design. Some of the others would, I believe have been trapped during circuit simulation. Indeed, of the errors mentioned previously, I think that some 25 out of 31 should have been caught with full simulation, a lesser number with partial simulation. As a design exercise, it has emphasised a number of points about practical debugging of complex systems, such as the inadvisability of debugging both hardware and software at the same time.

A number of points have arisen from this design and build experience. The current system needs further software development, as well as a few minor hardware changes, and has highlighted the need for software simulation (at least) in future developments. Any second generation version of the design would obviously use the current technology in memory devices (available at four times the capacity of the current RAMs), and it would also be likely that the convolver chip count could be significantly reduced using the latest devices. Alternatively, using higher density devices in the convolver could allow for the adoption of a larger window size. These are in addition to the ideas discussed in this section. These modifications would

provide a more unified and adaptable design, as well as fulfilling the usual electronic design "goals" of "smaller, faster and cheaper".

We have covered the developments that follow directly from the work done in this thesis. What then, of the ideas that have occurred in parallel with the work already done? There can be no doubt, I believe, that the concept of global state from local interaction is going to heavily influence the development of vision algorithms and vision machines over the next decade at least. This really stems from the observation that the only way of currently providing *orders of magnitude* increases in computing power is to use massively parallel networks. Although there are fixed technology limits to this approach, this is likely to be the way forward until the advent of either the optical computer or the bio-computer. The latter of these at least will almost certainly be based upon large three-dimensional networks of computing elements, so algorithms will still need to execute largely at a local level to provide the necessary speed.

With this continuing emphasis on repetitive local computation, the hardware designer will still be faced with the problem of providing the necessary data bandwidth at all of the potential bottlenecks in his system. Here, the concepts of pipelining that we have examined over the last chapters will be just as relevant, if the designer is to be able to extract the full potential performance of his machine. Many of the methods mentioned will no doubt be used, as well as new techniques; these may be necessary if, for instance, parallel output cameras are developed.

It seems important, for the future, that further research into pipelining and other techniques for speeding data flow is performed. One possible way of doing some of this work is with a modular computing system. I would envisage such a system as being based around processing units, each consisting of a simple processor, such as an 8086 or similar device, and a small amount of local memory. There would be data bus connections in the four compass directions. The units would be designed in such a way that they could be plugged together as a linear pipeline, a square

array, or any other desired topology.

The chief design problem with the above would be in making the data bus sufficiently flexible to allow easy reconfiguration of the system. However, it would offer the opportunity to experiment with different algorithm classes, to find the most efficient way of passing the data through the system, and could allow experiments with two and even three dimensional pipelining. Clocking would be performed on a SIMD basis, using the standard instruction set of the microprocessor used.

It is also important that research should be performed into very high speed data links. Obviously, the best way of linking an image memory and a large processing array is image–parallel. This is prohibitively expensive at the moment because of the complexity of connections, and the number of links needed. However, it might be possible to use a high speed optical link, with a switching/multiplexing network at each end, working in a similar way to concurrent memory access.

Data links between the chips that make up the elements of such a processing array will also play an increasingly important role. As integrated circuit technology progresses, the number of processing elements and memory on a single chip will increase dramatically. This makes intra-chip communications far easier, but makes inter-chip communications increasingly difficult. Presently, there are only two solutions for this problem; very large chip pin-outs or making the inter-chip connections very simple, and hence often slow and inefficient.

One further possibility exists in Data-Flow architectures, which are, as yet, infrequently used in vision computation. By perhaps hybridising some of the concepts of data flow, such as attaching tokens to data packets, with some of the ideas of parallel processing, a new type of architecture might be created. At the lowest level, this concept would be of little use, because of the huge amounts of data involved. However, at a higher representitive level, with tokens identifying objects, edges, or other primitives, and perhaps acting automatically to reconfigure processing networks to the most efficient mode to suit the data, a very powerful architecture is

possible.

We can see that the latest generation of vision algorithms present numerous problems for real time computation. The short, and possibly long, term solution to these problems is to use very large arrays of computing elements. This leads to an I/O bandwidth problem, instead of the original computing bandwidth bottleneck. We have investigated a number of solutions to this problem, and have built a computer that utilises some of these ideas. We present further ideas for research in the field. In the near future, intelligent control by vision processing will become a possibility – the challenge facing the hardware designer to turn that possibility into reality is a severe one.

Bibliography

[AMD] Advance Micro Devices Bipolar Microprocessor Logic and Interface, Am2900 Family Data Book. AMD, 1985.

[AM8158] Advance Micro Devices Am8158 Data Sheet. AMD, U.S.A. 1986.

[Ballard & Brown 82] Ballard, D.H and Brown, C.M. Computer Vision, Prentice-Hall, 1982

[Batcher 80] Batcher, K. "Design of a Massively Parallel Processor", *IEEE Trans. on Computing*, **C-29**, no. 9, 1980.

[Batcher 82] Batcher, K., "Bit-Serial Parallel Processing Systems", *IEEE Trans. Comp.* vol.**C-31**, No.5, 1982.

[Blake 85] Blake, A., "Boundary Conditions for Lightness Computation in Mondrian World", *Comput. Vision, Graphics and Image Process.* **32**, 1985.

[Blake and Zisserman 85] Blake, A. and Zisserman, A. "Using Weak Continuity Constraints". Report **CSR-186-85**. Department of Computer Science, University of Edinburgh. 1985.

[Blake and Zisserman 87] Blake, A. and Zisserman, A. Visual Reconstruction, *Draft*, January, 1987.

[Brady and Horn 83] Brady, J.M. and Horn, B.K.P., "Rotationally Symmetric Operators for Surface Interpolation". *Computer Vision, Graphics and Image Processing***22** pp.70-94, 1983.

[Brooks and Horn 85] Brooks, M.J. and Horn, B.K.P, "Shape and Source from Shading", *M.I.T A.I. memo* **820**, Jan 1985.

[Burt 81] Burt,F.P, "Fast Filter Transforms for Image Processing", *Comp. Graphics and Image Proc.*, pp.20-51, 1981.

[Buxton and Murray 85] Buxton, B.F., Murray, D.W., Buxton, H., and Williams, N.S., " Structure from Motion Algorithms for Computer Vision on an SIMD Architecture." *Computer Physics Communications* **37** 1985.

[Buxton and Buxton 84] Buxton, B.F., Buxton, H., and Stephenson, B.K., "Parallel Computations of Optic Flow in Early Image Processing" *IEEE Proc.* vol.**131**, Pt.F, No.6, 1984.

[Canny 85] Canny,J. "Finding Edges and Lines in Images", *M.I.T. Ph.D. Thesis*, 1985.

[Chellappa 87] Chellappa, R. and Frankot, R.T., "A Method for Enforcing Integrability in Shape from Shading Algorithms", Proc. 1st Int. Conf. on Computer Vision, London, June 8-11, 1987.

[Clowes 71] Clowes, M. "On Seeing Things", *Artificial Intelligence* **vol. 2**, no. 1, 1971.

[Comon and Robert 87] Comon, P., and Robert, Y. "A Systolic Array for Computing BA^{-1}" *IEEE Trans. Acoustics, Speech and Signal Proc.* vol. **ASSP-35**, no. 6, 1987.

[Computer 86] Special Issue of *Computer* on Dataflow Architectures, March 1986.

[Drumheller and Poggio 86] Drumheller, M. and Poggio, T. "On Parallel Stereo", *Robotics and Automation Conf.* pp1439, 1986.

[Duff 76] Duff, M.J.B, "CLIP 4. A Large Scale Integrated Circuit Array Parallel Processor". *3rd Joint Int. Conf on Pattern Recognition*, 1976.

[Falk 76] Falk, H. "Reaching for the Gigaflop", *IEEE Spectrum*, **13(10)**, pp.65-70, 1976.

[Fountain 81] Fountain, T.J., "Towards CLIP 6 - An Extra Dimension." IEEE CAPIDM Conf. 1981.

[Geman and Geman 84] Geman, S. and Geman, D. "Stochastic Relaxation, Gibbs Distributions and the Bayesian Restoration of Images", *IEEE Trans. PAMI* **PAMI-6**, no.6, Nov. 1984.

[Gerritsen 83] Gerritsen,F.A. "A Comparison of the CLIP4, DAP and MPP Processor Array Implementations", *Computer Structures for Image Processing*, Academic Press, London (1983).

[Granlund 83] Granlund, G.H. and Arvidsson, J. "The GOP Image Computer", in Fundamentals of Computer Vision–An Advanced Course, ed. O. Faugeras, C.U.P, 1983.

[Grimson 81] Grimson, W.E.L , *From Images to Surfaces*, M.I.T Press, 1981.

[Handler 77] Handler,W. "The Impact of Classification Schemes on Computer Architecture", *Proc Int Conf. on Parallel Proc.*,pp7-15 (1977).

[Hannaway 84] Hannaway,W., Shea,G., and Bishop, W.R. "Handling Real Time Images Comes Naturally to Systolic Array Chip". *Electronic Design*, Nov. 15, 1984.

[Haralick and Sternberg 86] Haralick, R.M., Sternberg, S.R., and Zhuang, X. "Grayscale Morphology", Proc. IEEE Comp. Soc. Conf. on Computer Vision and Pattern Recognition, June 22-26, 1986.

[Hellerman 67] Hellerman, H. *Digital Computer System Principles*, McGraw-Hill, New York, 1967.

[Hildreth 83] Hildreth, E.C. *The Measurement of Visual Motion,* M.I.T Press, 1983.

[Hillis 85] Hillis, W.D, *The Connection Machine*, M.I.T. Press, 1985.

[Hockney and Jesshope 81] Hockney, R.W. and Jesshope, C.R., Parallel Computers, Adam Hilger, Bristol, U.K. 1981.

[Horn 86] Horn,B.K.P , *Robot Vision*, ,1986.

[Horn and Schunck 81] Horn, B.K.P and Schunck, B.G "Determining Optical Flow", in *Computer Vision*, ed. Brady, J.M. North Holland, 1981.

[Horn 74] Horn, B.K.P., "Determining Lightness from an Image", *Computer Graphics and Image Processing*, vol.**3**, pp277-299, 1974.

[Huang 79] Huang,T.S., Yang,G.J., and Tang,G.Y. "A Fast Two Dimensional Median Filtering Algorithm". IEEE Trans. Acoustics, Speech and Signal Proc. Vol.**ASSP-27** No.1, 1979.

[Huffman 71] Huffman, D. "Impossible Objects as Nonsense Sentances". in *Machine Intelligence 6*, eds. Meltzer, B. and Michie, D., Edinburgh University Press, 1971.

[Hwang and Briggs 84] Hwang,K and Briggs,F.A. *"Computer Architectures and Parallel Processing"*, M^{c}Graw Hill (1984).

[IBM] IBM Personal Computer XT Technical Reference Manual. Serial Number 6936833. 1983.

[Ikeuchi and Horn 81] Ikeuchi, K. and Horn, B.K.P "Numerical Shape from Shading and Occluding Boundaries", in *Computer Vision*, ed Brady,J.M North Holland, 1981.

[Kamal 74] Kamal et al, *IEEE Trans. on Computers*, 1974.

[Kent] Kent, E. PIPE - A Specialised Computer Architecture For Robot Vision.

[Kittler 83] Kittler,J. "On the Accuracy of the Sobel Edge Detector", *Image and Vision Computing*, **vol.1** no.1, Feb. 1983.

[Knight 83] Knight,T.F., "Design of an Integrated Optical Sensor with On-Chip Preprocessing". *Ph.D Thesis*, MIT (1983).

[Knuth 75] Knuth, D.E, and Rao, G.S, "Activity in Interleaved Memory", *IEEE Trans. on Computers*, **C-24**, no. 9,pp943-4, Sept. 1975.

[Land and McCann 71] Land, E.H. and McCann, J.J., "Lightness and Retinex Theory", *Journal of the Optical Society of America*, vol.**61** no.1. 1971

[Marr 82] Marr,D. "Vision", W.H.Freeman & Co, San Francisco (1982).

[Marr and Hildreth 79] Marr, D. and Hildreth,E. , "Theory of Edge Detection", M.I.T. AI Memo no.**518**, April 1979.

[Marr and Poggio 82] Marr,D. and Poggio,T. "Stereo", in *Computer Vision,* eds. Ballard, D.H. and Brown, C.M. Prentice-Hall, pp82, 1982.

[Mick 80] Mick,J. and Brick,J. "Bit-Slice Microprocessor Design" McGraw-Hill, N.Y. 1980.

[Nagel 84] Nagel, H.H., "Spatio-Temporal Modelling Based on Image Sequences". in Proc. Int. Symposium on Image Processing and its Applications. Tokyo, 1984.

[Narenda 78] Narenda, P.M. "A Separable Median Filter for Image Noise Smoothing". Proc. IEEE Conf. on Pattern Recognition and Image Processing, pp.137-141, 1978.

[Nishihara 81] Nishihara,H.K, and Larson,N.G. "Towards a Real-Time Implementation of the Marr and Poggio Stereo Matcher", *SPIE Vol* **281**, Techniques and Applications of Image Understanding (1981).

[Oflazer 83] Oflazer, K. "Design and Implementation of a Single-Chip 1-D Median Filter". *IEEE Trans. Acoustics, Speech and Signal Processing*, Vol.**ASSP-31**, No.5, 1983.

[Page 83] Page, I., DisArray : A 16 × 16 RasterOp Processor., Eurographics '83, Elsevier Science Publishers, B.V., (North Holland), 1983.

[Parkinson 83] Parkinson, D., "The Distributed Array Processor (DAP)". *Computer Physics Communications* **28**, 1985.

[Pollard 85] Pollard, S.B., Mayhew, J.E.W., and Frisby, J.P., "PMF; A Stereo Correspondence Algorithm Using a Disparity Gradient Limit", *Perception*, vol. **14**, 1985.

[Pratt 78] Pratt, W.K., Digital Image Processing, J. Wiley and Sons, N.Y. 1978.

[Ramamoorthy and Li 77] Ramamoorthy, C.V and Li,H.F, "Pipeline Architectures", *ACM Computing Surveys*, **Vol. 9**, no. 1, pp61-102, Mar. 1977.

[Reddaway 73] Reddaway, S."DAP - A Distributed Processor Array", *First Annual Symposium on Computer Architecture,* Florida, pp61, 1973.

[Rosen 76] Rosenfield A., Hummel R.A., and Zucker S.W., "Scene Labelling by Relaxation Operators", *IEEE Trans. on Systems, Man and Society*, vol **6**, 1976.

[Scott 86] Scott, G.l., "Local and Global Interpretation of Moving Images." D.Phil Thesis, University of Sussex, 1986.

[Serra 82] Serra, J.P., " Mathematical Morphology and Image Analysis", Academic Press, London. 1982.

[Shooman 60] Shooman, W. "Parallel Computing with Vertical Data", *AFIPS Conf. Proc.*, pp 110-5, 1960.

[Slotnick et al 60] Slotnick, D.L, Borck, W.C, and McReynolds, R.C, "The SOLOMON Computer", *AFIPS Conf. Proc.* **22**, pp97-107, 1960.

[Sobel et al 69] Tennenbaum, J.M., Kay, A.C., Binford, T., Falk, G., Feldman, J., Grape, G., Paul, R., Pringle, K., and Sobel, I., "The Stanford Hand-Eye Project", *Proc. IJCAI*, pp 521-526, 1969.

[Sternberg 78] Sternberg, S. "Cytocomputer Real Time Pattern Recognition", *Proc. 8th Automatic Imagery Pattern Recognition Symposium*, pp205, 1978.

[Tanimoto 84] Tanimoto, S.L., "A Hierarchical Cellular Logic for Pyramid Computers.", *J. Parallel and Distributed Computing*, **1**, 1984.

[Terzopoulos 84] Terzopoulos, D. "Multiresolution Computation of Visible Surface Representations", *M.I.T Ph.D.*, 1984.

[Ullman 79] Ullman, S. "The Interpretation of Visual Motion", M.I.T. Press, 1979.

[Waltz 75] Waltz, D. "Understanding Line Drawings of Scenes with Shadows ", in Psycology of Computer Vision, ed. Winston, P.H., Mcgraw-Hill, N.Y.

[Witkin 81] Witkin, A.P., "Recovering Surface Shape and Orientation From Texture", *Artificial Intelligence* **17**, 1981.